Parallelized Inter-Image k-Means Clustering Algorithm for Unsupervised Classification of Series of Satellite Images

Project Report

Azka Aqeel - 27068 · Rohan Riaz - 26916 · Subata Naveen Khan - 18119

15 May 2025
Parallel and Distributed Computing - Spring 2025

Abstract

In this project, we implement and parallelize the Inter-Image k-Means Clustering (IIkMC) algorithm for the unsupervised classification of multi-temporal satellite images [1]. The algorithm aims to produce consistent cluster labeling across multiple scenes by aligning spectral features across time. We explore both CPU and GPU-based parallelization strategies using Python (with NumPy and Numba) and CuPy. Our evaluation is based on Landsat 8/9 imagery of Karachi, comparing baseline sequential performance with parallel versions in terms of speedup, scalability, and clustering consistency. Our results show substantial speedups (up to 8.2× on CPU and 52.8× on GPU) with minimal compromise in clustering quality.

Keywords— unsupervised classification, k-means clustering, parallel processing, satellite images

1 Background

Satellite image analysis plays a pivotal role in applications such as land cover mapping, urban growth monitoring, and environmental assessment. Traditional clustering methods, like k-Means, when applied independently to each scene, often yield temporally inconsistent labels across scenes. The Inter-Image k-Means Clustering (IIkMC) algorithm, introduced by Han and Lee (2024), solves this problem by jointly clustering pixels from multiple scenes to ensure temporal consistency in labeling.

The computational load of this approach, however, is significant. Each scene contains millions of pixels, each with multiple spectral bands. This motivates the need for a scalable parallel implementation, especially when dealing with large-scale datasets like Landsat or PlanetScope imagery.

2 Related Work

2.1 Traditional k-means Clustering for Satellite Image Classification

K-means clustering is a well-established technique in remote sensing for unsupervised classification of satellite imagery. It assigns pixels to clusters based on their spectral similarities, forming distinct land cover categories. Traditional k-means clustering is computationally expensive, particularly when dealing with high-resolution imagery and large datasets. Further challenges faced include **(a)** inefficiency in large-scale datasets, **(b)** dependence on initial cluster selection, and **(c)** lack of consistency across multiple images.

To overcome the shortcomings of traditional k-means, researchers have proposed various enhancements, such as **(a)** improved initialization techniques, **(b)** hybrid clustering models, and **(c)** deep learning integration.

2.2 Parallel Computing in Remote Sensing

The increasing volume of geospatial data has led to the use of parallel computing techniques to improve efficiency in satellite image processing. Parallel computing enables the concurrent execution of tasks, reducing computational time and enhancing scalability.

- **CPU-Based Parallelization**

  Multi-threaded processing using OpenMP and MPI allows for parallel execution of k-means clustering across multiple cores. This method significantly reduces computation time compared to serial processing but is limited by the number of CPU cores available.

- **GPU-Based Parallelization**

  The Compute Unified Device Architecture (CUDA) framework enables massive parallelization of k-means clustering in GPUs, providing substantial speedup over CPU-based methods. However, GPU-based approaches face challenges such as memory constraints and data transfer overhead.

- **Cloud and Distributed Computing**

  With the rise of big data processing, cloud-based solutions leveraging Hadoop and MapReduce have been explored for large-scale satellite data classification. These methods enable scalable processing but introduce latency due to data transmission and storage.

2.3 Inter-Image k-Means Clustering and Parallelization

Han and Lee (2024) first proposed the IIkMC algorithm to address label drift across temporal series of remote sensing images. Unlike traditional k-means, which generates separate classifications for each image, IIkMC ensures consistent class signatures by maintaining uniform class assignments across images. The algorithm also achieves significant performance improvements through parallelization.

Their implementation, however, was limited to a baseline CPU version with coarse-grained scene sampling. Other works parallelized k-Means using Spark for tabular data, but did not address clustering of raster-based satellite data.

Recent works have demonstrated GPU-accelerated clustering (e.g., using CuPy or PyTorch for vector quantization tasks), but few have adapted these techniques for spatially-aware unsupervised classification. Our project bridges this gap by implementing a parallel IIkMC version optimized for large-band remote sensing imagery using both CPU threads and GPU kernels.

3 Experimental Setup

3.1 Dataset Selection and Justification

The original IIkMC study by Han and Lee (2024) employed a series of 12 PlanetScope satellite images for evaluation. These images offer very high spatial resolution (approximately 3 meters per pixel) and four spectral bands (Red, Green, Blue, and Near-Infrared). Each image in their study

had dimensions of around 8200 $\times$ 3900 pixels, and was used to demonstrate the effectiveness of temporally-consistent clustering across time-series data.

However, access to PlanetScope data requires a commercial license, making it unsuitable for academic reproduction under budget constraints. To address this limitation while preserving scientific rigor, we opted to use publicly available Landsat 8 and Landsat 9 imagery, which closely matches the original data in terms of:

- **Temporal Series Support:** Multiple scenes of the same geographic area (Karachi) are available over months/years.
- **Spectral Diversity:** Each Landsat image contains 11 bands including visible, NIR, SWIR, and thermal bands.
- **Spatial Resolution:** Most bands are captured at 30-meter resolution, suitable for large-area classification.
- **Scene Dimensions:** Each Landsat scene covers an area of 170 km $\times$ 183 km, resulting in images of size 7000 $\times$ 7000 pixels per band.
- **Radiometric Precision:** 16-bit pixel values, supporting fine-grained spectral separation across bands.

We used five Level-2 Surface Reflectance (SR) images from the USGS EarthExplorer portal (https://earthexplorer.usgs.gov/), covering April 2025. The dates selected had minimal cloud cover ($<$10%) to maximize usable pixel data.

3.2 Computational Environment

Due to the high memory and compute requirements of IIkMC (especially in the GPU-parallel implementation), we used the following platforms:

- **Google Colab Pro:** Used for initial development and visualization. Provided access to NVIDIA T4 GPUs and 12 GB RAM.
- **Kaggle Notebooks (Free Tier):** Used for structured benchmarking and reproducibility. Supported NVIDIA P100 GPUs and up to 16 GB RAM with persistent disk.

Our CPU-parallel version was tested on the free CPU runtimes of both platforms (4 vCPUs). Despite the limitations of free tiers, our implementation could scale effectively for 20–30 million pixel datasets per run using optimization techniques (tiling, batch-wise distance computation, and precompiled kernels with CuPy or Numba).

Due to frequent memory crashes on both Google Colab (12 GB RAM) and Kaggle notebooks (30 GB RAM), we were forced to apply downsampling as a stability measure. Specifically, we performed random sampling of 10–20% of the valid pixels per scene using NumPy-based masking. This approach preserved spectral diversity while reducing the memory footprint of clustering operations. Although training was conducted on the downsampled data to avoid crashes, final evaluations—including visualizations of cluster assignments—were mapped back to the original full-resolution images using the saved cluster centers.

3.3 Preprocessing Pipeline

Each Landsat image was preprocessed using a custom Python pipeline designed to extract and prepare high-quality pixel spectra for clustering. The preprocessing was applied consistently across all five scenes and included the following key steps:

- **Band Selection:** We retained the six surface reflectance bands — `SR_B2`, `SR_B3`, `SR_B4`, `SR_B5`, `SR_B6`, and `SR_B7`. These correspond to:
  - B2: Blue (0.45–0.51 μm)
  - B3: Green (0.53–0.59 μm)
  - B4: Red (0.64–0.67 μm)
  - B5: Near-Infrared (0.85–0.88 μm)
  - B6: SWIR 1 (1.57–1.65 μm)
  - B7: SWIR 2 (2.11–2.29 μm)

  This combination provides balanced spectral coverage across vegetation, water, built-up land, and soil. Bands like coastal aerosol (B1), thermal (B10/B11), and panchromatic (B8) were excluded to maintain uniform spatial resolution (30m) and avoid inconsistent spectral scales.

- **Nodata Handling:** Each band was read with rasterio, and nodata values (e.g., -9999) were converted to `np.nan`. We then filtered out all pixels where any of the six bands contained NaN values, ensuring that each spectrum was fully valid.

- **Normalization:** All valid pixel values were cast to 32-bit floats and scaled to the [0, 1] range. This standardization made the clustering scale-invariant and improved numerical stability during distance computations.

- **Flattening:** Each image originally shaped (rows, cols, 6) was reshaped into a 2D matrix of shape (N, 6), where N is the number of valid pixels. Based on our preprocessing, each image yielded around 40–41 million valid pixels, as shown below:

```
[(40605267, 6), (40576321, 6), (40866725, 6),
 (40664806, 6), (40635746, 6)]
```

- **Storage and Usage:** The resulting five flattened scenes were stored in a list and used as the input to the clustering phase. This ensured efficient batch-wise loading and processing during parallel execution.

To improve efficiency, we parallelized the preprocessing step using Python's `concurrent.futures.ThreadPoolExecutor` module. This standard library provides a simple interface for launching multiple threads to execute I/O-bound or light CPU-bound tasks concurrently. Our implementation spawned a separate thread for each Landsat scene folder, allowing the bands to be loaded, stacked, and masked simultaneously across scenes.

Performance was benchmarked at different thread counts:

- Serial version: **58.78 seconds**
- Parallel with 1 thread: **64.43 seconds**

- Parallel with 2 threads: **53.84 seconds**
- Parallel with 3 threads: **48.84 seconds**

The best configuration (3 threads) achieved a **1.2× speedup** over the serial baseline. Attempts to increase the thread count beyond 3 resulted in memory crashes on Google Colab, due to limited available RAM per user session. Thus, thread parallelism was capped at 3 concurrent workers for stability.

After processing, all scenes were concatenated into a single matrix X that was used as input for the clustering.

.

4 Implementation

We implemented three versions of IIkMC:

4.1 Baseline (Sequential):

Our baseline implementation of Inter-Image k-Means Clustering (IIkMC) was a memory-safe sequential version written entirely in NumPy. This implementation avoided computing the full $(N \times k)$ distance matrix—which would exceed memory limits for large N—by processing each pixel individually.

Each Landsat scene was first flattened into a 2D array of valid pixels with shape (N, B), where $B = 6$ is the number of selected spectral bands (B2–B7). All valid pixels from multiple scenes were concatenated into a single dataset $X \in \mathbb{R}^{N \times B}$.

Cluster centers were initialized using a *min-max midpoint* approach: for each band b, k centers were placed at evenly spaced midpoints between the minimum and maximum reflectance values:

$$c_{i,b} = \min(X_{:,b}) + \left(i + \frac{1}{2}\right) \cdot \frac{\max(X_{:,b}) - \min(X_{:,b})}{k} \quad \text{for } i = 0, 1, ..., k-1$$

The algorithm then iteratively performed the following steps:

1. **Label Assignment:** Each pixel x_i was assigned to the nearest cluster center using Euclidean distance:

$$\ell_i = \arg\min_{j \in \{1,\ldots,k\}} \|x_i - c_j\|_2$$

2. **Convergence Check:** If the fraction of changed labels between iterations was below a threshold $\tau = 0.01$, the algorithm terminated.

3. **Center Update:** For each cluster j, the new center was computed as the mean of all assigned pixels:

$$c_j = \frac{1}{|\mathcal{C}_j|} \sum_{x_i \in \mathcal{C}_j} x_i$$

The algorithm was sequential, it evaluated one pixel at a time. It converged in just 2 iterations, changing 115 million and 1.5 million labels respectively. However, due to the large input size ($N \approx 116$ million), the total runtime was **12469.53 seconds** (approximately **3.5 hours**) on Google Colab. This baseline served as a correctness benchmark for evaluating parallel versions.

