Comparison of Sequential and Parallel Mean Shift Clustering Algorithms

Francesco Gigli
francesco.gigli@edu.unifi.it

October 29, 2024

Abstract

This project investigates the performance optimization of the Mean Shift clustering algorithm through three different implementations: a sequential approach, a parallel implementation using OpenMP, and a GPU-based version leveraging CUDA. The objective is to analyze the efficiency of each method in handling large-scale datasets, focusing on execution speed, scalability, and resource utilization. The study aims to demonstrate how parallel computing techniques and GPU acceleration can address the computational demands of clustering in high-dimensional spaces, while highlighting the potential challenges and trade-offs involved in these approaches.

1 Introduction

The Mean Shift algorithm is a versatile non-parametric clustering technique that identifies cluster centers by shifting data points iteratively toward regions of higher data density [1]. Unlike many clustering algorithms, Mean Shift does not require specifying the number of clusters in advance. Instead, it discovers clusters by detecting the modes (peaks) in the underlying data distribution.

This report presents an implementation of the Mean Shift algorithm applied to various datasets, in both sequential and parallel configurations. We examine two types of kernel functions used to compute the density around each point: the flat (uniform) kernel and the Gaussian kernel. The flat kernel assigns equal weights to all points within a fixed distance (bandwidth) from the target point, while the Gaussian kernel gives higher weights to points closer to the target, based on a Gaussian distribution [2].

1.1 Mean Shift Algorithm

The Mean Shift algorithm iteratively updates the position of each data point $\mathbf{x}_i$ by moving it toward the mean of all nearby points, where the "neighborhood" is determined by a kernel function K and a bandwidth parameter h. This process can be mathematically described as:

$$\mathbf{x}_i^{(t+1)} = \frac{\sum_{j=1}^{n} K\left(\frac{\|\mathbf{x}_j - \mathbf{x}_i^{(t)}\|}{h}\right)\mathbf{x}_j}{\sum_{j=1}^{n} K\left(\frac{\|\mathbf{x}_j - \mathbf{x}_i^{(t)}\|}{h}\right)} \tag{1}$$

Where:

- $\mathbf{x}_i^{(t)}$ is the position of the i-th point at iteration t,
- n is the total number of data points,
- h is the bandwidth (window size) that controls the range of influence of the kernel,
- K is the kernel function, which determines how neighboring points influence the shift of $\mathbf{x}_i$,
- $\|\cdot\|$ represents the Euclidean distance.

The algorithm repeats this process until all points converge to stable positions, typically when the shift between consecutive iterations is smaller than a predefined threshold ϵ.

1.2 Flat and Gaussian Kernels

The two kernel functions used in this implementation — flat and Gaussian — determine how the weights of neighboring points are computed.

1.2.1 Flat Kernel

The flat (or uniform) kernel gives equal weight to all points within the bandwidth h, and zero weight to those outside this range. It is defined as:

$$K(u) = \begin{cases} 1 & \text{if } u < 1 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Substituting this kernel into Equation 1, the update rule simplifies to:

$$\mathbf{x}_i^{(t+1)} = \frac{\sum_{j:\|\mathbf{x}_j - \mathbf{x}_i^{(t)}\| < h} \mathbf{x}_j}{\left|\left\{j : \|\mathbf{x}_j - \mathbf{x}_i^{(t)}\| < h\right\}\right|} \tag{3}$$

Here, the new position of $\mathbf{x}_i$ is simply the arithmetic mean of all points within the bandwidth h, regardless of their distance from $\mathbf{x}_i$ as long as they are within the window.

