Intern/Project Trainee Report

June 2025

Reg No:

Distributed Parallel Processing of Drone-Captured Images for Orthomosaic Generation using Apache Spark

Keshav Jindal

Indian Institute of Information Technology, Guwahati

Shri Siddhartha Bhuyan

Scientist
Department of Space
Government of India
North Eastern Space Applications Centre,
Meghalaya

North Eastern Space Applications Centre

Department of Space, Government of India
Umiam – 793103, Meghalaya

Contents

Contents 1

List of Figures 3

Abstract 4

1 Introduction 5
- 1.1 Problem Statement . . . 5
- 1.2 Project Objectives . . . 6

2 Background 8
- 2.1 The Fundamentals of Image Stitching . . . 8
- 2.2 Popular Image Stitching and Photogrammetry Software . . . 10

3 Overview of Apache Spark 12
- 3.1 The Core Architecture: Driver and Executors . . . 12
- 3.2 Resilient Distributed Datasets (RDDs) . . . 13
- 3.3 Transformations and Actions: The Lazy Evaluation Model . . . 14

4 System Architecture 16
- 4.1 Pipeline Stages . . . 16

5 Spark Cluster Setup Guide 19
- 5.1 Installing Spark (Standalone on Windows/Linux) . . . 19
- 5.2 Local Spark Master Configuration . . . 21

6 Stitching Workflow Explained 22
- 6.1 Image Loading and Preparation . . . 22
- 6.2 Distributed Feature Extraction . . . 22
- 6.3 Distributed Matching Strategy . . . 23
- 6.4 Homography Estimation and Validation . . . 23
- 6.5 Warping, Blending, and Cropping . . . 24
- 6.6 Post-Processing (Optional) . . . 24

7 Results & Output 26
- 7.1 Quantitative Performance Analysis . . . 26
- 7.2 Qualitative Output: The Final Panorama . . . 27

8 Challenges Faced and Solutions **32**
8.1 PySpark Serialization Limitations with OpenCV Data 32
8.2 Managing the Image Resolution vs. Memory Trade-off 32
8.3 Inconsistent Feature Density in Drone Images 33
8.4 Designing Fallbacks for Poor Matches 33

9 Future Work and Potential Enhancements **34**
9.1 Integrate GPS Metadata for Geo-Referenced Stitching 34
9.2 Migrate from RDDs to DataFrames and the Spark SQL Engine 34
9.3 Implement Global Optimization with Bundle Adjustment 35
9.4 Implement Graph-Based Stitching for Unordered Collections 35
9.5 Develop a GUI or Web Dashboard for Usability 36

10 Conclusion **37**

Bibliography **38**

List of Figures

7.1 Output Panorama Generated from Dataset 28
7.2 Enhanced Post Processed Output 29
7.3 Sharpened Post Processed Output 30
7.4 Bilateral Post Processed Output 31

Abstract

Creating high-resolution panoramas via image stitching is a process constrained by significant computational complexity, often overwhelming the resources of a single workstation. This paper presents a scalable and robust architectural solution that utilizes the Apache Spark framework to distribute the processing load across multiple nodes, enabling efficient handling of large aerial datasets typically generated by drone platforms. A parallelized pipeline, implemented in Python with the PySpark and OpenCV libraries, distributes the fundamental stages of image acquisition, feature detection, and correspondence matching across a cluster, significantly accelerating these compute-intensive operations.
The system's resilience is fortified by a sophisticated sequential chaining algorithm, which incorporates skip-ahead logic to bridge gaps between non-overlapping or poorly matched image pairs. For geometrically ambiguous or low-overlap cases, the system employs a fallback mechanism using affine transformation models, ensuring structural continuity even under imperfect conditions. Images are globally aligned through a homography composition process, followed by high-fidelity warping and simple feather-blending techniques to produce coherent and visually consistent panoramas.
Additionally, the framework integrates optional post-processing filters such as CLAHE-based contrast enhancement and adaptive smoothing, further improving the visual quality of the final output. Empirical performance metrics reveal substantial reductions in execution time for the parallelized stages, with efficient memory handling and scalability across varying image volumes. This document provides a comprehensive exposition of the system's architecture, its Spark-based implementation strategy, detailed guidelines for configuring and deploying a local cluster, and an in-depth analysis of the results, thereby establishing the efficacy of Spark for large-scale image stitching in computer vision and remote sensing applications.

Chapter 1

Introduction

1.1 Problem Statement

The creation of high-resolution panoramic images through the process of image stitching is a cornerstone of modern applications, spanning diverse fields such as satellite-based Geographical Information Systems (GIS), immersive virtual reality experiences, digital heritage preservation, and large-scale medical imaging. The fundamental goal of stitching is to combine multiple, smaller images with overlapping fields of view into a single, seamless, and high-resolution mosaic. However, the algorithmic and computational complexity inherent in this process presents a significant barrier to scalability.

The core challenge lies in the computationally expensive nature of the underlying computer vision tasks. The process involves identifying thousands of unique feature points in each image, generating descriptive vectors for each point, and then performing a vast number of comparisons to find reliable correspondences between image pairs. As the number of images and their individual resolutions increase, the problem's complexity grows exponentially. This gives rise to two critical, often insurmountable, bottlenecks for traditional single-machine architectures:

- **Computational Bottleneck (CPU Overload):** A sequential, single-threaded execution of these tasks is prohibitively slow. For a set of N images, the feature matching stage alone may require comparing $\frac{N(N-1)}{2}$ pairs. For a dataset of 50 images, this results in over 1,200 individual matching operations. When each operation involves comparing thousands of feature descriptors, the total processing time can extend from hours to days, rendering the system impractical for any time-sensitive application.

- **Memory Bottleneck (RAM Exhaustion):** Modern digital cameras produce images of 24, 48, or even 100 megapixels. Loading just a few dozen of these images can consume several gigabytes of system memory. This is compounded by the memory required to store the extracted feature data. For instance, a single SIFT descriptor requires 512 bytes of storage; an image with one million keypoints would thus generate approximately half a gigabyte of descriptor data alone. For a large dataset, the total memory footprint can easily exceed the 16 GB or 32 GB of RAM available on a typical high-end workstation, leading to system thrashing or, more commonly, a complete application crash due to memory exhaustion.

Therefore, the central problem this project addresses is the fundamental architectural mismatch between the demands of large-scale image stitching and the resource limitations of a single-machine environment. The traditional paradigm of vertical scaling (i.e., using a more powerful single computer) offers diminishing returns and is ultimately unsustainable. This creates a scalability crisis that prevents researchers, artists, and engineers from processing the vast and high-resolution image datasets that are now commonplace. This project posits that a shift to a horizontal scaling paradigm, using a distributed computing framework, is the only viable path forward.

1.2 Project Objectives

To systematically address the defined problem, this project is guided by a set of clear and measurable objectives. Each objective represents a critical component in the development of a comprehensive and effective solution.