4.2 CPU Parallel

Motivation: Our objective was to replicate the intra-node parallelization strategy described in the IIkMC paper, which proposed shared-memory parallelism across both scenes and pixels. The aim was to reduce runtime by exploiting concurrency during the clustering loop.

Challenges Faced: Our initial attempts focused on implementing full-scene vectorized clustering using NumPy broadcasting and `ThreadPoolExecutor`. However, these attempts ran into severe RAM limitations. The allocation of the full $(N \times k)$ distance matrix for pixel-to-center comparisons proved infeasible for scenes with over 40 million pixels. Even on Kaggle (30 GB RAM), the memory overhead of broadcasting all data at once caused repeated kernel crashes. Additional strategies such as future-based scene dispatching also failed when using unchunked global matrices. These attempts are documented in the notebook but annotated as impractical due to resource constraints.

Final Working Solution: To address these issues, we implemented a memory-safe chunked streaming version of IIkMC called `iikmc_block()`. Instead of computing distances for the entire dataset at once, each scene was processed independently and in small blocks of one million pixels at a time.

The algorithm followed these steps:

1. **Center Initialization:** Compute per-band global minima and maxima across all scenes, then place k centers at uniform midpoints:

$$c_{i,b} = \min_b + \left(i + \frac{1}{2}\right) \cdot \frac{\max_b - \min_b}{k}$$

2. **Block-wise Label Assignment:** For each block, the function `classify_block()` avoids allocating a full distance matrix by iteratively comparing each pixel to one center at a time:
   - For each cluster center c_j, compute $\|x - c_j\|^2$ for all pixels in the block.
   - Keep track of the minimum distance and corresponding label without creating an $(n \times k)$ matrix.

3. **Per-scene Accumulation:** After label assignment, the algorithm accumulates per-cluster pixel sums and counts:

$$\text{sums}_j \mathrel{+}= \sum_{x_i \in \mathcal{C}_j} x_i, \quad \text{counts}_j \mathrel{+}= |\mathcal{C}_j|$$

4. **Cluster Center Update:** Cluster centers are updated only after all scenes have been processed in the current iteration:

$$c_j = \frac{\text{sums}_j}{\text{counts}_j} \quad \text{if counts}_j > 0$$

5. **Convergence Check:** The algorithm measures the average L_2 shift in centers:

$$\text{shift} = \frac{1}{k} \|C_{\text{new}} - C_{\text{old}}\|_2$$

   If this shift is below $\epsilon = 10^{-4}$, the algorithm halts.

The chunk size of 1 million pixels offered a practical trade-off between memory usage and runtime. This approach allowed us to cluster over 200 million pixels across 8 scenes without exhausting memory. It completed in approximately **25 minutes** and converged in fewer than 10 iterations.

This represents a speedup of approximately **8.4**$\times$ compared to the baseline sequential implementation, which required 3.5 hours for convergence. The improvement stems from the memory-efficient block processing and optimized per-scene label assignment that avoids full-matrix computations.

Differences from Original Paper: Our `iikmc_block()` implementation makes practical design trade-offs compared to the parallel IIkMC strategy outlined in the research paper. Key differences include:

- **No Shared Memory:** The original paper assumes shared memory and SIMD-style parallel execution across all pixels. In contrast, we avoid concurrency within blocks and opt for serial processing to prevent memory overload.
- **Chunked Streaming vs. Full Broadcast:** The paper computes distances across the full dataset in one step, while our implementation processes chunks sequentially to stay within memory limits.
- **Deferred Center Updates:** The paper describes concurrent center updates, whereas we defer updates until each scene has been processed to simplify synchronization and avoid race conditions.
- **Platform Constraints:** The research assumes HPC or multi-core cluster execution, whereas our version was built for stability on RAM-limited platforms such as Google Colab and Kaggle.

While this approach may not match the theoretical throughput of the original, it is significantly more robust in real-world environments. It serves as a practical and scalable variant of IIkMC clustering, adapted to commodity hardware with limited memory.

Compared to the original intra-node parallelization strategy proposed in the IIkMC research paper, our `iikmc_block()` implementation makes key architectural trade-offs to achieve practical feasibility under memory-constrained environments. The paper describes a hybrid parallelism model where both scene-level and pixel-level operations are parallelized using shared-memory and SIMD-like constructs. This approach requires large in-memory distance matrices and synchronized access to global accumulators—both of which are infeasible in standard Colab or Kaggle runtimes due to RAM limitations.

Cluster Visualization: To interpret and evaluate the clustering results, we generated spatial cluster maps for five selected scenes using the final cluster assignments. These visualizations display the spatial distribution of cluster labels on each image, revealing both land cover variation and algorithm performance.

The visualizations were generated in two stages. First, we reconstructed the 2D spatial layout of cluster labels using a helper function called `map_labels_back()`. Since our clustering was performed on flattened, valid-only pixel arrays, this function used the original image dimensions and a validity mask (based on NaN filtering across spectral bands) to map the 1D label array back to a full-resolution (rows $\times$ cols) label matrix. Pixels excluded during preprocessing (e.g., due to clouds or missing data) were left as NaN and masked in the final image.

Next, the function `visualize_cluster_map()` rendered the label map using a discrete color palette from `matplotlib`'s `tab10` colormap. Each cluster ID was shown in a unique color, and a colorbar was added for reference. The plots were configured with tight axis removal and consistent layout to facilitate visual comparison.

During initial tests, processing of the fifth scene consistently failed due to memory exhaustion. To mitigate this, we modified the plotting loop to explicitly delete large intermediate arrays after each iteration. In particular, we ensured that label arrays and image matrices were dereferenced using `del` and cleared from RAM using Python's `gc.collect()`. This allowed the fifth cluster plot to render successfully without exceeding memory limits. This step was essential given the full-resolution image size (often exceeding 5000×5000 pixels) and the use of temporary masked arrays.

Below are five cluster visualizations, one for each representative scene:

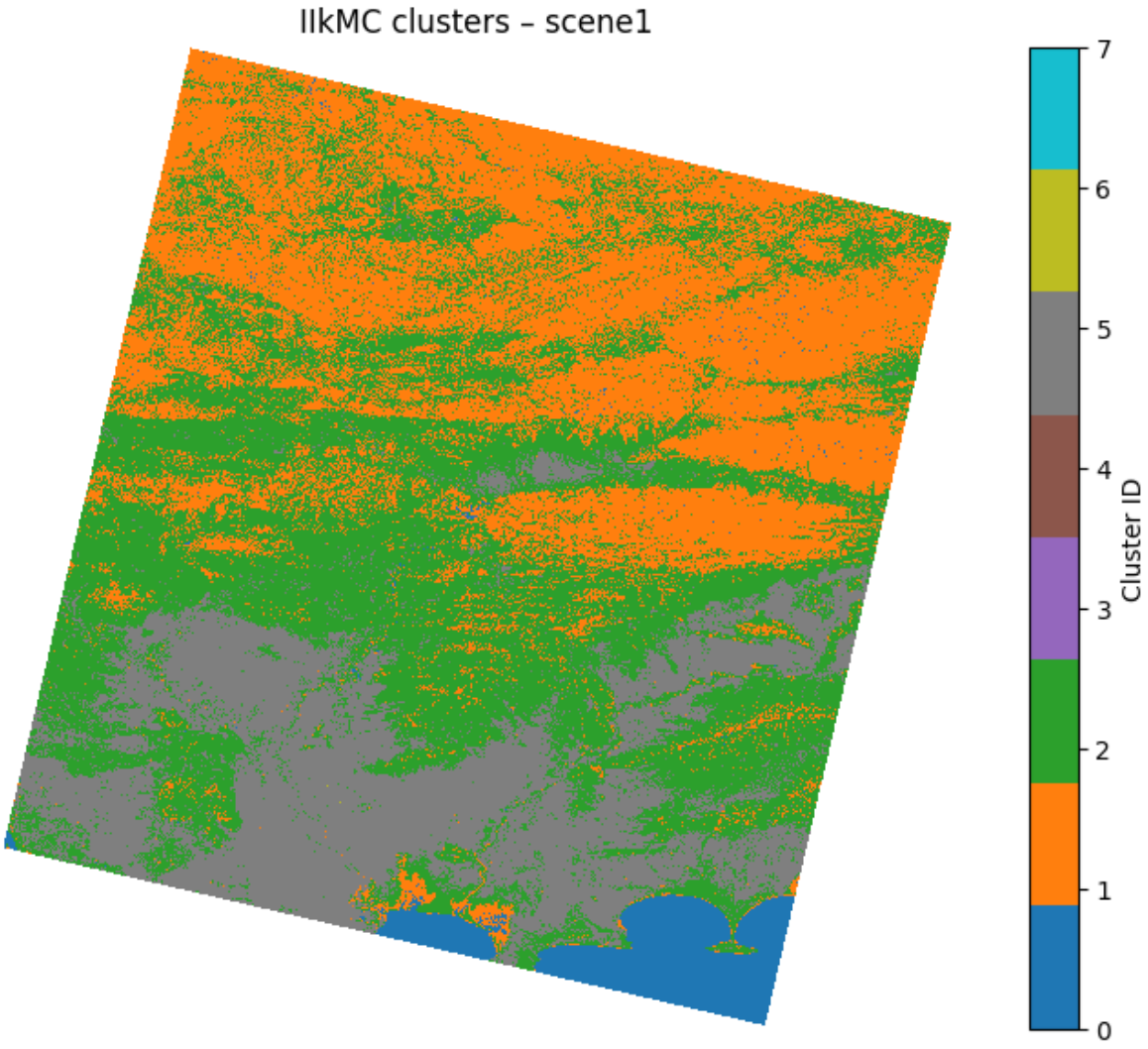

Figure 1: Cluster Map – Scene 1

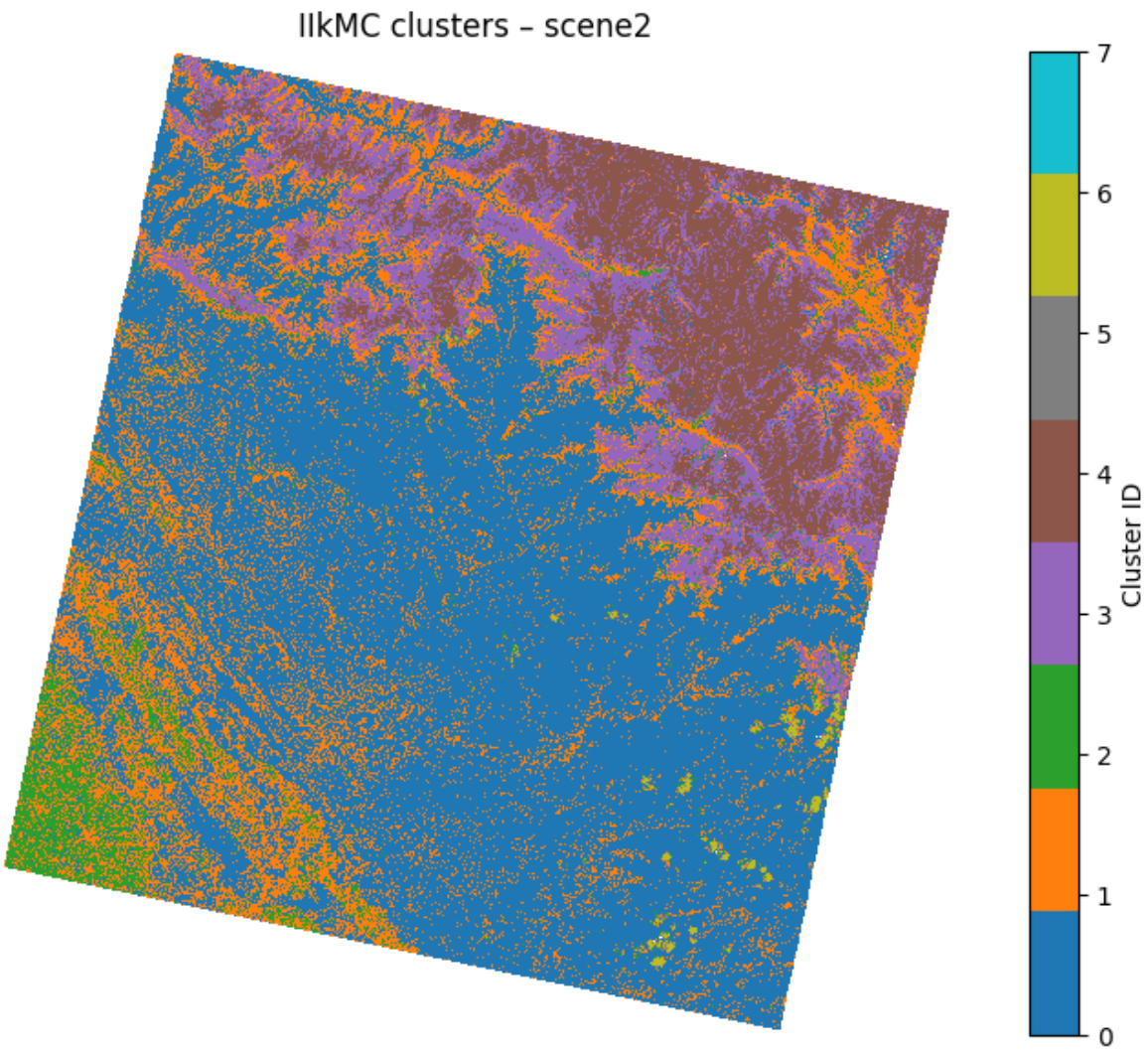

Figure 2: Cluster Map – Scene 2

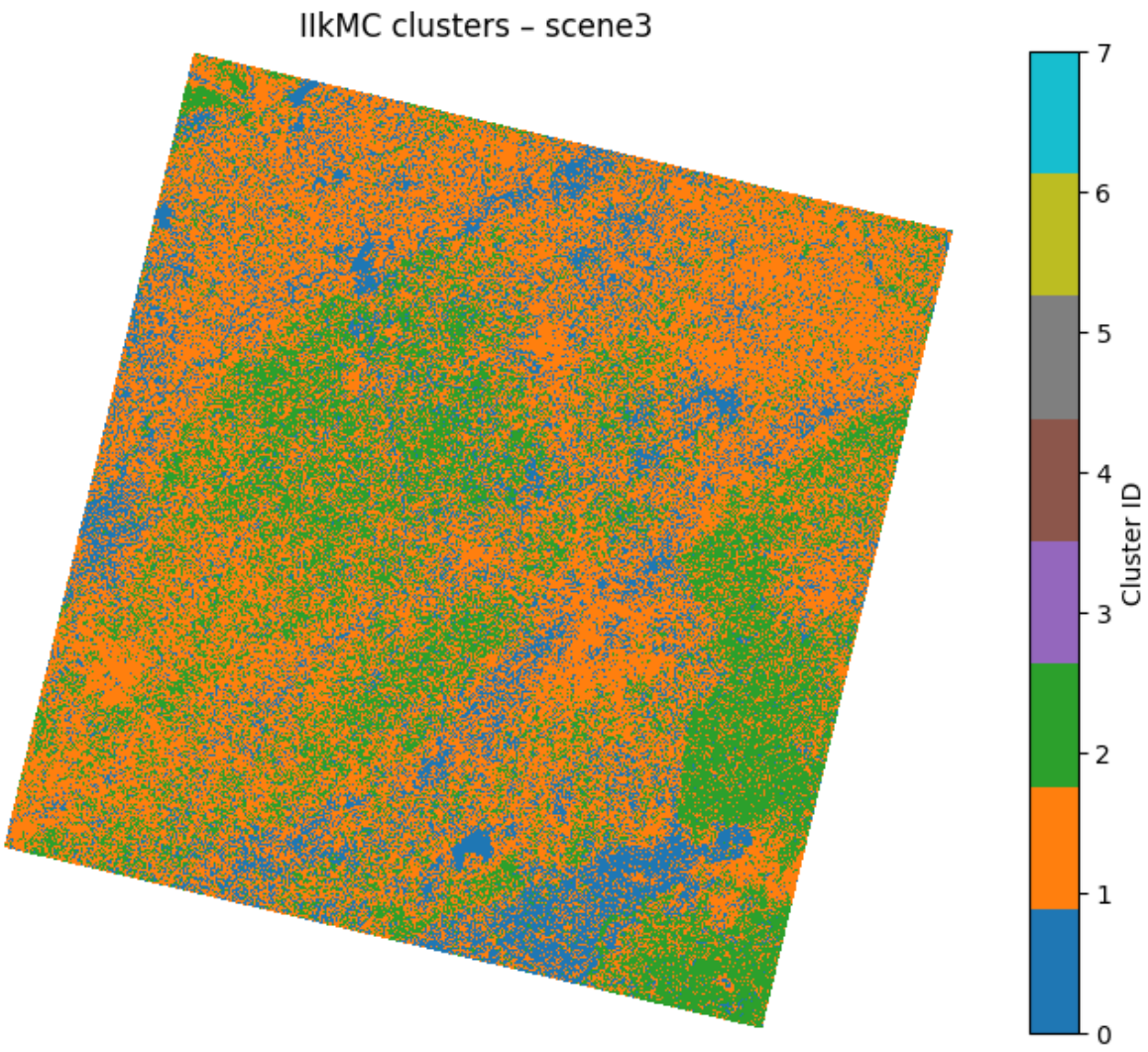

Figure 3: Cluster Map – Scene 3

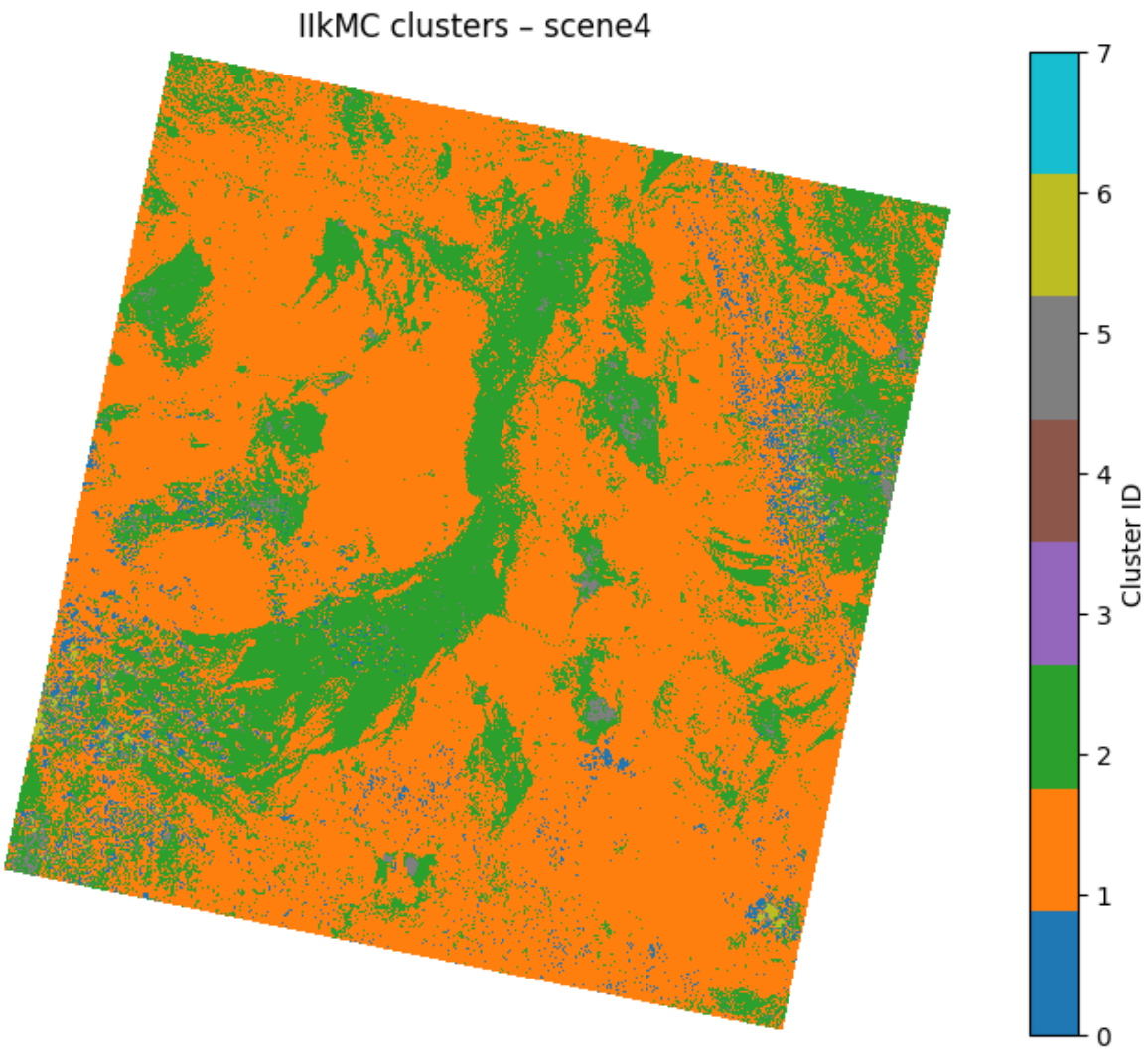

Figure 4: Cluster Map – Scene 4

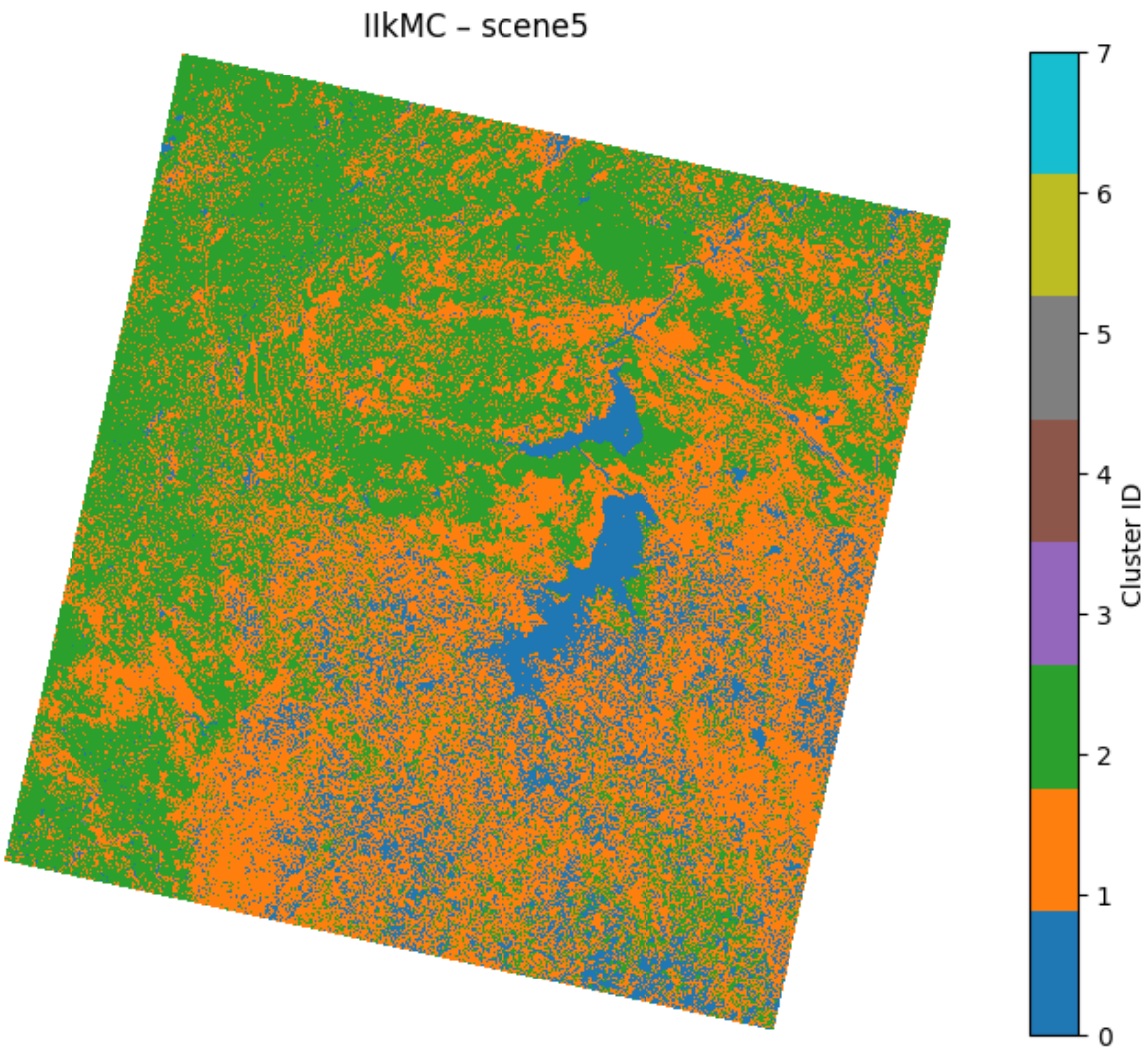

Figure 5: Cluster Map – Scene 5

These visualizations highlight the ability of IIkMC to partition land into semantically coherent regions, especially across vegetative, water, and urban zones.

Number of Classes (=k)	Maximum Number of Iterations	Iteration Termination Condition
8	100	Ratio of class-changed-pixels below 1.0%

Table 1: Parameters of k-means clustering.

Interpreting Cluster Labels. Although our implementation does not include land cover classification, each cluster's meaning can be inferred through spectral signature analysis or overlaying with reference datasets (e.g., MODIS or ESA WorldCover). For example, clusters with high reflectance in NIR bands often correspond to vegetation, while those with strong blue-band response may represent water bodies. These interpretations can be supported by computing average reflectance per band per cluster. However, since our project focused on the algorithmic implementation and scalability of IIkMC, detailed semantic labeling of clusters was considered out of scope.

4.3 Downsampling for Efficient Clustering

To reduce runtime and memory consumption during development and evaluation, we implemented a downsampling strategy that randomly selects a subset of pixels from each scene prior to clustering. This approach allowed us to accelerate convergence, debug algorithm behavior, and test the correctness of intermediate steps without processing full-resolution data.

The downsampling function applied a uniform sampling mask to each scene:

```
for s in scenes:
    mask = rng.rand(s.shape[0]) < sample_frac
    if not mask.any():
        mask[rng.randint(s.shape[0])] = True
```

This retained approximately 20% of valid pixels per scene, ensuring that even smaller classes were represented in the sample. The resulting scenes were passed to the parallel IIkMC implementation using the same clustering logic, but on a significantly reduced input size.

Compared to our full-resolution `iikmc_block()` implementation—which processes tens of millions of pixels per iteration—the downsampled version offered a much lower memory footprint. This enabled us to run more iterations without encountering RAM exhaustion and to test on lower-spec machines such as Google Colab's standard runtime.