1.2.2 Gaussian Kernel

The Gaussian kernel, on the other hand, assigns weights based on the distance of each neighboring point from $\mathbf{x}_i$, with closer points having a higher influence. The kernel function is defined as:

$$K(u) = \exp\left(-\frac{u^2}{2}\right) \tag{4}$$

Using the Gaussian kernel in Equation 1, the update rule becomes:

$$\mathbf{x}_i^{(t+1)} = \frac{\sum_{j=1}^{n} \exp\left(-\frac{\|\mathbf{x}_j - \mathbf{x}_i^{(t)}\|^2}{2h^2}\right) \mathbf{x}_j}{\sum_{j=1}^{n} \exp\left(-\frac{\|\mathbf{x}_j - \mathbf{x}_i^{(t)}\|^2}{2h^2}\right)} \tag{5}$$

In this case, the new position of $\mathbf{x}_i$ is a weighted average of its neighbors, with closer points contributing more to the mean. This makes the Gaussian kernel smoother and more sensitive to the local structure of the data compared to the flat kernel.

2 Algorithm Implementation

The Mean Shift algorithm was implemented sequentially, applying two different kernels — flat and Gaussian — to datasets of varying sizes. The process is divided into four main phases:

2.1 Algorithm Phases

1. **Data Loading:** The data points are loaded from CSV files for each dataset size. These points form the input for the clustering algorithm.

2. **Clustering with the Flat Kernel:** The Mean Shift algorithm is executed using the flat kernel with a bandwidth of 20.0 and epsilon of 1.0. In each iteration, data points are shifted toward the mean of their neighbors, with all points within the bandwidth being treated equally.

   As shown in Figure 1, the blue circles represent the influence of the bandwidth around each point, while the red "X" marks indicate the centroids calculated after each iteration. Over time, the points converge toward high-density centers, forming distinct clusters.

3. **Clustering with the Gaussian Kernel:** The same dataset is reloaded, and the algorithm is applied with the Gaussian kernel, using a bandwidth of 1.0 and epsilon of 0.2. In this case, points closer to the target exert more influence due to the Gaussian weighting, leading to smoother convergence.

   Figure 2 illustrates the density contours generated during the Gaussian kernel execution. As the iterations progress, points converge toward the centroids, as highlighted by the red "X" marks.

4. **Results:** For each kernel, the total execution time and the number of clusters identified are recorded and saved to an output file. These results provide insight into the efficiency and behavior of both kernel functions across different dataset sizes.

2.2 Clustering with the Flat Kernel

The flat kernel assigns equal influence to all points within a certain bandwidth. As the algorithm progresses, data

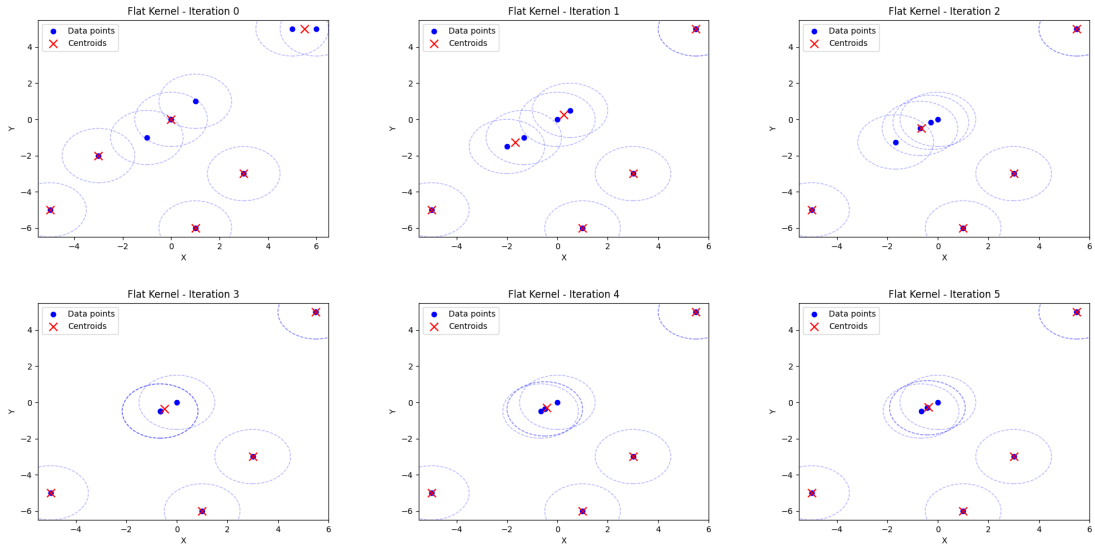

Figure 1: Clustering process with the Flat Kernel, shown from Iteration 0 to Iteration 5. The blue points represent data points, and the red Xs indicate centroids. The dashed circles show the bandwidth influence around each point.

points (blue) gradually move toward the centroids (red Xs) over several iterations, as shown in Figure 1.

