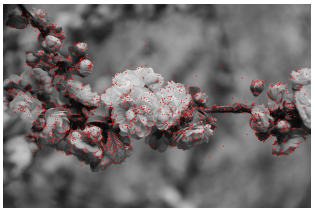
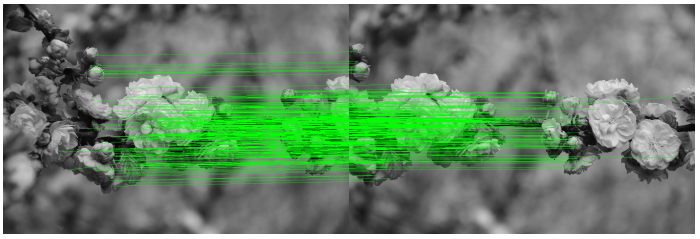
Image Stitching & SIFT Acceleration

B11902044 鄒睿謙　B11902043 劉軒齊　B11902085 顏佐霏

This project focuses on SIFT acceleration and integrating it to OpenCV for image stitching.



Finding features image.



Matching features of 2 images.

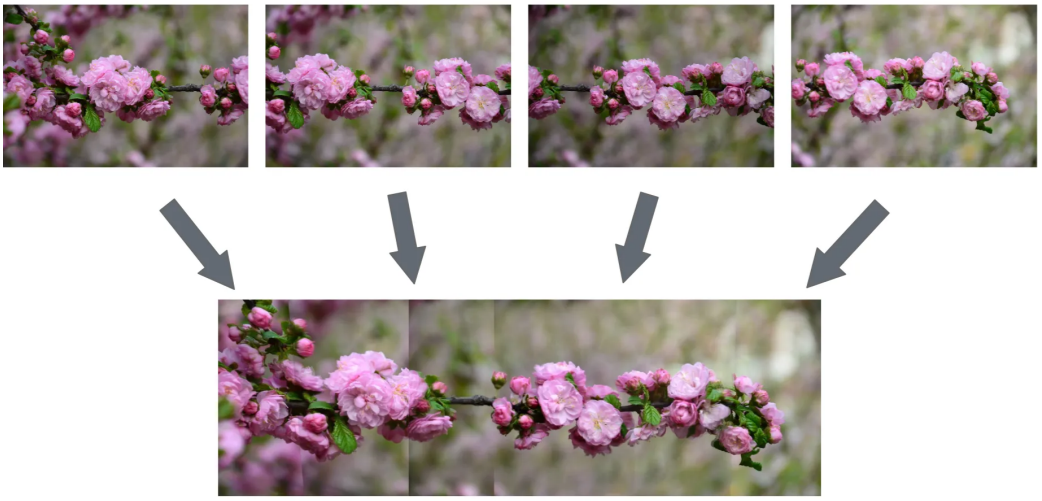


Image stitching results.

Github repo : https://github.com/sofiyen/PP2024-Image-Stiching-Acceleration

Motivation

Image stitching is a critical technique in computer vision that combines multiple overlapping images into a seamless panoramic view. It is widely used in applications such as panoramic photography, virtual reality, and geographic information systems. Besides, image stitching is also widely used in medical imaging for creating high-resolution composites, in satellite imaging for generating detailed maps, and in autonomous navigation systems where robots and vehicles rely on stitched images for situational awareness. There are various algorithm being embedded into image stitching for the purpose of

features extract and mapping. Among the key algorithms powering image stitching is the Scale-Invariant Feature Transform (SIFT), renowned for its ability to detect and describe local features that are robust to scale, rotation, and illumination changes. However, the computational cost of SIFT often makes the image stitching process time-consuming, especially for high-resolution images or large-scale datasets. This problem scales up when it comes to handling real-time or resource-constrained environments.

In this project, we focus on accelerating the image stitching process by parallelizing SIFT through ROCm HIP and OpenMP, and multicore GPUs. By leveraging ROCm HIP for efficient GPU computation and OpenMP for multi-threaded CPU processing, we aim to significantly reduce the execution time of SIFT, thereby shorten the entire image stitching pipeline. This optimization not only enhances the performance of SIFT within OpenCV for keypoints detect, but also enables faster and more efficient panoramic image creation, making it more practical for real-time applications and large-scale deployments.