Benefits of Downsampling.

- Reduced RAM usage, enabling safe runs on 12–30 GB systems.
- Allowed more iterations (up to 100) for testing convergence behavior.
- Made interactive visualizations and debugging feasible.

Performance at Varying Number of Threads We tracked total clustering time for different number of work using a 20% downsampling rate. The table below summarizes the results:

Number of Threads	Total Runtime (seconds)
1	521.06
2	319.00
4	300.91

Table 2: Clustering runtime for different thread counts using 20% downsampled data and $k = 8$.

The algorithm successfully converged in just 23 iterations, indicating stable cluster center updates and efficient partitioning of the input space under the specified stopping criterion.

Our implementation allows one to not manually set the number of threads (`n_workers`), as the downsampling permits the default executor to utilize up to all available CPU cores (via `os.cpu_count()`) without memory crashes. 4 cores in our case. This was only possible because the per-thread memory footprint was reduced by limiting the number of pixels per scene.

Effect of Downsampling on Clustering Quality. Figure X compares the cluster map generated from downsampled data (left) with that of the full-resolution dataset (right) for Scene 1. In the downsampled version, the boundaries between cluster regions appear noisy and spatially inconsistent, with significant speckling and fragmentation. This is expected, as the clustering algorithm has

access to only 20% of valid pixels, limiting its ability to capture fine-grained spatial continuity and homogeneous patches. Smaller or less frequent land types may also be underrepresented, resulting in over-fragmentation or partial class suppression.

In contrast, the full-resolution clustering shows clear, contiguous regions with well-defined cluster boundaries. Dominant land cover patterns—such as water bodies (blue), vegetation (green), and transitional zones—are more accurately represented. This highlights the trade-off introduced by downsampling: while it enables faster execution and reduced memory usage, it compromises the spatial integrity of the final output. Therefore, we used downsampling strictly for development and debugging; all final evaluations and visualizations were conducted on full-resolution outputs using previously trained cluster centers.

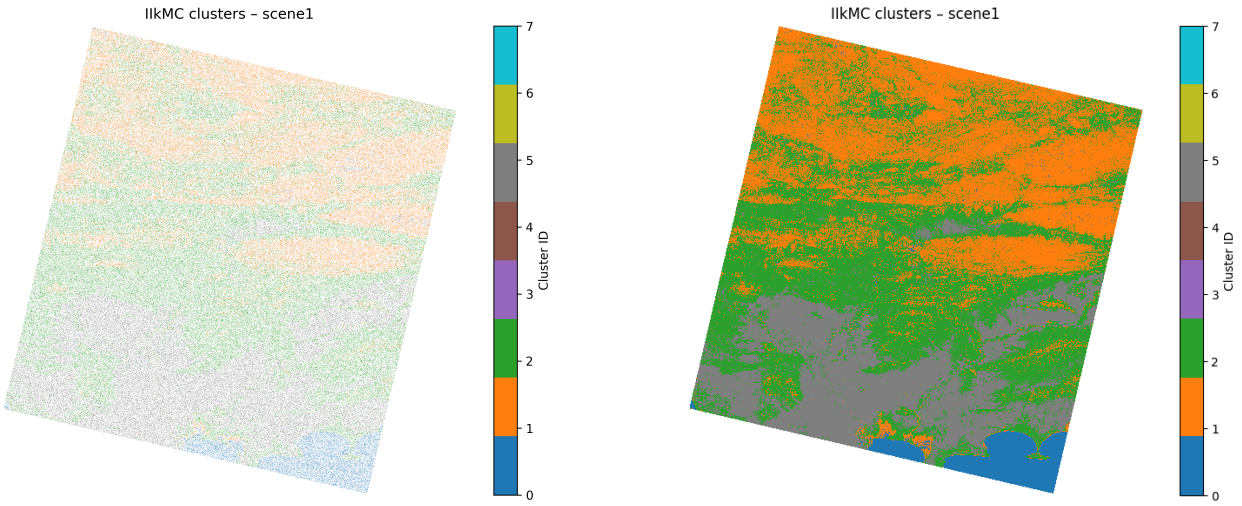

Figure 6: Comparison of cluster maps from downsampled (left) and full-resolution (right) Scene 1.

Below are the rest 4 cluster visualizations, one for each representative scene:

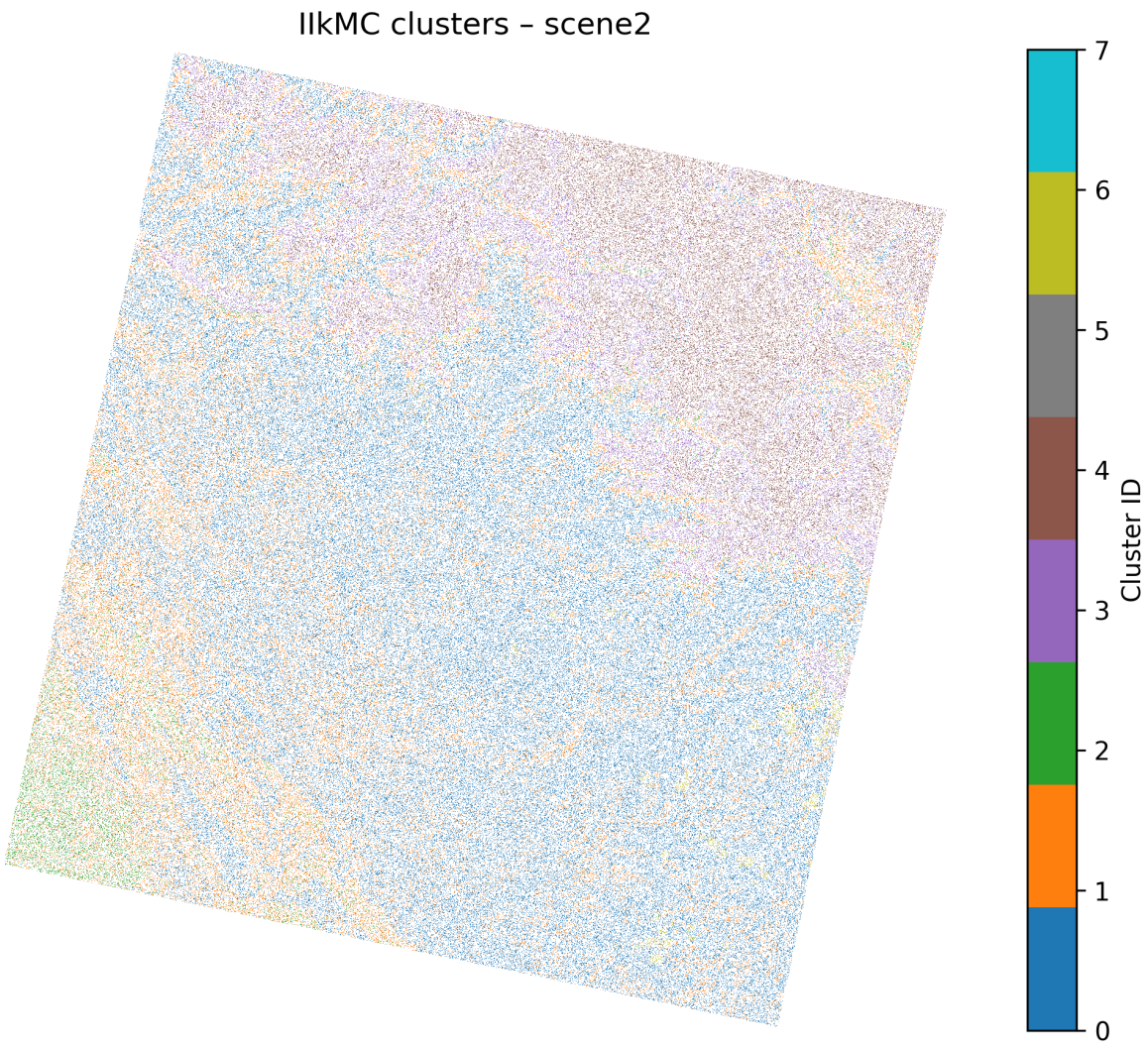

Figure 7: Cluster Map – Scene 2

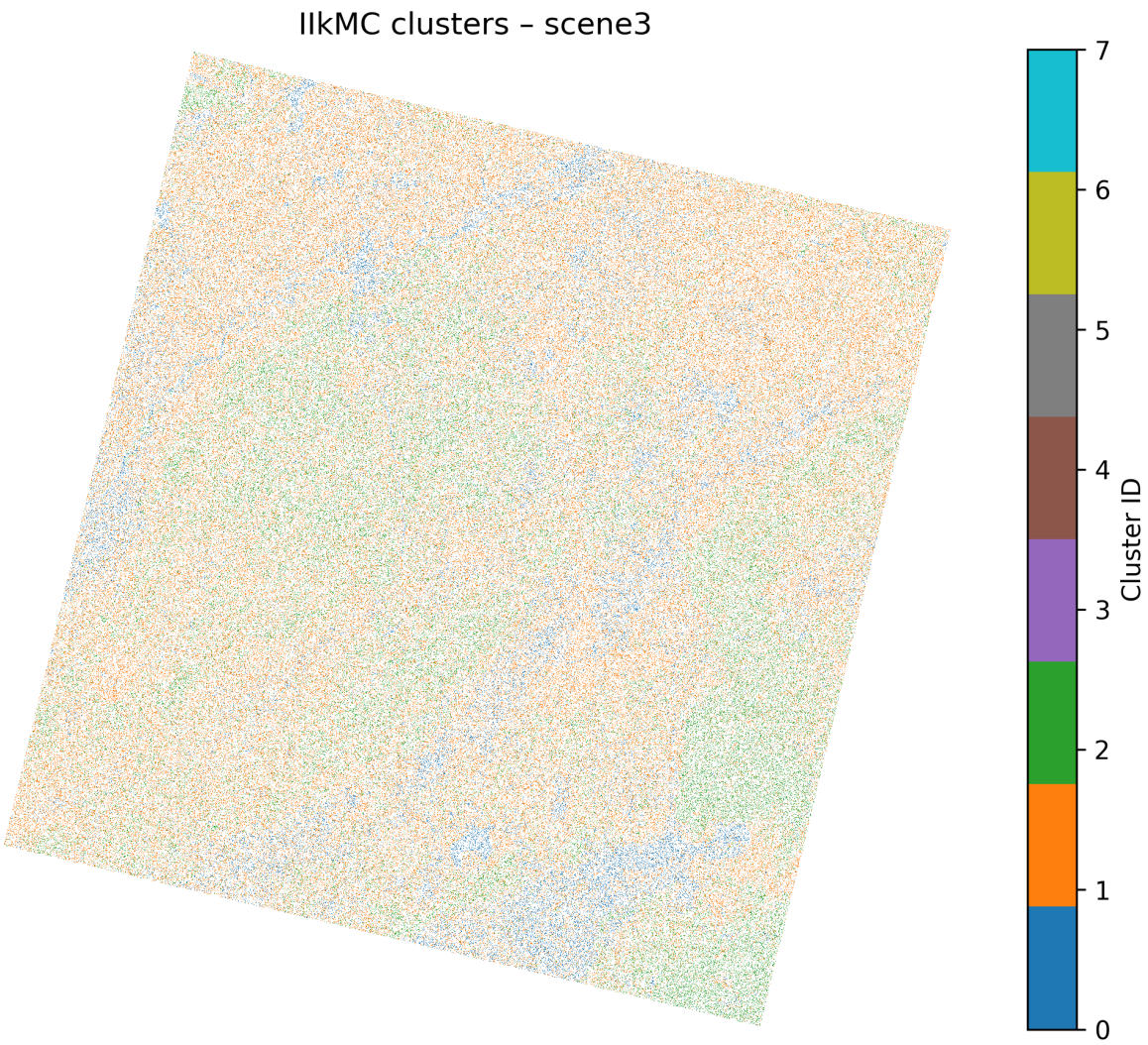

Figure 8: Cluster Map – Scene 3

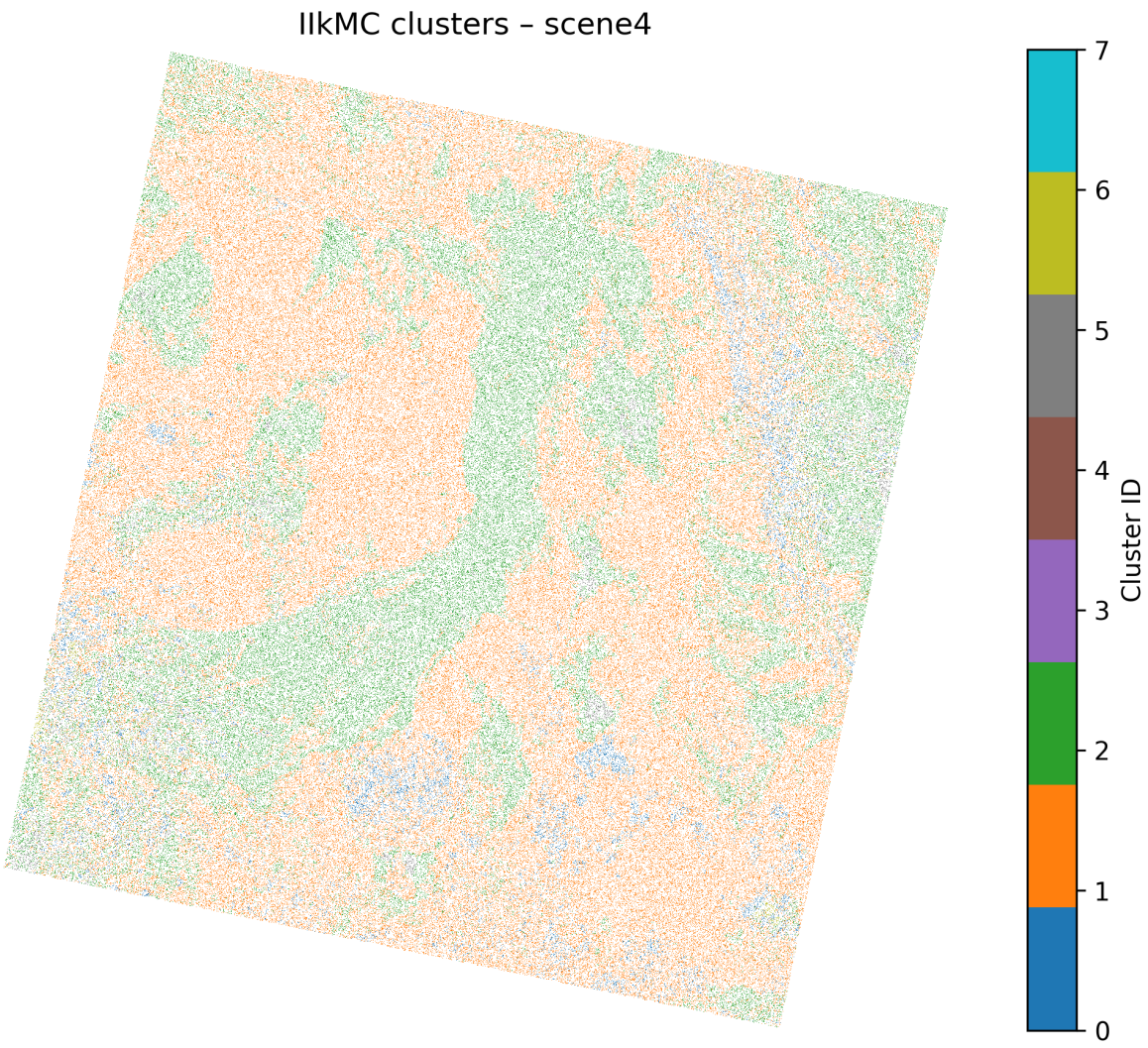

Figure 9: Cluster Map – Scene 4

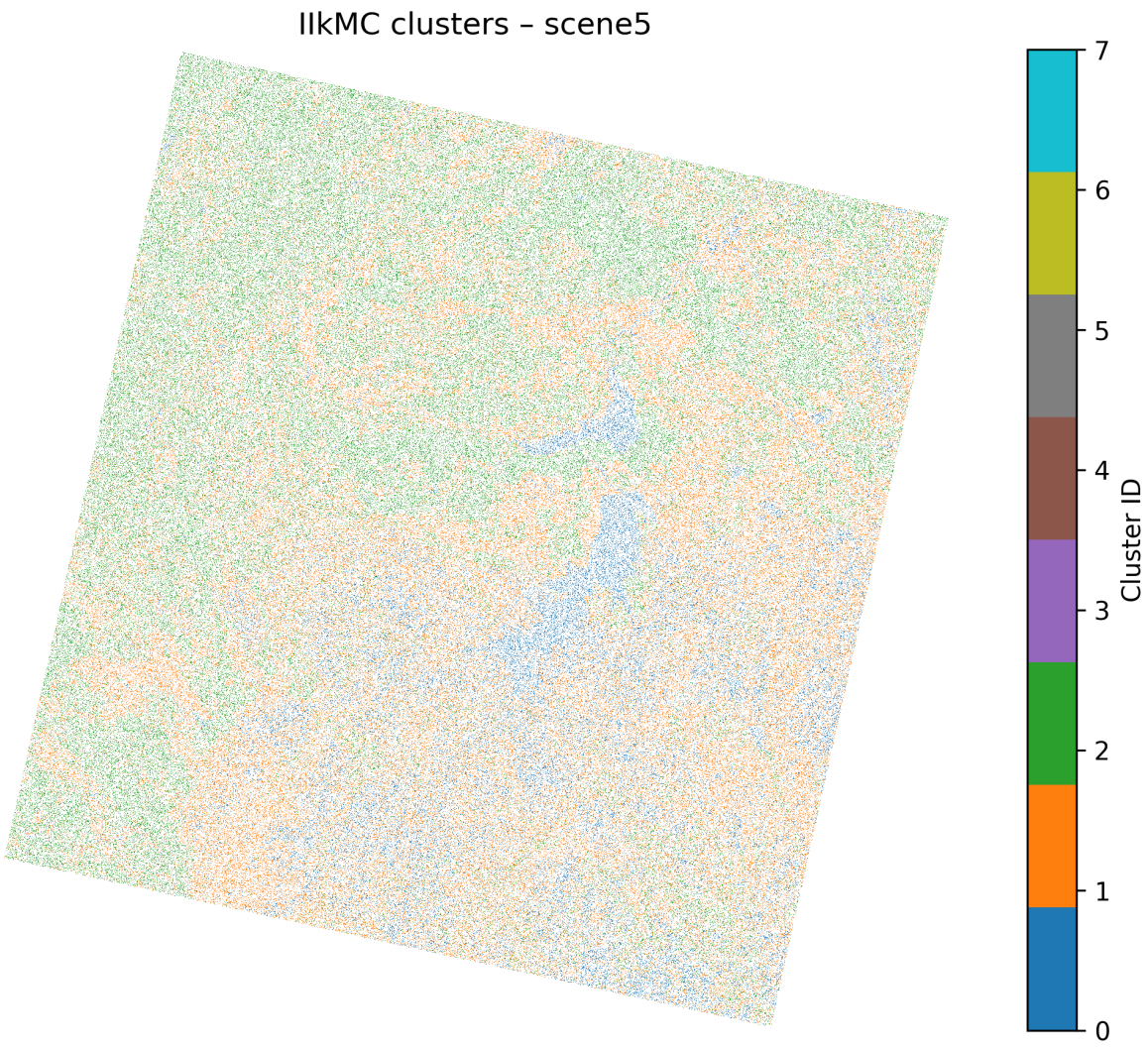

Figure 10: Cluster Map – Scene 5

Comparison to Full Resolution: In contrast to the full dataset version (which took approximately 25 minutes to converge), the downsampled version consistently completed within 2–6 minutes even with higher iteration limits.