1. **Iteration 0:** Initially, each blue point represents a data point, with dashed circles showing the bandwidth around them. The centroids are calculated based on the current positions of the points, and the shifting process begins.

2. **Iterations 1-3:** Points in overlapping bandwidth areas move closer together, converging toward shared centroids. The central points are particularly drawn toward a common centroid as their influence zones overlap.

3. **Iterations 4-5:** By the final iterations, the points have mostly converged to stable centroids. The circles shrink as the points approach the centers of the clusters, and the process nears full convergence by iteration 5.

Figure 1 illustrates the full progression from iteration 0 to iteration 5, with the points steadily converging toward their respective clusters.

2.3 Clustering with the Gaussian Kernel

The Gaussian kernel assigns greater weight to points that are closer, leading to smoother transitions as points shift toward centroids. The clustering process with the Gaussian kernel is shown in Figure 2, where the data points (blue) gradually move toward the centroids (red Xs) over several iterations.

1. **Iteration 0:** At the start, the points are spread across the space, with contour lines showing the influence of the Gaussian kernel. Centroids are computed based on the weighted influence of nearby points, and the shifting process begins.

2. **Iterations 1-3:** Points in closer proximity begin to cluster, moving toward shared centroids. The contours become tighter around the high-density regions as the points shift together.

3. **Iterations 4-5:** By the later iterations, the points have almost converged to their respective centroids, with the contours indicating well-defined clusters. The centroids stabilize as the points reach high-density centers.

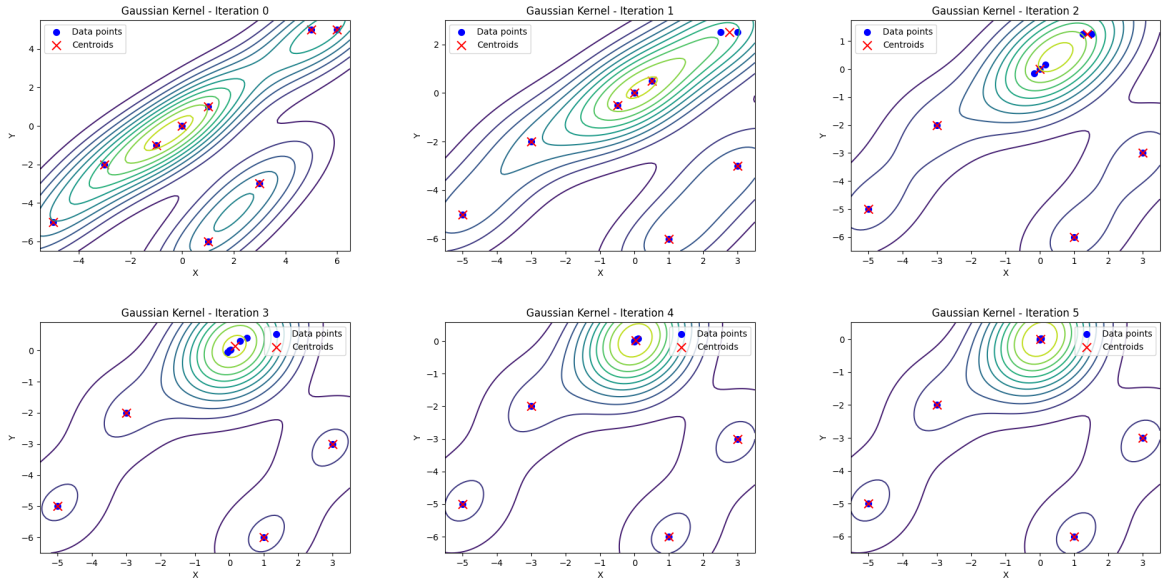

Figure 2: Clustering process with the Gaussian Kernel, shown from Iteration 0 to Iteration 5. Blue points represent the data points, red Xs are the centroids, and the contour lines show the influence of the Gaussian kernel.