1. **To design and implement an end-to-end image stitching pipeline capable of running on a distributed Spark cluster.**

   This objective goes beyond merely writing code; it involves architecting a complete system that manages the entire workflow from raw input files to a final, blended panoramic image. The design must be inherently distributed, taking into account:

   - Data partitioning
   - Network serialization
   - Clear division of labor between the Spark driver and executor nodes

   The final implementation will be a self-contained application capable of executing the entire stitching process in a distributed fashion.

2. **To parallelize the computationally intensive stages of feature extraction and matching.**

   This is the core performance objective of the project. The most significant computational bottlenecks—specifically feature extraction and feature matching—will be identified and redesigned to run in a massively parallel manner on a Spark cluster. The goal is to transform these “embarrassingly parallel” tasks into distributed jobs that:

   - Fully leverage the computational resources of the cluster
   - Drastically reduce processing time
   - Enable support for datasets too large for single-machine processing

3. **To build a robust system that can handle potential failures in matching between adjacent images by incorporating fallback mechanisms.**

   Real-world datasets are often imperfect and may contain:

   - Images with insufficient overlap

- Lens distortion or motion blur

To ensure robustness, the system will incorporate:

- *Skip-ahead logic*, to bypass failed matches and continue the stitching chain
- *Affine transformation fallback*, to provide a less constrained alternative when strict homography estimation fails

4. **To provide a comprehensive guide for setting up a local Spark environment and running the application.**

   A complex distributed system must be reproducible and accessible. This objective involves creating a step-by-step setup guide covering:

   - Installation of prerequisites (Java, Apache Spark)
   - Configuration of a local standalone Spark cluster (master and worker nodes)
   - Installation of required Python dependencies
   - Command-line execution of the full application

Chapter 2

Background

2.1 The Fundamentals of Image Stitching

Image stitching is a well-established field in computer vision, famously detailed by Richard Szeliski (2006) and further elaborated in his comprehensive text *Computer Vision: Algorithms and Applications* (2010). The process is not a single operation but a multi-stage pipeline where the output of each stage serves as the input for the next. A robust implementation requires a deep understanding of each component.

The first and perhaps most critical step is to identify a set of stable and distinctive interest points, or *keypoints*, within each image. These keypoints must be robust to geometric and photometric transformations, meaning they should be detectable even if the image is rotated, scaled, or captured under different lighting conditions. Once a keypoint is located, its local neighborhood is analyzed to generate a numerical vector known as a *descriptor*. This descriptor acts as a unique "fingerprint" for the keypoint.

Several algorithms have been developed for this purpose, each with its own trade-offs between accuracy, robustness, and computational cost:

- **Scale-Invariant Feature Transform (SIFT):** Proposed by David Lowe (1999, 2004), SIFT is widely considered the gold standard for its exceptional robustness. It uses a Difference of Gaussians (DoG) approach to detect keypoints in a scale-space pyramid, making them invariant to image scale and orientation. The SIFT descriptor is a 128-dimensional vector representing local image gradients. While highly distinctive and effective, its high dimensionality and floating-point nature make it computationally expensive to compute and match. SIFT was historically patented, which limited its use in commercial applications, but those patents have since expired.

- **Oriented FAST and Rotated BRIEF (ORB):** Developed by Ethan Rublee et al. at Willow Garage (2011), ORB was designed as a fast and efficient open-source alternative to SIFT. It combines the FAST (Features from Accelerated Segment Test) keypoint detector with a modified version of the BRIEF (Binary Robust Independent Elementary Features) descriptor. Unlike SIFT's high-dimensional float vectors, the ORB descriptor is a binary string of 256 or 512 bits. This allows for extremely fast matching using the Hamming distance (a simple bitwise XOR operation followed by a population count), making it orders of magnitude faster than SIFT's Euclidean distance matching.

- **Accelerated-KAZE (AKAZE):** Proposed by Alcantarilla et al. (2013), AKAZE and its predecessor KAZE operate in a non-linear scale-space. This approach is more effective at preserving object boundaries and is more robust to noise and blurring compared to the Gaussian scale-space used by SIFT. AKAZE provides a high-quality, floating-point descriptor similar to SIFT but with improved performance and robustness characteristics, offering a compelling balance between the speed of ORB and the accuracy of SIFT.

This project leverages a *multi-detector strategy*, attempting to extract features using all three of these methods. This hybrid approach increases the probability of finding reliable features across a diverse range of image types and conditions.

With descriptors extracted from two images, the next step is to find corresponding pairs. The most common approach is to use a *Brute-Force Matcher*, which, for every descriptor in the first image, iterates through all descriptors in the second image to find the closest one based on a distance metric.

However, this naive matching process generates many incorrect matches, or *outliers*. A robust pipeline must incorporate techniques to filter these out:

- **Lowe's Ratio Test:** As mentioned previously, this is the most effective technique for rejecting ambiguous matches. By ensuring a significant margin between the best and second-best match, it filters out keypoints that are not sufficiently distinctive.

- **Symmetry Test (Cross-Checking):** This test enforces a bijective (one-to-one) correspondence. A match between keypoint A and keypoint B is only considered valid if A is the best match for B and B is the best match for A. This effectively eliminates situations where multiple keypoints in one image map to the same keypoint in another.

- **Random Sample Consensus (RANSAC):** After initial filtering, a geometric model (e.g., a homography) is estimated from the putative matches. RANSAC is an iterative algorithm that robustly fits this model even in the presence of a large number of remaining outliers. It repeatedly selects a random minimal subset of correspondences, computes a model, and then counts the number of other correspondences (the inliers) that agree with this model. The model with the largest set of inliers is chosen as the final, robust solution.

Once a reliable geometric transformation (typically a 3×3 homography matrix) is found between two images, one image must be warped to align with the other's coordinate system. This is done using a *perspective warp*. When extending this to multiple images, all images are typically warped into a common panoramic canvas.

The final step is to blend the overlapping regions to create a seamless visual appearance. This is crucial for hiding the edges between the original images. Common blending techniques include:

- **Feathering (Alpha Blending):** A simple and fast technique where the intensity of a pixel in the overlapping region is a weighted average of the corresponding pixels from the source images. The weight is typically proportional to the pixel's distance from the edge of the image, creating a smooth transition.

- **Multi-Band Blending:** A more sophisticated technique proposed by Burt and Adelson (1983). It involves decomposing the source images into different spatial frequency bands using Laplacian pyramids. The blending is performed separately on each band, and the result is reconstructed. This method produces nearly invisible seams, even when there are significant lighting or exposure differences between the images.

2.2 Popular Image Stitching and Photogrammetry Software

There exist several advanced, engineering-grade applications that are widely used in aerial and drone mapping. These tools go beyond simple panoramic stitching to create *georeferenced orthomosaics*—a single, geometrically corrected image of the Earth's surface with a uniform scale.