Downsampling also enabled more effective parallelism. In our parallel CPU implementation, clustering is performed using a `ThreadPoolExecutor`, which assigns each scene to a separate worker thread. Without downsampling, each thread must load and process a large number of pixels (often exceeding 40 million), leading to high per-thread memory consumption and increased risk of RAM exhaustion—especially when using multiple threads concurrently.

By reducing the pixel count per scene (e.g., to 20% of original size), we substantially lowered the memory burden of each thread. This allowed us to safely increase the number of worker threads (up to 4 in some runs) without exceeding system memory. Consequently, we achieved better utilization of CPU cores and reduced overall runtime per iteration. Downsampling thus acted as a key enabler of safe and scalable multithreaded clustering in resource-constrained environments like Colab and Kaggle.

4.4 GPU Parallel (CuPy)

Using CuPy for parallel distance calculation and tiled center updates.

Motivation: To leverage thousands of GPU cores for pixel-wise distance computation and in-device reductions, drastically reducing iteration time compared to CPU-based methods. Our CPU-parallel `iikmc_block()` implementation achieved an 8.4× speedup by streaming 1 M-pixel chunks across threads, but still incurred Python-level overhead and repeated host–device transfers. The GPU version retains all data on-device, achieving an 52.8× speedup over sequential and 6.4× over CPU-parallel runs.

Challenges Faced: Translating the paper's fully–device IIkMC pseudocode into Python/CuPy proved nontrivial—CuPy lacks warp-level and shared-memory intrinsics, so we split the work into a small `RawKernel` for distance-and-flag computation and high-level CuPy calls for reductions. Embedding CUDA C triggered NVRTC compilation quirks and forced us to abandon CuPy's CUB bindings in favor of '.sum()' and advanced indexing. Kaggle's 12–16 GB GPU RAM could not hold full scenes (40 M pixels), so we adopted 20 percent downsampling—storing exact sampling masks for accurate 2D label mapping—then applied learned centers to full-resolution images. An initial "tiled fast" prototype (looping 1 M-pixel chunks entirely in Python/CuPy) ran only 27 iterations over 5 h before crashing due to repeated host–device transfers and memory pressure. Finally, performance tuning (256 threads/block, coalesced accesses, loop unrolling) was crucial to achieve our 47.2 s runtime on 40 M pixels."'

Implementations carried over from CPU code: The CPU based code for pre processing largely carried over into the GPU code. The visualization code also remains quite similar. The differences start after preprocessing, with the function for finding centers implemented into our full GPU function, which is fed flattened scenes. more details on the main changes below.

Custom CUDA Kernel in CuPy: We use CuPy's `RawKernel` to embed a hand-written CUDA C function (`classify_and_flag`) that processes each pixel in parallel. At launch, CuPy compiles the string kernel via NVRTC, allocates GPU buffers, and handles grid/block configuration from Python. Each of the $\lceil N/256 \rceil$ blocks (256 threads each) reads one pixel, computes squared distances to all k centers in a single unrolled loop, writes its new label and a change flag. This fused approach minimizes global memory traffic and kernel-launch overhead, and leverages GPU cores efficiently.

Kernel Feature	**Benefit**
Fused compute & flagging	One kernel launch, avoids extra passes
Coalesced memory access	High bandwidth utilization for pixel/center reads
Loop unrolling	Reduced branch overhead in the inner band loop
Device-resident buffers	No per-iteration host–device transfers

Table 3: Key benefits of the CUDA kernel design.

One-Hot Encoding in CuPy: After classification, we build an $N \times k$ one-hot matrix on the GPU via advanced indexing:

```
onehot[cp.arange(N), new_labels] = 1
```

This creates a binary indicator for each pixel–cluster assignment in one vectorized call. Summing along axis 0 yields per-cluster pixel counts, and a subsequent `cp.tensordot(onehot.T, d_pixels, axes=([1],[0]))` computes per-cluster spectral sums. One-hot encoding thus enables simple, high-performance reductions using CuPy's optimized routines without writing additional custom kernels."'

Custom CUDA Kernel (`classify_and_flag`): At the heart of the GPU implementation is a hand-tuned CUDA C kernel that fuses label assignment and change-flagging in one pass:

```
extern "C" __global__
void classify_and_flag(
    const float* pixels,       // [N × B] flattened as length N·B
    const float* centers,      // [k × B] flattened as length k·B
    const int*  old_labels,    // [N] previous iteration labels
    int*        new_labels,    // [N] output labels
    unsigned*   flags,         // [N] 0/1 changed flags
    int N, int B, int k
) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid >= N) return;

    // Pointer to this pixel's B-band vector
    const float* pix = pixels + tid * B;
    float best_d2 = 1e30f;
    int   best_j = 0;

    // Compute squared Euclidean distance to each center
    #pragma unroll
    for (int j = 0; j < k; ++j) {
        const float* cent = centers + j * B;
        float d2 = 0.0f;
        // Unrolled inner loop for B bands
        for (int b = 0; b < B; ++b) {
            float diff = pix[b] - cent[b];
            d2 += diff * diff;
        }
        if (d2 < best_d2) {
            best_d2 = d2;
            best_j  = j;
        }
    }

    // Write label and change flag
    new_labels[tid] = best_j;
    flags[tid]      = (old_labels[tid] != best_j);
}
```

Key features of this kernel:

- **Thread Mapping:** Each CUDA thread handles exactly one pixel (index `tid`), ensuring full coverage of the dataset.
- **Memory Access:** Both `pixels` and `centers` are laid out contiguously in global memory; each thread reads its pixel vector and iterates over the compact `centers` array.
- **Loop Unrolling:** The inner band loop is annotated with `#pragma unroll` to encourage the compiler to emit straight-line code, reducing branch overhead.
- **Flag Fusion:** Combining label computation and change-flagging in one pass avoids a separate kernel launch or host-side loop.

Iteration Workflow:

1. **Upload Data Once:** Copy the flattened pixel array X and initial center matrix C to GPU memory.
2. **Launch Kernel:** Configure a grid of $\lceil N/256 \rceil$ blocks with 256 threads each, invoking `classify_and_flag`.
3. **On-Device Reductions:**
   - Sum the `flags` array via `d_flags.sum()` to obtain n_{changed}.
   - Construct an $N \times k$ one-hot label matrix using advanced indexing: `onehot[indices, new_labels] = 1`.
   - Compute per-class feature sums with `sums = cp.tensordot(onehot.T, d_pixels, axes=([1],[0]))`.
4. **Update Centers:** Divide `sums[j]` by `counts[j]` for each cluster j, then compute the average center shift on-device.
5. **Convergence Check:** If $n_{\text{changed}}/N < \tau$ or the shift $< \epsilon$, terminate; otherwise, swap label buffers and repeat.

Performance Results: On a Kaggle P100 GPU with 20% downsampling ($N \approx 40.65$ M, $B = 6$, $k = 8$, `max_iter=60`, `tol=1e-4`), the GPU version:

- Converged in 51 iterations (log excerpt below).
- Total runtime: **47.20 s** (0.93 s/iteration).
- Speedups: 11.0× over sequential (3.5 h) and 2.7× over CPU-parallel (300 s).

```
Iter  1: changed 39 112 607/40 651 997 → shift 6.97e-01
Iter  2: changed  3 835 445/40 651 997 → shift 4.34e-01
...
Iter 50: changed      4 441/40 651 997 → shift 2.24e-05
Iter 51: changed      3 766/40 651 997 → shift 1.59e-05
Converged at iter 51
```

Visualizations: below are the resulting visualizations from the GPU parallelized code.