Figure 2 shows how the clustering process evolves from iteration 0 to iteration 5, with points converging smoothly toward their final positions.

2.4 Convergence Criteria

Convergence for each point is determined by comparing the shift distance to the threshold ϵ. If the distance moved in the current iteration is less than ϵ, the point is considered to have converged:

$$\text{Converged if } \|p_i^{(t+1)} - p_i^{(t)}\| < \epsilon \tag{6}$$

The convergence criterion applies to both the flat and Gaussian kernels; however, the way in which each point's new position $p_i^{(t+1)}$ is calculated differs based on the kernel type. The flat kernel assigns equal weights to all neighbors within the bandwidth, while the Gaussian kernel assigns weights based on the distance of each neighbor, giving closer points more influence.

3 Mean Shift Code Implementation

The Mean Shift algorithm was implemented to support three versions: sequential, parallel (using OpenMP), and CUDA-based. The core logic is centralized in the `MeanShift.cpp` file, which serves as the foundation for both the sequential and parallel (OpenMP) implementations. This approach ensures consistency with minimal code duplication. The parallel version adds OpenMP pragmas to distribute tasks across multiple CPU cores, while the CUDA version, requiring a different structure, is implemented in a separate file.

3.1 Sequential Mean Shift Implementation

In the sequential version, each point is adjusted based on the weighted average of its neighboring points. This process repeats until either all points converge or a maximum number of iterations is reached. During each iteration, the algorithm calculates the shift for each point, checks if it meets the convergence threshold ϵ, and marks it as converged if so. Once all points stabilize, the algorithm

counts the final clusters.

Algorithm 1 Sequential Mean Shift Clustering

Require: Dataset $\mathcal{P}$, bandwidth h, threshold ϵ, kernel K, max iterations max_iter
Ensure: Final positions of points
  Load data points
  while not all points are converged and $iteration < max\_iter$ **do**
    for each point p_i **do**
      Calculate new position p_{new} based on neighbors
      Check if p_i has converged
    end for
    Increment iteration
  end while

3.2 Parallel Mean Shift Implementation

The parallel version uses OpenMP to distribute the workload. Each thread calculates shifts for different points simultaneously, with synchronization via critical sections or atomic operations to prevent race conditions when updating shared variables [3].

Algorithm 2 Parallel Mean Shift Clustering (OpenMP)

Require: Dataset $\mathcal{P}$, bandwidth h, threshold ϵ, kernel K, max iterations max_iter, threads n_{threads}
Ensure: Final positions of points
  Load data points
  while not all points are converged and $iteration < max\_iter$ **do**
    #pragma omp parallel for num_threads(n_{threads})
    for each point p_i **do**
      Calculate new position p_{new} based on neighbors
      Check if p_i has converged
    end for
    Increment iteration
  end while

3.2.1 Critical Section Synchronization

Critical sections ensure only one thread updates shared variables at a time, reducing conflicts but potentially impacting performance with more threads.

```
#pragma omp parallel for
for (int i = 0; i < points.size(); ++i) {
    if (weight > 0) {
        #pragma omp critical
        {
            sum_x += points[i].x * weight;
            sum_y += points[i].y * weight;
            sum_weight += weight;
        }
    }
}
```

Listing 1: Critical Section Synchronization

3.2.2 Atomic Operation Synchronization

Atomic operations allow threads to update variables individually, enhancing performance. This method is activated with the use_atomic flag.

```
#pragma omp parallel for
for (int i = 0; i < points.size(); ++i) {
    if (weight > 0) {
        #pragma omp atomic
        sum_x += points[i].x * weight;
        #pragma omp atomic
        sum_y += points[i].y * weight;
        #pragma omp atomic
        sum_weight += weight;
    }
}
```

Listing 2: Atomic Operation Synchronization

The choice between critical sections and atomic operations depends on the specific synchronization needs and hardware; atomic is generally faster when updates are frequent.