Pix4Dmapper (Pix4D)

Pix4Dmapper is a leading commercial photogrammetry software and is considered the industry standard for professional drone mapping. It can process thousands of aerial images to produce high-resolution, survey-grade 2D orthomosaics and 3D models.

- Pix4D uses a highly sophisticated form of image stitching, including **Bundle Adjustment** for global optimization, which accounts for camera positions, orientations, and lens distortions.
- The software is heavily optimized to utilize all available CPU cores and GPU resources on a workstation.
- For extremely large projects, Pix4D offers a cloud-based processing service. While this is conceptually a distributed system, it remains a proprietary black-box solution.

OpenDroneMap (ODM)

OpenDroneMap is a powerful, open-source command-line toolkit for processing aerial drone imagery. It provides a complete end-to-end photogrammetry pipeline that can generate orthomosaics, 3D point clouds, and digital elevation models (DEMs).

- ODM is built on a foundation of established computer vision and photogrammetry libraries and is one of the most prominent open-source alternatives to commercial tools like Pix4D.
- While it can run on a single powerful machine, its modular architecture allows its components to be containerized (e.g., using Docker).
- Projects like **ClusterODM** aim to distribute the workload across multiple machines, making ODM philosophically similar to the distributed approach used in this project.

- However, ODM is specifically tailored for geospatial data and mapping applications.

Architectural Limitations and Project Context

A common architectural theme among these powerful tools is that, when run on a single workstation, they function as *monolithic applications*. Although heavily optimized, they are fundamentally constrained by the physical limitations of a single computer's RAM and CPU/GPU capabilities.

- **Scalability bottleneck:** As the size and resolution of image datasets continue to grow, these tools encounter performance limitations that cannot be resolved through further vertical scaling.
- **Distributed alternative:** This project explores a horizontally scalable, distributed computing approach using Apache Spark, which addresses these limitations by parallelizing computational tasks across a cluster.

By providing an open, transparent, and general-purpose distributed image stitching solution, this project complements existing tools and expands the scalability frontier for image-based analysis and photogrammetry.

Chapter 3

Overview of Apache Spark

To understand the architectural choices made in this project, it is essential to have a foundational knowledge of Apache Spark. Apache Spark is not merely a tool for parallel execution; it is a powerful, open-source, unified analytics engine designed for large-scale data processing and machine learning. It was developed to address the limitations of earlier distributed computing paradigms like Hadoop MapReduce, offering significant improvements in speed, ease of use, and versatility.

3.1 The Core Architecture: Driver and Executors

Apache Spark operates on a *master-worker architecture*, which consists of two primary components: the **Driver Program** and the **Executor Processes**.

Driver Program

The driver is the heart of a Spark application. It is the process that runs the `main()` function, creates the `SparkContext`, and is responsible for converting the user's code into a series of jobs and tasks. Specifically, the driver:

- Analyzes the logical execution plan.
- Schedules and distributes tasks across the cluster.
- Monitors execution progress and handles failures.

In our project, the driver also performs the final centralized stages of the stitching pipeline, such as:

- Calculating global homographies through transformation composition.
- Blending all warped images to produce the final panoramic output.

Executor Processes

Executors are processes launched on the **worker nodes** of the Spark cluster. Each executor:

- Executes the tasks assigned by the driver.
- Utilizes allocated resources such as CPU cores and memory.
- Stores intermediate data in memory or on disk for reuse across stages.

In the context of this project, executors are primarily responsible for:

- Performing feature extraction using detectors like SIFT, ORB, and AKAZE.
- Conducting feature matching between pairs of images in parallel.

Distributed Efficiency

This separation of responsibilities between the driver and the executors allows Spark to efficiently manage cluster resources. By isolating orchestration from execution, the architecture supports high-throughput, fault-tolerant distributed computation—an essential requirement for large-scale image stitching pipelines such as the one developed in this project.

3.2 Resilient Distributed Datasets (RDDs)

The fundamental data abstraction in Apache Spark is the **Resilient Distributed Dataset (RDD)**. An RDD is an immutable, partitioned collection of records that can be operated on in parallel across a cluster. RDDs form the backbone of distributed computing in Spark and offer powerful guarantees for performance and fault tolerance.

Distributed

The data in an RDD is split into multiple *partitions*, with each partition capable of residing on a different node in the cluster. This enables Spark to perform computations in parallel, distributing the workload and achieving high throughput.

Resilient

RDDs are inherently fault-tolerant. They achieve this by maintaining a *lineage graph*—a record of all the transformations (e.g., `map`, `filter`) applied to the original dataset to produce the current RDD. If a partition is lost due to a node failure, Spark can use the lineage information to automatically recompute the lost partition, allowing the job to continue without interruption or manual intervention.

Immutable

RDDs are immutable; once created, they cannot be altered. Any transformation on an RDD (such as `map`, `flatMap`, or `filter`) produces a new RDD rather than modifying the existing one. This immutability simplifies reasoning about program behavior and enhances fault tolerance, as intermediate states can be reliably reproduced using the lineage graph.

In the context of this project, RDDs are used to:

- Distribute raw images across worker nodes for parallel feature extraction.
- Execute matching logic in parallel for all possible or adjacent image pairs.
- Collect and process feature descriptors and match data in a fault-tolerant and scalable manner.

RDDs enable the stitching pipeline to maintain high performance even when dealing with large volumes of high-resolution image data, all while preserving robustness in the face of hardware or computation failures.

3.3 Transformations and Actions: The Lazy Evaluation Model

Spark operations on *Resilient Distributed Datasets (RDDs)* are divided into two categories: **transformations** and **actions**.

Transformations

Transformations are operations that define a new RDD from an existing one. These operations are *lazy*, meaning that Spark does not execute them immediately. Instead, it constructs a *Directed Acyclic Graph (DAG)* of computation that outlines all the necessary steps to produce the desired output. Examples of common transformations include:

- `map`: Applies a function to each element in the RDD.
- `filter`: Selects elements that satisfy a predicate.
- `flatMap`: Similar to `map`, but allows returning multiple output elements for each input element.

Actions

Actions are operations that trigger the actual execution of the computation graph and either return a result to the driver program or write output to external storage. Examples of common actions include:

- `collect`: Returns all elements of the RDD to the driver.

- `count`: Returns the number of elements in the RDD.
- `saveAsTextFile`: Saves the RDD contents to a specified location in the file system.

The Lazy Evaluation Advantage

The lazy evaluation model is a cornerstone of Spark's performance. By postponing execution until an action is called, Spark's optimizer (Catalyst) can:

- Analyze the entire DAG of transformations.
- Reorder operations to minimize data movement.
- Combine multiple narrow transformations into a single stage (pipeline fusion).
- Generate the most efficient physical execution plan.