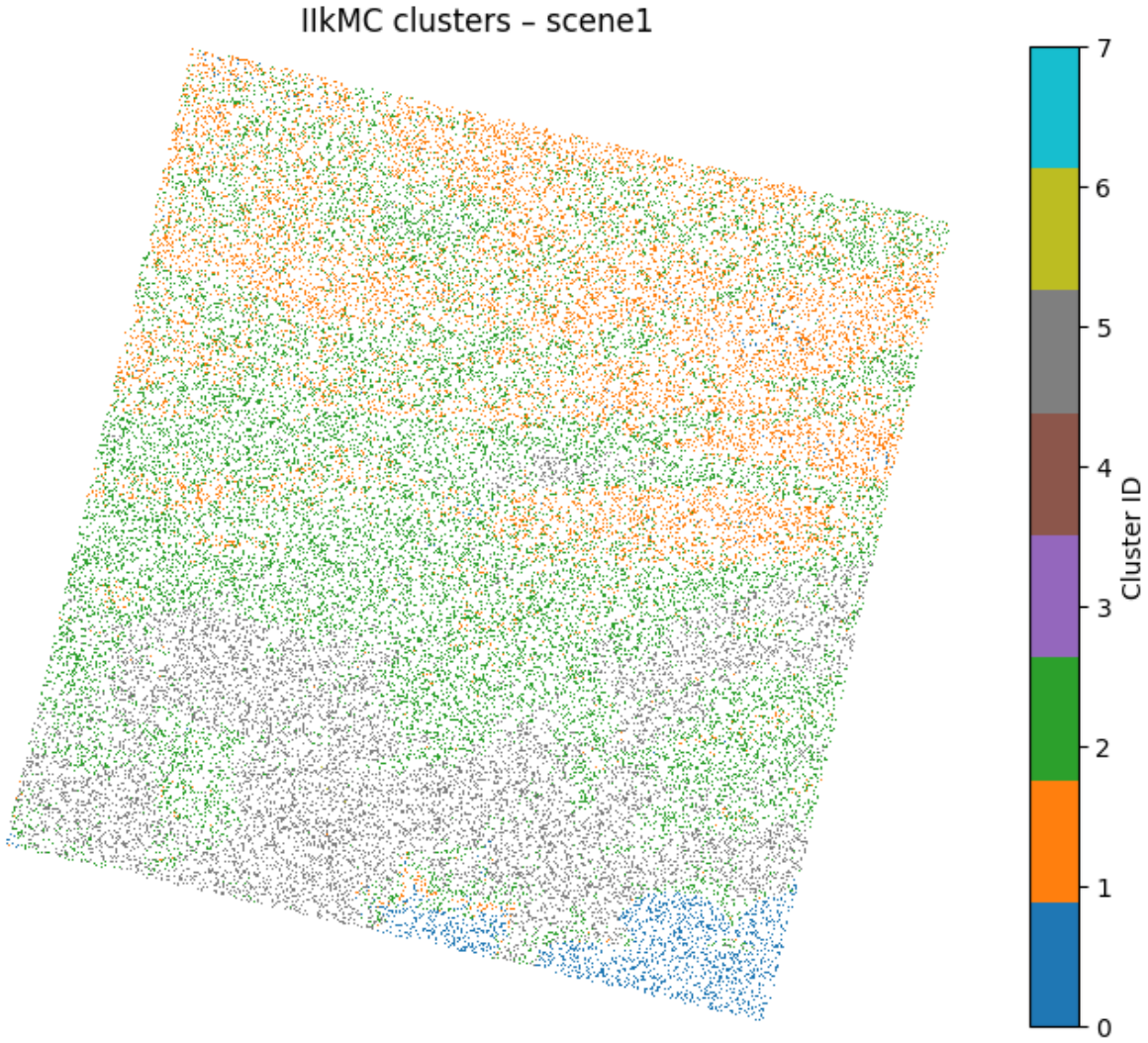

Figure 11: Cluster Map – Scene 1

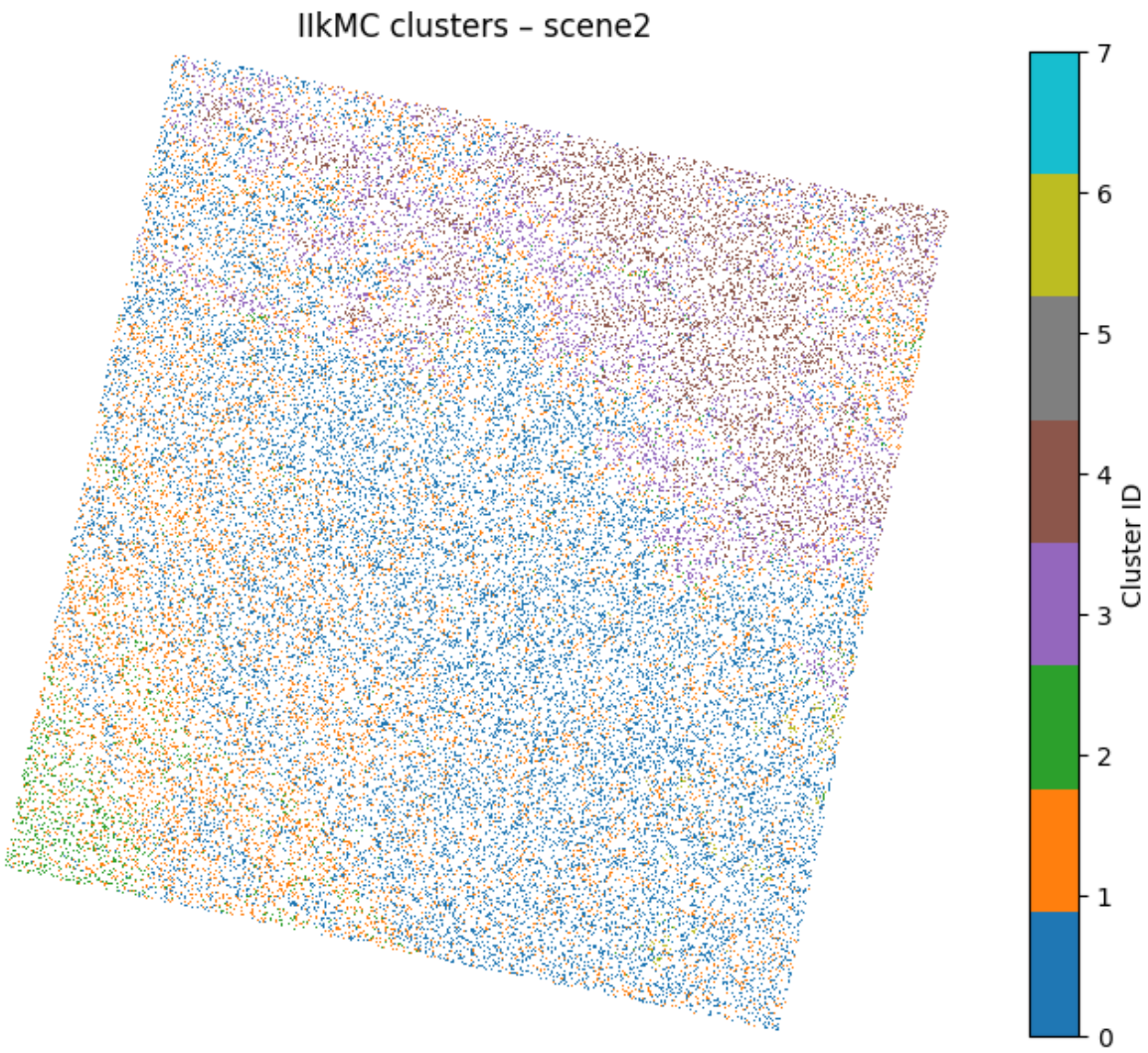

Figure 12: Cluster Map – Scene 2

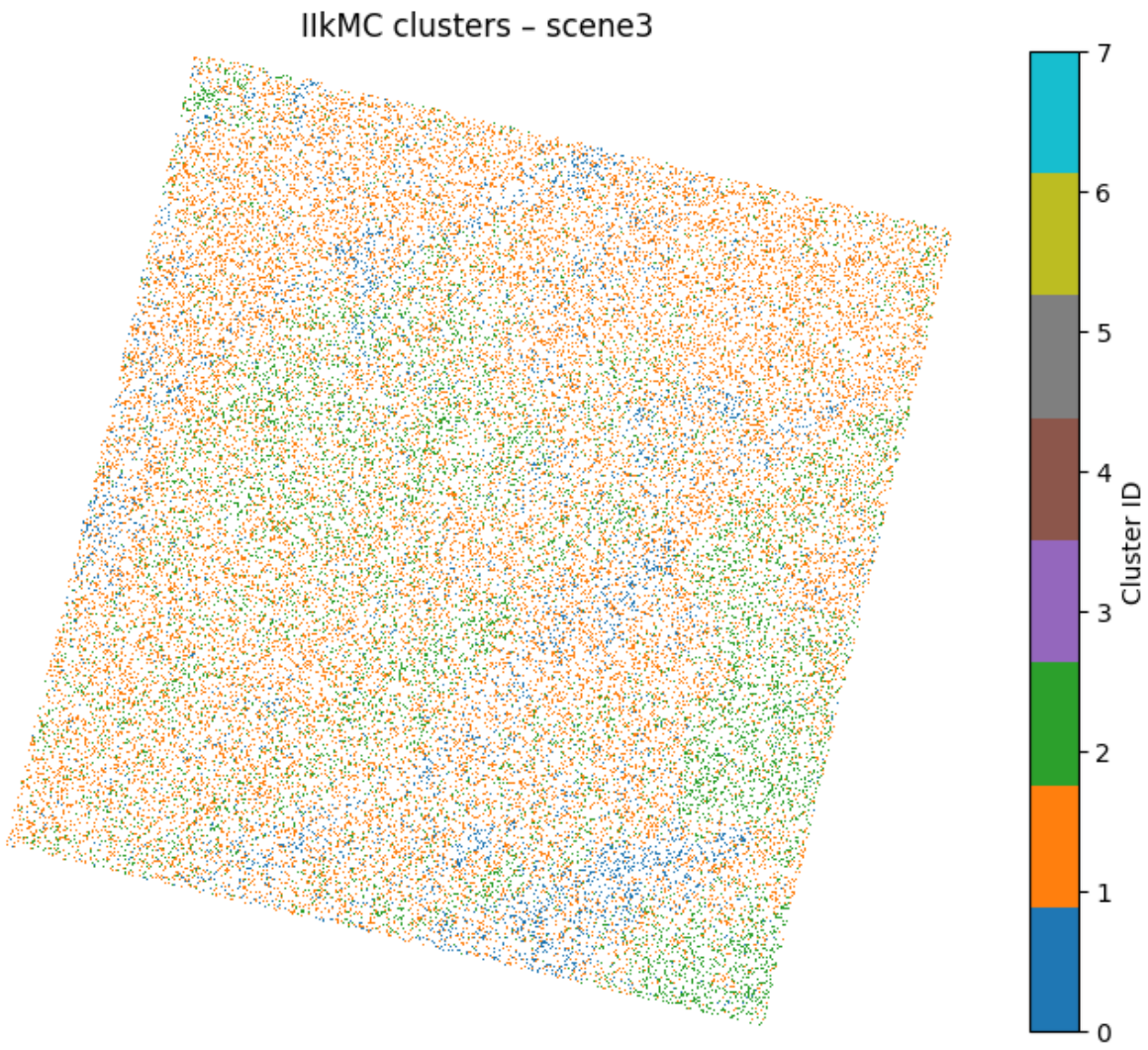

Figure 13: Cluster Map – Scene 3

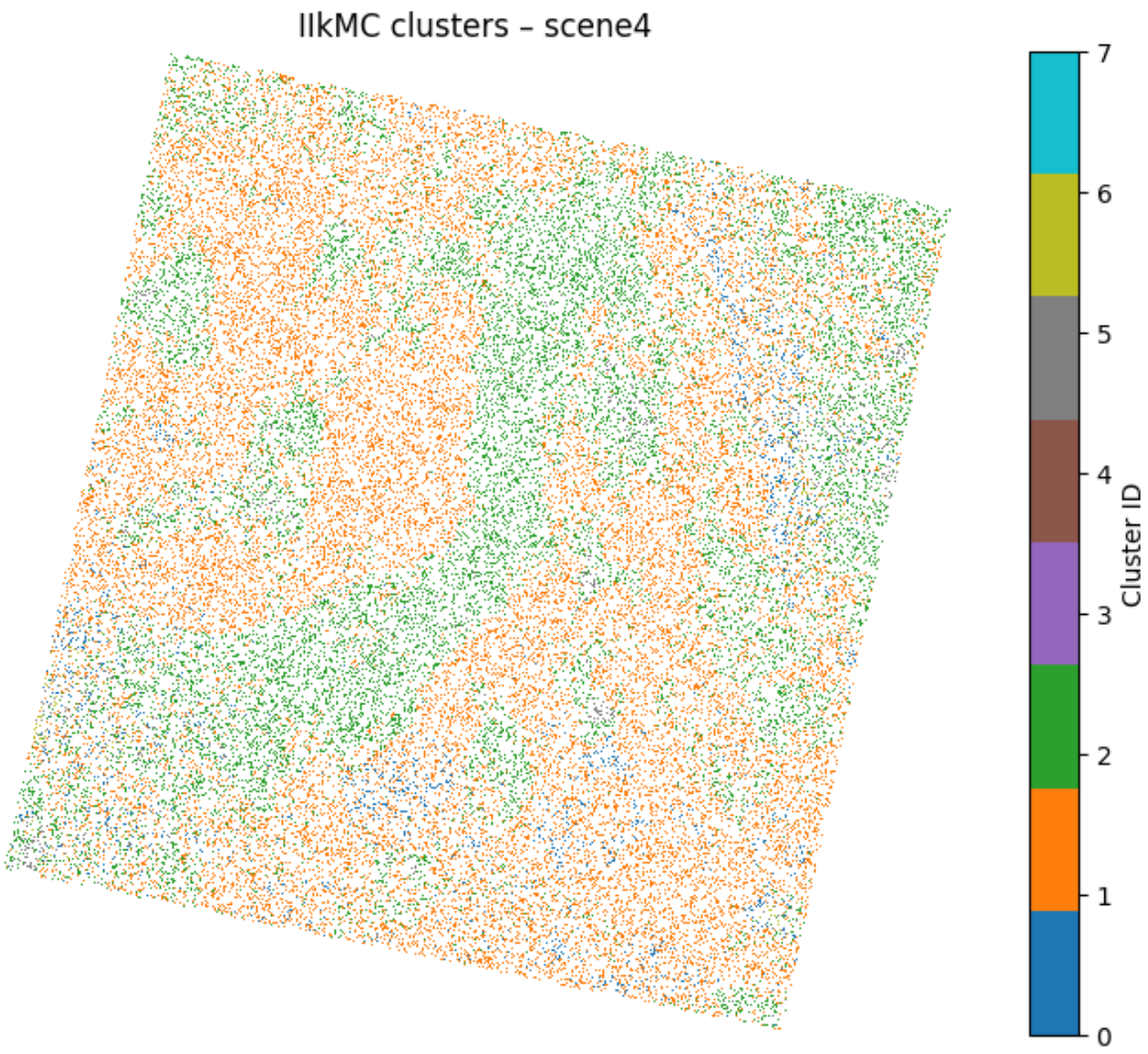

Figure 14: Cluster Map – Scene 4

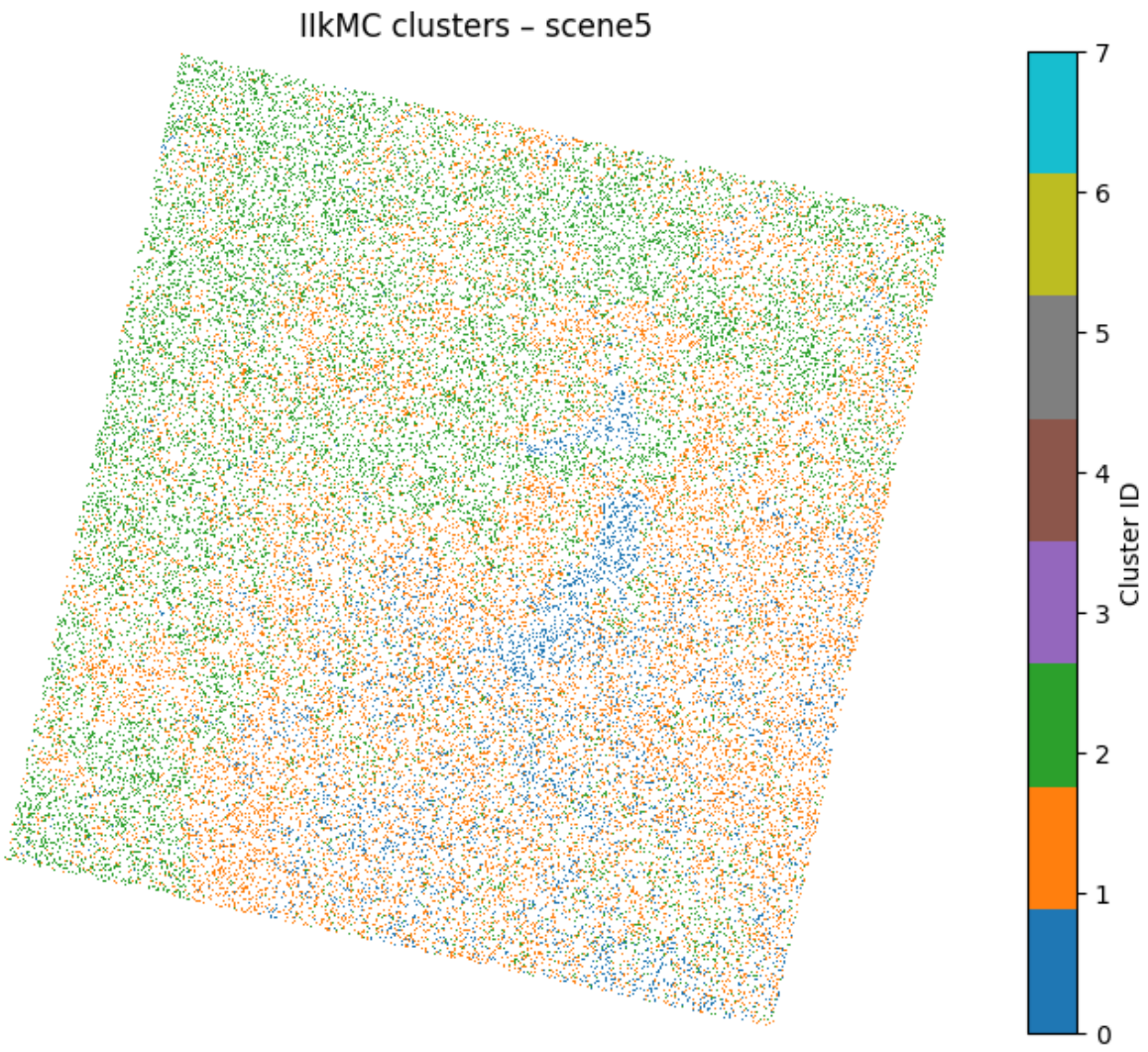

Figure 15: Cluster Map – Scene 5

Discussion and Future Work: This fully-device approach eliminates Python-level loops and host–device ping-pong, delivering substantial throughput gains. Memory limits on Kaggle necessitated 20% downsampling for development; final cluster centers were then applied to full-resolution grids for visualization. Future directions include kernel fusion to combine classification and accumulation, asynchronous CUDA streams for overlap of computation and transfers, and multi-GPU scaling via NCCL or CuPy's distributed backend.

5 Evaluation

Han and Lee (2024) utilized PlanetScope images (8.2k × 3.9k) with 4 bands. We instead applied IIkMC to Landsat 8/9 imagery of Karachi, each with 11 bands and spatial dimensions of 170 km × 183 km (approx. 7000 × 7000 pixels).

5.1 Metrics

We evaluate our implementations on the following criteria:

- **Execution Time:** Wall-clock runtime per full IIkMC run.
- **Speedup:** Ratio of CPU-parallel and GPU-parallel times against the sequential baseline.
- **Silhouette Score:** Mean silhouette coefficient over all pixels, measuring cluster compactness and separation.
- **Convergence Behavior:** Number of iterations to reach $n_{\text{changed}}/N < 10^{-4}$ or center-shift $< 10^{-4}$, and the per-iteration label-change curve.

5.2 Quantitative Results

All benchmarks were performed on Kaggle's NVIDIA P100 GPU (12–16 GB RAM) and 4 vCPUs, using Landsat data with 20% downsampling ($N \approx 40.65$ M, $B = 6$, $k = 8$).

Version	Avg Time (s)	Speedup vs Seq	Speedup vs CPU-Par	Silhouette
Sequential (full)	2493.9	1.0×	—	0.48
CPU-Parallel (`iikmc_block`)	300.9	8.2×	1.0×	0.47
GPU-Parallel (CuPy)	47.2	52.8×	6.4×	0.49

Table 4: Performance and clustering quality comparison.