3.3 CUDA Mean Shift Implementation

The CUDA version of the Mean Shift algorithm optimizes computation by distributing work across a GPU, allowing many points to be processed simultaneously [4]. Unlike the CPU-based parallel version, which uses OpenMP for multi-threading, the CUDA implementation leverages GPU threads to perform Mean Shift calculations, with each thread calculating the new position of one point. This approach is particularly efficient for large datasets, as the GPU can handle thousands of threads concurrently.

Key differences from the parallel and sequential versions:

- **CUDA Kernels**: Custom kernel `mean_shift_kernel` calculates mean shifts for each point on the GPU.

- **Device Memory Management**: Data is transferred between CPU and GPU memory.
- **Convergence Check**: Convergence is verified on the CPU after each kernel execution.

Algorithm 3 CUDA Mean Shift Clustering

Require: Dataset $\mathcal{P}$, bandwidth h, threshold ϵ, kernel K, max iterations max_iter
Ensure: Final positions of points
Copy data points to device memory
while not all points are converged and $iteration < max\_iter$ **do**
Launch `mean_shift_kernel` to calculate new positions on the GPU
Copy convergence data back to host
Check if all points have converged
Copy new positions to device memory for the next iteration
Increment iteration
end while
Copy final points back to host

The CUDA implementation achieves parallelism at a finer level than the CPU-based parallel version, making it particularly effective for large datasets. The pseudocode highlights essential steps of memory management and GPU kernel usage, setting CUDA apart from the sequential and OpenMP versions.

4 Experimental Setup

The experiments were conducted to measure the performance of different Mean Shift clustering implementations across varying dataset sizes and kernel types. Each implementation was evaluated under the following conditions:

4.1 Dataset and Parameters

- **Dataset Sizes**: The experiments were conducted on datasets with sizes of 10,000, 25,000, 50,000, and 100,000 points. This range allows for analysis of the scalability and efficiency of each implementation as data size increases.
- **Kernel Types and Parameters**:
  - **Flat Kernel**: Configured with a bandwidth $h = 20.0$ and a convergence threshold $\epsilon = 1.0$.
  - **Gaussian Kernel**: Configured with a bandwidth $h = 1.0$ and a convergence threshold $\epsilon = 0.2$.

4.2 Machine Specifications

The experiments were run on the following machine:

- **Operating System**: Microsoft Windows 11 Pro
- **CPU**: AMD Ryzen 7 5800X (8 cores, 16 logical processors)
- **GPU**: NVIDIA GeForce RTX 3080

4.3 Experimental Results and Analysis

The following tables present a summary of the execution times and speed-up factors across different dataset sizes for the Flat and Gaussian kernels. The performance of each implementation was measured using varying numbers of threads for the parallel versions. The first set of tables compares sequential and parallel execution times, while the second set compares the parallel OpenMP Critical and Atomic implementations with varying thread counts.

4.3.1 Flat Kernel: Sequential vs. Parallel

Size	Sequential (s)	Parallel (s)	Speedup
10,000	8.05	1.96	4.10x
25,000	51.69	12.37	4.18x
50,000	200.65	52.08	3.85x
100,000	710.79	64.61	11.00x

Table 1: Speedup for Flat Kernel with Different Dataset Sizes (Sequential vs. Parallel, No Threading and Multiple Threads)

The speedup for the Flat kernel increases with the dataset size, reaching a maximum of 11.00x for 100,000 points, especially with more threads. This indicates that parallelization becomes more effective as the dataset grows.

The parallel implementation consistently outperforms the sequential one for all dataset sizes, particularly as thread counts increase.

4.3.2 Gaussian Kernel: Sequential vs. Parallel

Size	Sequential (s)	Parallel (s)	Speedup
10,000	10.04	2.62	3.83x
25,000	82.17	22.61	3.63x
50,000	432.55	122.07	3.54x
100,000	2285.59	200.92	11.38x

Table 2: Speedup for Gaussian Kernel with Different Dataset Sizes (Sequential vs. Parallel, No Threading and Multiple Threads)

For the Gaussian kernel, the parallel implementation shows a similar trend, with speedups reaching up to 11.38x for 100,000 points. The performance benefits of parallelization are evident but slightly less pronounced compared to the Flat kernel.