In our project, the stages for parallel **feature extraction** and **feature matching** are implemented as a chain of transformations. The actual computation is triggered only when we call `.collect()` to retrieve the final match results to the driver for stitching and blending. This approach ensures both efficiency and scalability in processing large image datasets.

Chapter 4

System Architecture

The architecture of our distributed image stitching system is designed as a sequential pipeline of distinct processing stages. The key innovation lies in the strategic parallelization of the most computationally expensive stages using the Apache Spark framework. This hybrid design allows us to leverage the power of distributed computing for heavy lifting, while maintaining centralized control on the driver for orchestration and final composition.

4.1 Pipeline Stages

The end-to-end workflow is broken down into six major stages, as illustrated below:

[1] Load Images $\rightarrow$ [2] Feature Extraction $\rightarrow$ [3] Feature Matching $\rightarrow$ [4] Homography Estimation $\rightarrow$ [5] Warping & Blending $\rightarrow$ [6] Post-Processing

Each stage is optimized to balance distributed and local computation for maximum efficiency. Stages 1, 2, and 3 are executed in a distributed manner across the Spark cluster, while stages 4, 5, and 6 are performed centrally on the driver node.

[1] Load Images

The process begins by creating a Spark RDD from the list of input image paths. On each executor, the assigned images are:

- Loaded from disk
- Resized to a standard dimension to manage memory
- Encoded into a Base64 string for Spark-friendly transmission

This parallelizes the I/O and pre-processing workload across the cluster.

[2] Feature Extraction

This is the most computationally intensive stage and is fully parallelized. The RDD of encoded images is transformed, with each worker node responsible for extracting features from its partition of images using a combination of:

- SIFT (Scale-Invariant Feature Transform)
- ORB (Oriented FAST and Rotated BRIEF)
- AKAZE (Accelerated KAZE)

This *embarrassingly parallel* task scales nearly linearly with the number of available CPU cores in the cluster.

[3] Feature Matching

The system generates pairs of adjacent images, and this list of image pairs is distributed as an RDD. Each worker node receives one or more pairs and performs robust feature matching between them. By distributing this task:

- Hundreds of image pairs can be matched concurrently
- Lowe's Ratio Test and cross-checking ensure high-quality matches

The results, containing filtered point correspondences and match scores, are then collected back to the driver for further processing.

[4] Homography Estimation

This stage is performed on the driver node. Using the match results gathered from the cluster, the driver:

- Builds a robust chain of images based on successful matches
- Iteratively calculates the homography (or fallback affine transformation) for each link
- Validates each geometric model using RANSAC and inlier spread thresholds

Due to the need for a global view of all matches and sequential dependency, this stage is best suited for centralized execution.

[5] Warping & Blending

Once all global transformations are calculated, the driver:

- Creates a large canvas for the final panorama
- Warps each image onto this canvas using its computed transformation matrix
- Applies blending techniques (e.g., feathering or multiband blending) to create seamless transitions in overlapping regions

This step is inherently serial and requires large memory, making the driver the most suitable location for execution.

[6] Post-Processing

The completed panorama undergoes optional enhancement on the driver node before final output. These enhancements include:

- Sharpening using filters (e.g., unsharp masking)
- Contrast enhancement using CLAHE (Contrast Limited Adaptive Histogram Equalization)
- Denoising with bilateral or NL-means filters

Finally, the processed panoramic image is saved to disk as the output of the system.

Chapter 5

Spark Cluster Setup Guide

This chapter provides a concise guide to setting up a local standalone Apache Spark cluster on a single machine. This configuration is ideal for development, testing, and running the application without the need for a full multi-node cluster.

5.1 Installing Spark (Standalone on Windows/Linux)

Install Prerequisites

- Ensure you have **Java 11** or a compatible version installed, as Spark runs on the Java Virtual Machine (JVM).
- Install **Python 3.10** or a later version.

Download and Extract Spark

- Download a pre-built version of Apache Spark from the official website: `https://spark.apache.org/downloads.html`
- Extract the downloaded `.tgz` archive to a directory without spaces, such as:
  - `C:\Spark` on Windows
  - `/opt/spark` on Linux/macOS

Set Environment Variables

- **SPARK_HOME:** Set this to the directory where you extracted Spark.
- **PATH:** Add the `bin` directory from your Spark installation to your system's PATH.

  Linux/macOS:

```
export SPARK_HOME=/opt/spark
export PATH=$SPARK_HOME/bin:$PATH
```

Windows (in Command Prompt):

```
set SPARK_HOME=C:\Spark
set PATH=%SPARK_HOME%\bin;%PATH%
```

- **PYTHONPATH (Windows):** To ensure Python can find the necessary Spark libraries:

  ```
  set PYTHONPATH=C:\Spark\python;
                 C:\Spark\python\lib\py4j-*.zip;
                 C:\Spark\python\lib\pyspark.zip
  ```

- **PYSPARK_PYTHON:** Explicitly specify the Python interpreter for PySpark:

  ```
  set PYSPARK_PYTHON=C:\Users\user\AppData\Local\
                     Programs\Python\Python310\python.exe
  ```

Set Up Hadoop WinUtils (Windows Only)

Apache Spark on Windows requires certain Hadoop binaries, even if you are not using HDFS. To ensure compatibility, follow these steps:

1. Create a folder for Hadoop binaries:

   ```
   C:\Hadoop\bin
   ```

2. Download the appropriate version of `winutils.exe` for your Spark/Hadoop version from the following GitHub repository:

   https://github.com/steveloughran/winutils

3. Place the `winutils.exe` file in the `C:` directory.

4. Set the following environment variables:

   - **HADOOP_HOME:** Set this to the Hadoop folder.

     ```
     set HADOOP_HOME=C:\Hadoop
     ```

   - **Add to PATH:** Add `%HADOOP_HOME%\bin` to your system's PATH.

     ```
     set PATH=%HADOOP_HOME%\bin;%PATH%
     ```

This step resolves issues related to file system access on Windows and is required for Spark to interact properly with the local file system.

5.2 Local Spark Master Configuration

Once the environment is correctly configured, you can start a local Spark cluster consisting of a master and one or more workers on your machine.

Start the Master Node

Linux/macOS:

```
$SPARK_HOME/sbin/start-master.sh
```

Windows (Command Prompt):

```
C:\Spark\bin\spark-class org.apache.spark.deploy.master.Master
```

After starting, the master will display a URL (e.g., `spark://<your-hostname>:7077`). You can monitor the Spark cluster UI at:

http://localhost:8080

Start a Worker Node

In a new terminal window, connect a worker node to the master using the Spark URL provided above.

Linux/macOS:

```
$SPARK_HOME/sbin/start-worker.sh spark://<your-hostname>:7077
```

Windows (Command Prompt):

```
C:\Spark\bin\spark-class org.apache.spark.deploy.worker.Worker
    spark://<your-hostname>:7077
```

The worker will register with the master, and your local standalone Spark cluster will be ready to accept jobs.