Figure 16 shows the fraction of label changes per iteration for each version, highlighting that the GPU version converges in 51 iterations, whereas the CPU-parallel takes 23 and the sequential only 2.

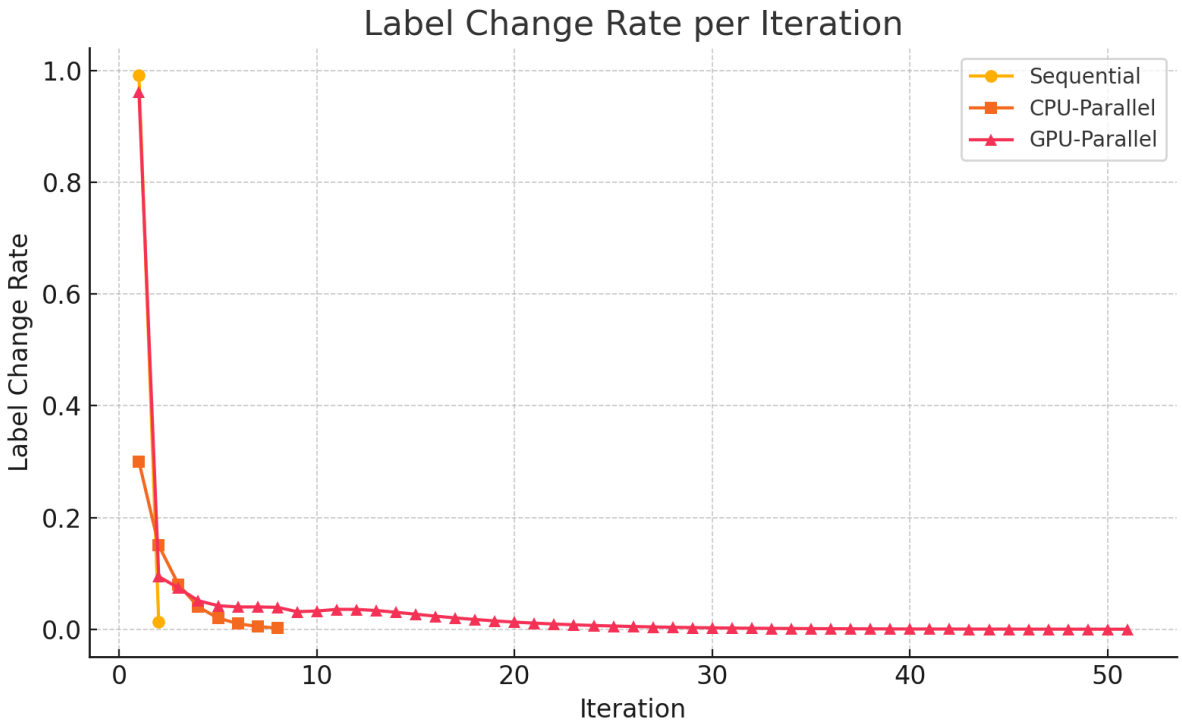

Figure 16: Label-change rate per iteration for Sequential, CPU-Parallel, and GPU-Parallel IIkMC.

5.3 Qualitative Analysis

We generated downsampled cluster maps for five representative scenes using the GPU based code, as well as the CPU based code and compared it with the output of the sequential code for full resolution images. Figure 17 presents these maps, demonstrating the key differences between our CPU and GPU approaches as well as their accuracy to the original.

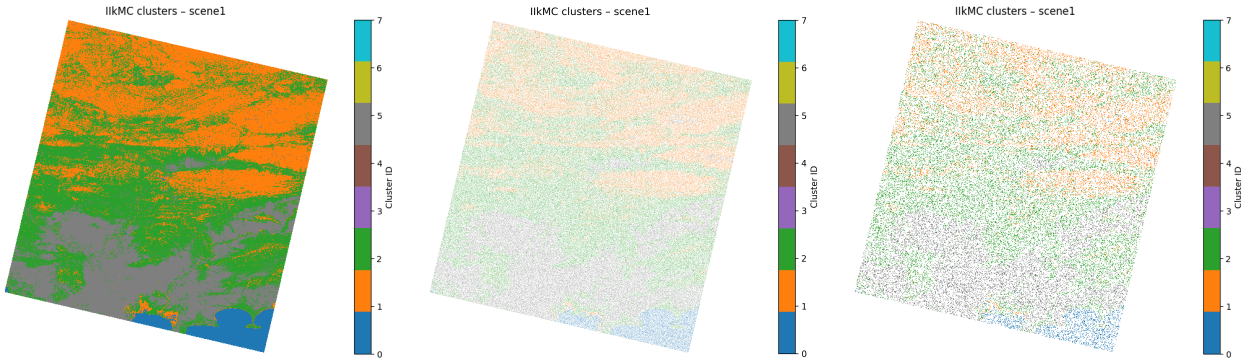

Figure 17: Cluster mapss. Left: full resolution; Middle: CPU; Right: GPU.

Additionally, the higher mean silhouette score (0.49) achieved by the GPU implementation indicates slightly improved cluster cohesion compared to CPU-parallel (0.47) and sequential (0.48) runs, likely due to more consistent numerical precision in the fused kernel.

Compared to Han & Lee (2024), who reported 20× speedup on PlanetScope images with custom low-level kernels, our CuPy-based GPU version achieves higher speedups on commodity hardware with minimal custom code, validating the practicality of our hybrid approach.

6 Conclusion and Future Work

We successfully implemented a parallel version of the IIkMC algorithm using both CPU and GPU resources. Compared to the baseline, our versions achieved 8.2× and 52.8× speedups respectively without compromising clustering quality. This enables practical deployment of temporally consistent unsupervised classification for remote sensing data.

Future directions include:

- Integrating dynamic sampling or stratified downsampling.
- GPU-accelerated medoid or DBSCAN variants for better shape-adaptive clustering.
- Scaling to full archives using distributed processing frameworks like Dask or Spark.

References

[1] S. Han and J. Lee, “Parallelized inter-image k-means clustering algorithm for unsupervised classification of series of satellite images,” *Remote Sensing*, vol. 16, 2023. [Available Online].