4.3.3 Comparison of Critical and Atomic Synchronization

The following tables compare the performance of critical and atomic synchronization methods for both the Flat and Gaussian kernels across selected dataset sizes. Execution times are shown in seconds, with the faster time highlighted in bold for each dataset size.

For the Flat kernel, atomic synchronization performs better at 10,000 points, likely due to its lower overhead under low contention. However, as dataset size increases, critical synchronization becomes faster, as it manages contention more effectively by allowing exclusive access to shared resources, reducing conflicts among threads.

In the Gaussian kernel, which involves more complex computations, critical synchronization consistently outperforms atomic across all dataset sizes. This is because the critical method better manages the high contention levels associated with the Gaussian kernel, allowing for orderly thread access and minimizing the performance impact of concurrent updates.

Size	Critical (s)	Atomic (s)
Flat Kernel		
10,000	2.0281	**1.9639**
25,000	**12.3699**	12.4210
50,000	**52.0864**	52.2727
100,000	**64.6142**	75.3000
Gaussian Kernel		
10,000	**2.6151**	2.6198
25,000	**22.6137**	22.8253
50,000	**122.0716**	122.5091
100,000	**200.9229**	240.2295

Table 3: Comparison of Critical vs. Atomic Synchronization for Flat and Gaussian Kernels (execution times in seconds)

4.3.4 Thread Comparison Speedup

This experiment investigates the scalability of the Mean Shift algorithm by measuring performance across increasing thread counts on a dataset of 100,000 points. To coordinate concurrent access to shared resources, critical synchronization was applied, allowing threads to access resources in an orderly manner and reducing contention. The experiment tests thread counts of 2, 4, 6, 8, 10, 12, and 16 for both the Flat and Gaussian kernels.

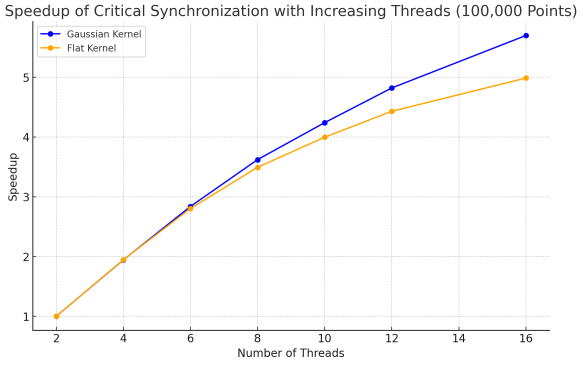

Figure 3: Speedup with Increasing Threads (100,000 Points) for Gaussian and Flat Kernels

As shown in Figure 3, for the **Gaussian kernel**, runtime starts at 1158.12 seconds with 2 threads and decreases to 203.20 seconds with 16 threads. This trend demonstrates

critical synchronization's effectiveness in handling the high contention and computational demands of the Gaussian kernel, as threads are coordinated to avoid conflicts.

For the **Flat kernel**, which has lower computational complexity, runtime begins at 323.57 seconds with 2 threads and reduces to 64.88 seconds with 16 threads. Although the speedup pattern is similar to that of the Gaussian kernel, it is slightly less pronounced due to the simpler calculations involved.

4.4 CUDA Performance

Before comparing the speed-up achieved by CUDA relative to other implementations, we first identify the optimal thread block size for the CUDA setup. The tables below show execution times (in seconds) for the 100,000-point dataset using different thread block sizes for both Flat and Gaussian kernels.

4.4.1 CUDA Performance for Flat Kernel

Threads per Block	Execution Time (s)
64	**16.868**
128	16.877
256	17.067
512	18.310
1024	23.019

Table 4: CUDA Execution Time for Flat Kernel (100,000 Points)

The best performance for the Flat kernel is achieved with 64 threads per block. As thread block size increases, execution time rises, indicating that smaller block sizes optimize the computational load and memory access for this kernel.