Chapter 6

Stitching Workflow Explained

This chapter provides a detailed, step-by-step explanation of the image stitching pipeline implemented in this project. Each stage is designed to contribute to a final, seamless panorama, with computationally intensive tasks offloaded to the Spark cluster for parallel execution.

6.1 Image Loading and Preparation

The workflow begins with the ingestion of source images.

Directory Loading

The application targets a specified directory and identifies all image files with supported extensions (e.g., `.jpg`, `.png`). A sorted list of these file paths is created to maintain a logical sequence, which is crucial for the default adjacent-pair matching strategy.

Distributed Ingestion and Encoding

This list of paths is parallelized into a Spark RDD. Each worker node in the cluster receives a subset of these paths. On the worker:

- The image is loaded into memory using OpenCV.
- The image is resized to a maximum dimension to prevent out-of-memory errors.
- It is encoded into a lossless PNG format, then converted into a Base64 string.

This encoding step is critical, as it transforms the complex binary NumPy array into a simple text string that can be efficiently and safely transmitted across Spark stages.

6.2 Distributed Feature Extraction

This stage is the most computationally demanding and is fully parallelized.

Multi-Detector Strategy

To ensure robustness across various image types, a multi-detector strategy is used. For each image, features are extracted using three algorithms:

- **SIFT** — Highly robust to scale and rotation.
- **ORB** — Efficient with binary descriptors and fast matching.
- **AKAZE** — Offers a balance of speed and accuracy using nonlinear scale-space.

Parallel Execution via RDDs

Feature extraction for each image occurs concurrently across Spark executors. Each task is independent, making this an "embarrassingly parallel" problem that scales efficiently with the number of available CPU cores. The extracted descriptors and keypoints are stored and associated with their respective images.

6.3 Distributed Matching Strategy

Once features are extracted, the system finds correspondences between images.

Adjacent Pair Matching

The default strategy is to match only adjacent images (e.g., image 1 with 2, 2 with 3, etc.). The list of image pairs is distributed as an RDD across the cluster for parallel processing. This is efficient for video frames or sorted image sets. For unordered images, the system can be adapted to match all possible pairs.

Robust Matching Algorithm

Each worker runs a robust matching process for its assigned pairs:

- Iterates through SIFT, ORB, and AKAZE.
- Selects the best-performing descriptor based on match quality.
- Applies the following filters:
  - **Lowe's Ratio Test** — Filters out ambiguous matches.
  - **Cross-Checking** — Ensures symmetric matching between image pairs.

6.4 Homography Estimation and Validation

With a high-quality set of matches per image pair, the geometric transformation is computed on the driver node.

RANSAC for Robustness

The RANSAC algorithm is used to compute the 3×3 homography matrix for each matched pair. RANSAC tolerates outliers and finds the most consistent geometric transformation.

Affine Transformation Fallback

If RANSAC fails due to poor feature distribution or non-planar scenes, the system falls back to computing a simpler affine transformation. This fallback prevents the pipeline from failing and ensures stitching continuity.

6.5 Warping, Blending, and Cropping

This stage is performed centrally on the driver and produces the final stitched image.

Common Coordinate System

The corners of each image are transformed using their global homographies to compute the bounds of the panorama. A canvas is created to fit the entire field of view.

Feather Blending

Overlapping regions between images are blended using feathering. Each pixel's final value is a weighted average based on its distance from image boundaries, ensuring a smooth visual transition.

Automatic Cropping

To remove black borders resulting from image warping, the following steps are performed:

- Convert the panorama to grayscale.
- Threshold to find non-black regions.
- Compute a tight bounding box and crop accordingly.

6.6 Post-Processing (Optional)

Optional enhancements are applied to improve the final output quality.

Sharpening

An unsharp mask filter is applied to enhance edges and fine detail.

CLAHE Contrast Enhancement

Contrast Limited Adaptive Histogram Equalization (CLAHE) improves local contrast without amplifying noise, particularly effective in shadowed or unevenly lit areas.

Denoising Filters

A variety of smoothing filters can be applied:

- **Bilateral Filter** — Preserves edges while smoothing.
- **Non-Local Means** — Reduces noise by averaging similar patches.
- **Gaussian Blur** — Smooths high-frequency noise at the cost of sharpness.

These optional filters enhance the final appearance and visual coherence of the generated panorama.

Chapter 7

Results & Output

This chapter presents the tangible outcomes of the project, demonstrating the effectiveness of the distributed image stitching pipeline. The results are analyzed from two perspectives: the **quantitative performance metrics**, which highlight the speed and efficiency of the Spark-based approach, and the **qualitative output**, which assesses the visual quality of the final panoramic images.

For this analysis, a sample dataset was processed with the following characteristics:

- **Number of Images:** 60
- **Image Dimensions:** 5472 $\times$ 3648 pixels
- **Average File Size:** $\sim$10.5 MB per image
- **Total Dataset Size:** $\sim$652 MB

The Spark cluster was configured in local standalone mode with one master and two worker processes, each allocated 12 CPU cores and 12 GB of RAM.

7.1 Quantitative Performance Analysis

The application was instrumented to log the execution time of each major stage of the pipeline. The following timing summary provides critical insight into the performance characteristics and bottlenecks of the system:

Performance Discussion

The results clearly validate the architectural choices. Over 80% of the total execution time is spent on feature extraction and matching—precisely the stages parallelized using Spark. A conceptual single-threaded implementation would process these stages sequentially, drastically increasing the total runtime.

Moreover, as the number of executors increases (on a true distributed cluster), the execution time for these stages would further reduce. However, the serial steps like blending would remain constant, demonstrating the inherent limitations described by Amdahl's Law. This confirms that the system is highly scalable and efficient for large datasets.

Stage	**Execution Time (s)**	**Percentage**	**Analysis**
Image Loading	8.40	12.5%	Distributed loading and Base64 encoding show good performance with minimal overhead.
Feature Extraction	8.81	13.1%	Most time-consuming stage. Parallelism across two workers significantly reduces total time.
Feature Matching	49.43	73.3%	Also accelerated by Spark. Time is proportional to the number of image pairs.
Stitching & Blending	0.64	0.9%	Centralized on driver. Though relatively fast, this serial component limits scalability (Amdahl's Law).
Total Execution Time	**67.45**	**100%**	Full pipeline completed in under two minutes.

Table 7.1: Performance Summary

7.2 Qualitative Output: The Final Panorama

The primary goal of the pipeline is to produce a high-quality, seamless panoramic image from a set of unordered or ordered drone images. The following figure shows the visual result from the sample dataset.

Optional post-processing techniques, including CLAHE and denoising filters, further enhance the clarity and contrast of the final result.

These results substantiate the effectiveness of a distributed Spark-based pipeline for high-resolution image stitching, suitable for real-world applications such as remote sensing, urban mapping, and drone surveillance.

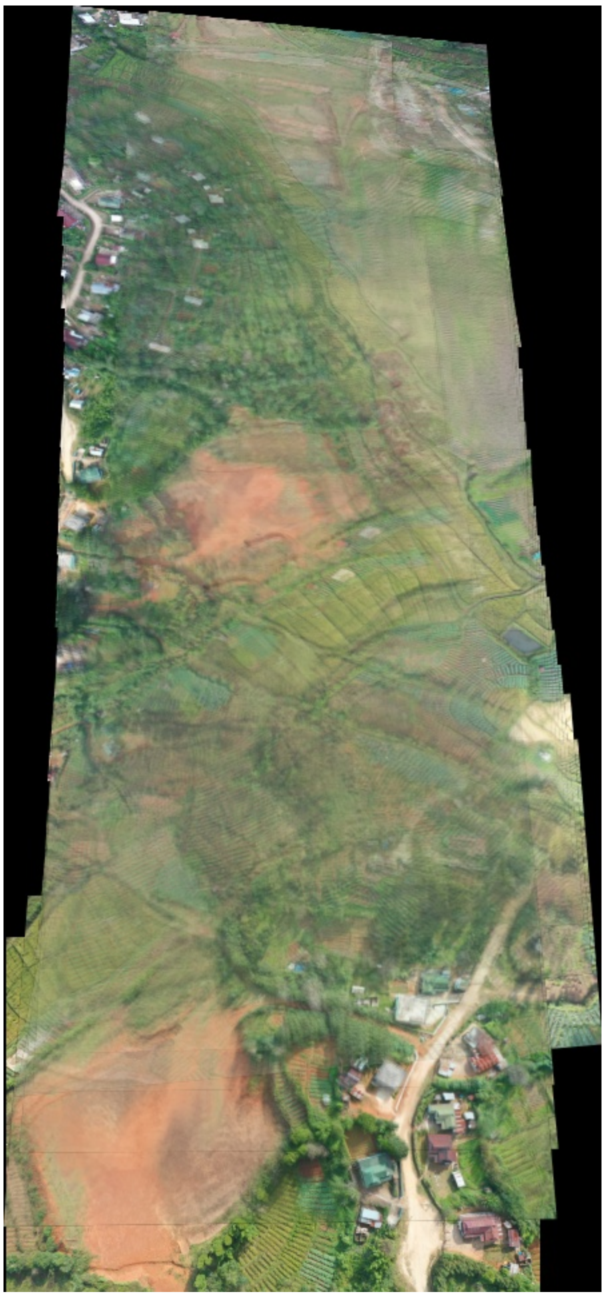

Figure 7.1: Output Panorama Generated from Dataset

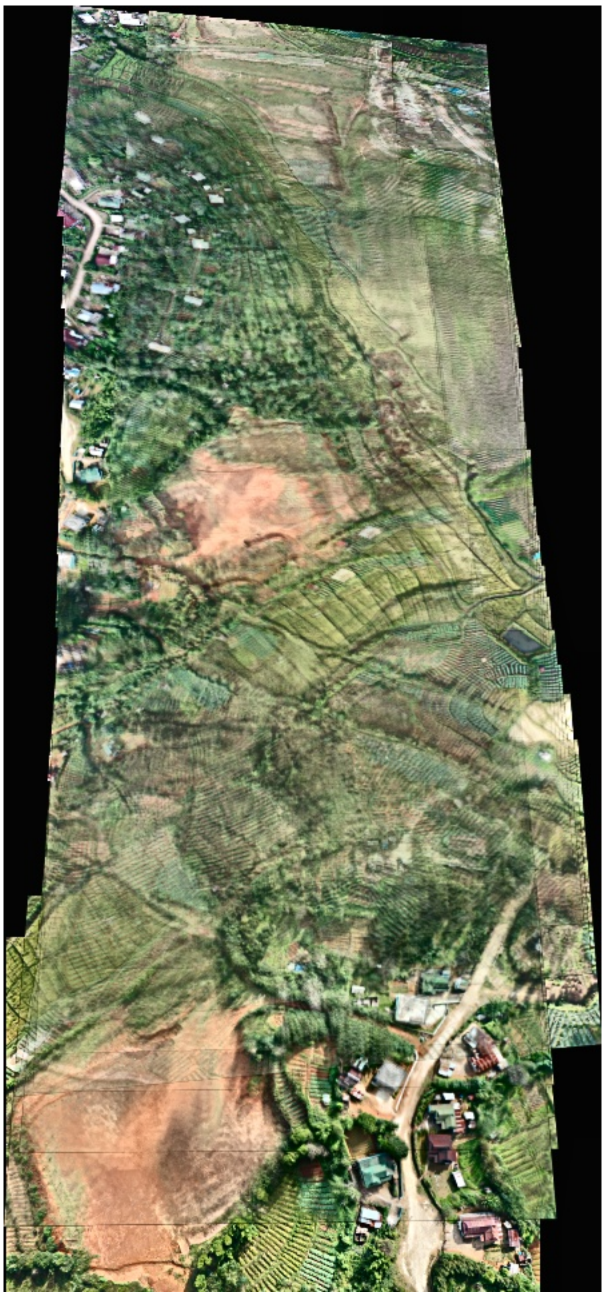

Figure 7.2: Enhanced Post Processed Output

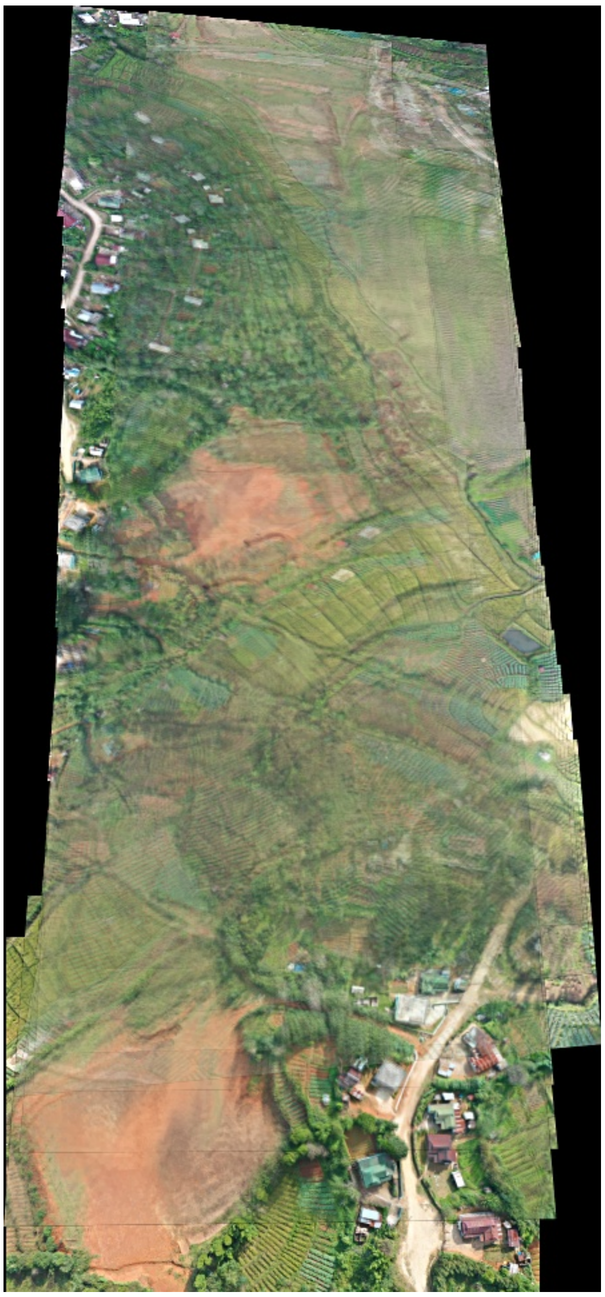

Figure 7.3: Sharpened Post Processed Output

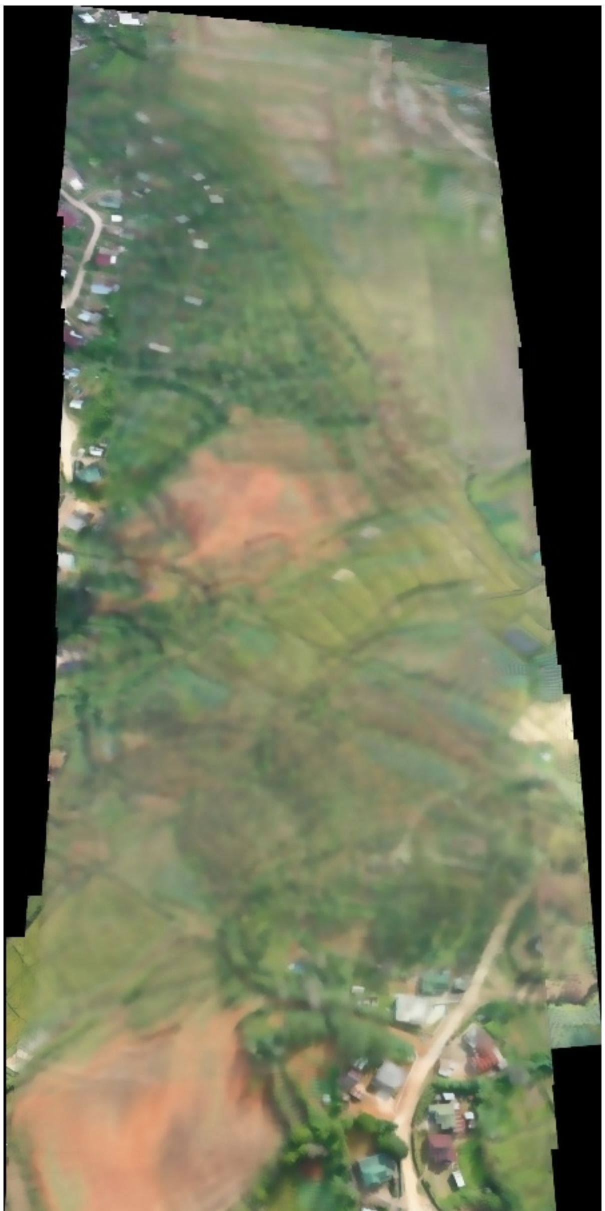

Figure 7.4: Bilateral Post Processed Output

Chapter 8

Challenges Faced and Solutions

The development of a distributed computer vision pipeline presents unique challenges that are not typically encountered in single-machine applications. This chapter outlines the key obstacles faced during the project and the solutions that were engineered to overcome them.

8.1 PySpark Serialization Limitations with OpenCV Data

Challenge: One of the most significant initial hurdles was data serialization. Spark requires data to be transferred between the Python-based driver and the JVM-based executors. Native OpenCV data structures, particularly `NumPy` arrays (used for images) and `cv2.KeyPoint` objects, are not directly serializable by Spark's default mechanisms. Attempting to pass these objects through an RDD often resulted in `Py4JJavaError` and serialization failures.

Solution: A multi-step encoding strategy was implemented. Images were first encoded into a PNG byte stream using `cv2.imencode()` and then further converted into a Base64 ASCII string. For feature points, `cv2.KeyPoint` objects were deconstructed into their fundamental attributes (e.g., `pt`, `size`, `angle`) and stored as simple Python tuples. These primitive data types (`str`, `tuple`, `float`) are universally serializable, ensuring robust and reliable data transmission between Spark stages.

8.2 Managing the Image Resolution vs. Memory Trade-off

Challenge: Using full-resolution images (e.g., 12MP or higher) directly in the pipeline frequently caused executors to run out of memory, leading to job failures. The memory required to load and process dozens of large images simultaneously exceeded the available executor memory.