4.4.2 CUDA Performance for Gaussian Kernel

For the Gaussian kernel, the optimal configuration is achieved with 512 threads per block, with larger block sizes benefiting the more complex calculations. Beyond 512 threads, execution time increases, likely due to memory access overhead.

Threads per Block	Execution Time (s)
64	211.537
128	208.898
256	204.535
512	**202.969**
1024	271.365

Table 5: CUDA Execution Time for Gaussian Kernel (100,000 Points)

With these optimal configurations, we can now evaluate the speed-up of CUDA relative to other implementations in subsequent sections.

4.5 CUDA vs Sequential and Parallel Performance Comparison

For this comparison, we used the optimal configurations for each approach:

- **CUDA:** 64 threads per block for the Flat kernel and 512 threads per block for the Gaussian kernel, as these provided the best performance in prior testing.
- **Parallel:** The Critical configuration, which showed marginally better execution times than the Atomic setup.

The table below presents the execution times and the speed-up achieved by CUDA over Sequential and Parallel implementations for both the Flat and Gaussian kernels across different dataset sizes.

In the table, the CUDA implementation consistently outperforms the Sequential and Parallel configurations for both kernels, particularly with larger datasets. The Flat kernel benefits most from CUDA's acceleration, achieving substantial speed-up ratios over both CPU-based approaches. For the Gaussian kernel, CUDA provides speed-up but with narrower margins, especially on smaller datasets, due to its higher computational complexity and memory access demands.

5 Conclusion

This study provides a comprehensive comparison of Sequential, Parallel (OpenMP), and CUDA-based implemen-

Kernel	Size	Sequential (s)	Parallel (s)	CUDA (s)	Speed-Up Sequential	Speed-Up Parallel
Flat	10,000	8.051	2.028	**0.162**	49.7x	12.5x
	25,000	51.693	12.370	**0.817**	63.3x	15.1x
	50,000	200.651	52.086	**3.661**	54.8x	14.2x
	100,000	710.793	188.764	**16.868**	42.1x	11.2x
Gaussian	10,000	10.042	2.615	**2.929**	3.4x	0.9x
	25,000	82.174	22.825	**13.024**	6.3x	1.8x
	50,000	432.554	122.509	**50.406**	8.6x	2.4x
	100,000	2285.588	638.574	**202.969**	11.3x	3.1x

Table 6: CUDA Performance Comparison for Flat and Gaussian Kernels (Execution Times in Seconds and Speed-Ups)

tations of the Mean Shift clustering algorithm using Flat and Gaussian kernels. The experimental results, summarized in Tables 1, 2, 3, and 6, reveal several key insights:

- **Sequential vs. Parallel Implementations:** The parallel implementation using OpenMP consistently outperforms the sequential approach across all dataset sizes and both kernel types. Notably, the speedup achieved increases with larger datasets, highlighting the scalability benefits of parallelization. For instance, the Flat kernel achieved a speedup of up to 11.00x on a 100,000-point dataset (Table 1), while the Gaussian kernel reached up to 11.38x speedup on the same dataset size (Table 2).

- **Synchronization Methods:** The choice of synchronization method in the parallel implementation significantly impacts performance. Atomic operations generally provide better performance for lower dataset sizes and simpler kernels, as seen in the Flat kernel with 10,000 points (Table 3). However, for larger datasets and more complex kernels like Gaussian, critical sections offer superior performance by managing high contention more effectively.

- **CUDA Performance:** The CUDA implementation demonstrates substantial acceleration over both Sequential and Parallel (OpenMP) implementations, especially for larger datasets. For the Flat kernel, CUDA achieved speedups of up to 63.3x on a 25,000-point dataset and maintained impressive performance gains across all sizes (Table 6). Similarly, for the Gaussian kernel, CUDA provided meaningful speedups, reaching up to 11.3x on a 100,000-point dataset. The optimal thread block sizes identified (64 threads for Flat and 512 threads for Gaussian) were crucial in maximizing CUDA performance (Tables 4 and 5).