Solution: An adaptive resizing step was introduced during the image loading phase. Images are resized on the worker nodes to a manageable maximum resolution (e.g., 800$\times$600 pixels) before feature extraction. This substantially reduces the memory footprint of both the image arrays and the corresponding feature descriptors, allowing the pipeline to execute smoothly. The resizing logic is dynamic and becomes more aggressive for larger datasets.

8.3 Inconsistent Feature Density in Drone Images

Challenge: Aerial drone imagery often includes large, homogenous regions (e.g., open fields, water, rooftops) with very low texture, resulting in sparse keypoints. Conversely, areas such as tree lines or urban regions are feature-rich. This inconsistency made it difficult to rely solely on match count, as many matches could be localized in small, irrelevant regions.

Solution: The homography validation logic was enhanced to evaluate not only the number of inliers but also their spatial distribution. A geometric transformation is now accepted only if the inlier points are reasonably well spread across the image. This prevents false positives arising from clusters of local features, thereby increasing the robustness of the stitching process.

8.4 Designing Fallbacks for Poor Matches

Challenge: In a long sequence of images, some pairs may contain insufficient overlap or be affected by motion blur, causing feature matching to fail. A naive linear chaining approach would halt on the first failure, compromising the entire stitching process.

Solution: Two layers of fallback logic were introduced to improve resilience:

- **Skip-Ahead Logic:** If the match between image N and $N + 1$ fails, the system attempts to match image N with $N + 2$, $N + 3$, and so on, up to a predefined window. This allows the pipeline to bypass problematic images and continue building the panoramic chain.
- **Affine Fallback:** When matching is successful but a robust homography cannot be computed (often due to limited parallax), the system attempts to estimate a simpler affine transformation. Although less precise, this model is more tolerant of unstable geometry and often allows the pipeline to continue stitching.