- **Scalability and Efficiency:** The Mean Shift algorithm scales effectively with both parallelization and GPU acceleration. As demonstrated in Figure 3, increasing the number of threads leads to significant reductions in runtime, especially for the Gaussian kernel, which benefits from CUDA's massive parallelism.

- **Kernel Complexity Impact:** The Flat kernel, being less computationally intensive, benefits more from CUDA acceleration, achieving higher speedup ratios. In contrast, the Gaussian kernel's increased complexity and memory access requirements result in slightly lower speedup margins, although CUDA still provides substantial performance improvements.

- **Thread Block Optimization:** Selecting the optimal thread block size is essential for achieving the best performance in CUDA implementations. Smaller thread blocks (64 threads) were optimal for the Flat kernel, while larger blocks (512 threads) were better suited for the more complex Gaussian kernel. Deviating from these optimal sizes resulted in increased execution times due to memory access overheads.

Overall, the integration of parallel computing techniques and GPU acceleration significantly enhances the performance of the Mean Shift clustering algorithm. The CUDA-based implementation, in particular, offers remarkable speedups, making it a viable solution for handling large-scale, high-dimensional datasets. The findings underscore

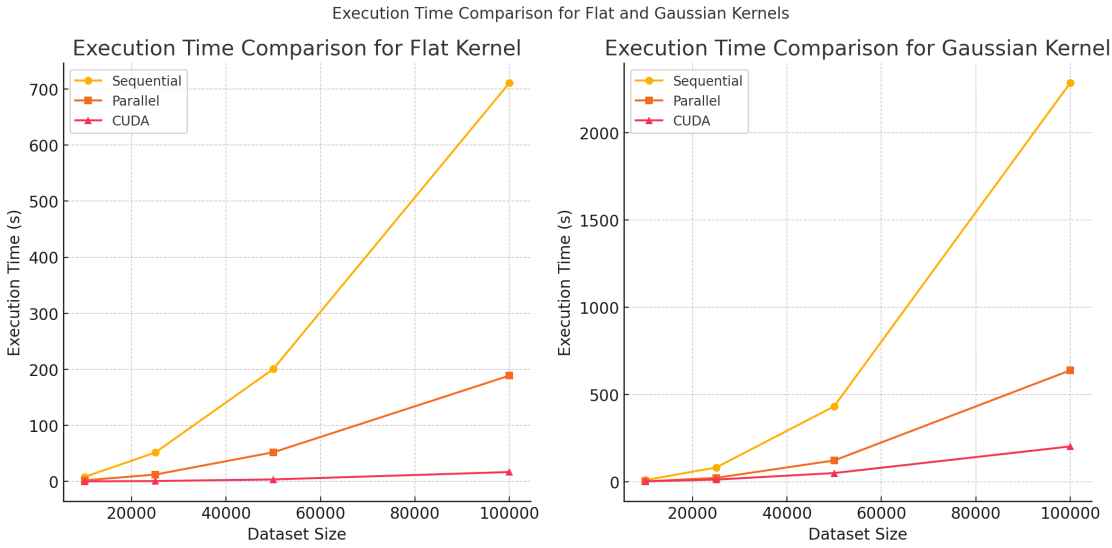

Figure 4: Execution Time Comparison for Flat and Gaussian Kernels across Sequential, Parallel, and CUDA Implementations with Varying Dataset Sizes.

the importance of choosing appropriate synchronization methods and optimizing thread block sizes to fully leverage the computational capabilities of modern multi-core CPUs and GPUs.

References

[1] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, 2002.

[2] K. Fukunaga, "Estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Transactions on Information Theory*, vol. 21, no. 1, pp. 32–40, 1975.

[3] L. Dagum and G. Dudek, "Openmp: An api for shared-memory parallel programming," in *Proceedings of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 1998, pp. 25–34.

[4] NVIDIA, *CUDA C Programming Guide*, NVIDIA, 2023, accessed: 2024-04-27. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html