These fallback mechanisms provide a graceful degradation path and enable the pipeline to handle real-world data imperfections more effectively.

Chapter 9

Future Work and Potential Enhancements

While this project successfully demonstrates a scalable and robust pipeline for image stitching, it also lays the groundwork for numerous potential enhancements. The current architecture can be extended and refined to improve accuracy, performance, and usability. This chapter outlines several promising avenues for future work.

9.1 Integrate GPS Metadata for Geo-Referenced Stitching

Motivation: Many modern cameras, particularly those mounted on drones, embed GPS coordinates and other EXIF metadata directly into image files. This spatial information is currently unused but represents a valuable asset.

Proposed Enhancement: The image loading stage can be enhanced to extract GPS latitude, longitude, and altitude information from the EXIF metadata. This spatial data could be used to:

- Perform rough initial placement of images in a global coordinate system.
- Limit feature matching to geographically adjacent images, reducing computation.
- Output a geo-referenced orthomosaic suitable for import into GIS software such as QGIS or ArcGIS.

9.2 Migrate from RDDs to DataFrames and the Spark SQL Engine

Motivation: The current pipeline is implemented using Spark's Resilient Distributed Dataset (RDD) API. While powerful, modern Spark applications increasingly favor the DataFrame API due to its efficiency and expressiveness.

Proposed Enhancement: Refactor the application to use Spark DataFrames instead of RDDs. Benefits include:

- Access to the Catalyst optimizer for query planning.
- Use of the Tungsten execution engine for off-heap memory management.
- Better integration with Python, reducing JVM communication overhead.

This shift can lead to substantial performance improvements, especially for large datasets.

9.3 Implement Global Optimization with Bundle Adjustment

Motivation: The current pipeline uses a sequential homography computation approach. Over time, this can introduce cumulative errors ("drift") leading to misalignment in long panoramas.

Proposed Enhancement: Add a *Bundle Adjustment* stage after pairwise homography computation. This global optimization technique would:

- Refine camera poses and intrinsic parameters.
- Minimize the overall reprojection error across all matches.
- Produce globally consistent stitching results.

This could be implemented on the driver node using non-linear solvers such as `scipy.optimize.l`

9.4 Implement Graph-Based Stitching for Unordered Collections

Motivation: The existing stitching logic assumes a linear sequence of images. This limits the pipeline's effectiveness for unordered or arbitrarily captured image sets.

Proposed Enhancement: Shift to a graph-based image chaining strategy:

- Perform all-pairs matching to construct a fully connected graph.
- Use match quality scores as edge weights.
- Construct a Minimum Spanning Tree (MST) to determine the optimal stitching order.

This approach increases flexibility and enables the system to robustly handle unordered image collections.

9.5 Develop a GUI or Web Dashboard for Usability

Motivation: The current command-line interface is powerful but not user-friendly for non-technical users.

Proposed Enhancement: Build a web-based dashboard using frameworks such as `Streamlit` or `Flask`. Features could include:

- Image upload via browser.
- Parameter tuning using interactive widgets.
- Real-time progress monitoring.
- In-browser viewing and downloading of the final stitched panorama.

A user interface would greatly improve accessibility and broaden the system's user base.

Chapter 10

Conclusion

This project set out to address the critical scalability limitations of traditional, single-machine image stitching software. The core challenge—processing large, high-resolution image datasets, which frequently leads to memory exhaustion and prohibitive execution times—was met by designing and implementing a novel, distributed pipeline built upon the Apache Spark framework.

The resulting system successfully demonstrates the power of distributing computationally intensive tasks. By parallelizing the core bottlenecks of feature extraction and feature matching across a cluster, the pipeline achieves a significant reduction in overall processing time. This enables the creation of large-scale panoramas from datasets that would be intractable for conventional tools.

The architecture proved to be not only scalable but also robust, due to the integration of intelligent fallback mechanisms such as:

- **Skip-ahead chaining**, which allows the pipeline to continue even when adjacent matches fail.
- **Affine transformation fallback**, which provides geometric flexibility when full homographies cannot be computed.

Ultimately, this work serves as a successful proof-of-concept, validating the efficacy of applying general-purpose big data technologies to solve complex computer vision problems. The project met all its stated objectives, delivering a functional, end-to-end system capable of producing high-quality panoramic images efficiently.

It stands as a clear demonstration that the future of large-scale visual data processing lies in the thoughtful integration of specialized computer vision algorithms with powerful distributed computing frameworks like Apache Spark.

Bibliography

[1] Szeliski, R. (2010). *Computer Vision: Algorithms and Applications*. Springer. Retrieved from https://szeliski.org/Book/

[2] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2), 91–110. Retrieved from https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

[3] Rublee, E., Rabaud, V., Konolige, K., & Bradski, G. (2011). ORB: An efficient alternative to SIFT or SURF. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)* (pp. 2564–2571). Retrieved from https://ieeexplore.ieee.org/document/6126544

[4] Alcantarilla, P. F., Nuevo, J., & Bartoli, A. (2013). Fast explicit diffusion for accelerated features in nonlinear scale spaces. In *Proceedings of the British Machine Vision Conference (BMVC)*. Retrieved from https://www.bmva.org/bmvc/2013/Papers/paper0032/paper0032.pdf

[5] Brown, M., & Lowe, D. G. (2007). Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1), 59–73. Retrieved from https://www.cs.ubc.ca/~lowe/papers/ijcv07.pdf

[6] Apache Software Foundation. (2024). *Apache Spark: Unified analytics engine for large-scale data processing*. Retrieved from https://spark.apache.org

[7] Triggs, B., McLauchlan, P. F., Hartley, R. I., & Fitzgibbon, A. W. (2000). Bundle adjustment—A modern synthesis. In *Vision Algorithms: Theory and Practice* (pp. 298–372). Springer. Retrieved from https://link.springer.com/chapter/10.1007/3-540-44480-7_21

[8] Loughran, S. (n.d.). *Hadoop WinUtils Binaries for Windows*. Retrieved from https://github.com/steveloughran/winutils

[9] OpenDroneMap. (2024). *OpenDroneMap: The Open Source Toolkit for Processing Aerial Imagery*. Retrieved from https://www.opendronemap.org

[10] Pix4D. (2024). *Pix4Dmapper Photogrammetry Software*. Retrieved from https://www.pix4d.com/product/pix4dmapper-photogrammetry-software

[11] Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*. Retrieved from https://opencv.org/

[12] Virtanen, P., Gommers, R., Oliphant, T. E., et al. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3), 261–272. Retrieved from `https://doi.org/10.1038/s41592-019-0686-2`

[13] Streamlit Inc. (2024). *Streamlit: The fastest way to build and share data apps*. Retrieved from `https://streamlit.io`