Introduction : The pipeline of image stitching and SIFT algorithm

Image stitching is a multi-step process in computer vision that combines overlapping images into a seamless panorama. The key stages are:

1. **Keypoint Detection and Description**
    - Identify distinctive features (keypoints) e.g. corners and edges using algorithms such as SIFT or SURF.
    - Generate descriptor vectors for each keypoint, capturing their location, orientation, and context.
2. **Keypoint Matching**
    - Match keypoints between images by comparing descriptor vectors using metrics e.g. Euclidean distance.
    - Validate matches based on reliability to ensure accuracy.
3. **Transformation Estimation**
    - Use matched keypoints to find their homography, a transformation matrix H that represents the necessary tralsation, rotation, scale, and perspective changes in order to fully match the two images.
    - Refined with RANSAC (Random Sample Consensus) to filter out false matches.
4. **Warping and Stitching**
    - Apply the transformation matrix to warp one image onto the other.
    - Combine the warped image and the original image on a blank canvas.
5. **Border Blending and color adjustment**

- Smooth lighting and color differences along the stitching to soften the border and generate a more seamless result.

One of the most time-consuming part of the process is feature extraction, particularly during **keypoint detection and matching**. Algorithms like SIFT require significant computational resources to locate keypoints, compute gradients, and generate descriptor vectors. This stage is critical because the quality of feature extraction directly affects the accuracy of keypoint matching and, consequently, the overall success of image stitching.

Steps of Scale-Invariant Feature Transform (SIFT) Algorithm

1. **Scale-Space Extrema Detection**
    - SIFT identifies keypoints by detecting extrema in a series of Difference-of-Gaussian (DoG) images, which are computed across multiple scales. This ensures that the keypoints are invariant to lighting, scale of image, and rotation transformations.
2. **Keypoint Localization**
    - Detected keypoints are refined to improve stability by discarding low-contrast points and points located along edges, ensuring that only reliable features are retained.
3. **Orientation Assignment**
    - An orientation is assigned to each keypoint based on the local gradient direction. This step ensures that the keypoints are invariant to rotation.
4. **Keypoint Descriptor Generation**
    - For each keypoint, a descriptor is created by sampling the gradient magnitudes and orientations within a local region. These descriptors are designed to be robust to changes in illumination and minor geometric distortions.
5. **Keypoints matching for image stitching**
    - Feature matching involves finding correspondences between keypoints in two images. First, matches are filtered using the ratio test, where a match is kept if the distance ratio between the nearest and second nearest neighbor is below a threshold θ. Then, incorrect matches are further removed using RANSAC, which estimates a geometric transformation (e.g., homography) and discards outliers to retain only the most reliable matches.

Acceleration Methods

We focusing on accelerating different parts of SIFT algorithm. From this initial profiling result of sequential SIFT code :

> *// Finding images profiling*
>
> ***Gaussian Pyramid: 647 ms***
> *DoG Pyramid: 20 ms*
> *Keypoint Detection: 64 ms*
>
> ***Gradient Pyramid: 463 ms***
>
> ***Orientation & Descriptor Computation: 541 ms***
>
> *Finding keypoint time: 1739 ms*
> *Saving image time: 60 ms*
> *Total execution time: 1818 ms*

We see that the parts Gaussian Pyramid generation, Gradient Pyramid generation, and Orientation & Descriptor Computation parts should be the focus of acceleration.

1. ROCm HIP

- **Gaussian blur**

  We found out the bottleneck of Gaussian pyramid generation is the Gaussian blur process, and accelerated the computation by implementing a separable 2D convolution on the GPU using HIP. Instead of performing a full 2D convolution, we decomposed it into two 1D convolutions - vertical and horizontal - which significantly reduces the computational complexity from $O(n^2)$ to $O(2n)$ for an $n \times n$ kernel.

  Our implementation leverages several GPU optimization techniques:

  - Shared memory usage to cache the convolution kernel, reducing global memory access latency.
  - Thread block size of 16×16 for efficient GPU occupancy.
  - Coalesced memory access patterns in both vertical and horizontal convolutions.
  - Parallel processing where each thread handles one output pixel.

  We first apply vertical convolution, stores intermediate results, and then performs horizontal convolution to produce the final blurred image. All memory transfers between CPU and GPU are handled efficiently using `hipMemcpy()`, and we ensure proper synchronization using `hipDeviceSynchronize()`. This parallel implementation provides substantial speedup compared to a sequential CPU-based approach.

- **Gradient pyramid generation** :

We accelerated the gradient pyramid generation through several GPU optimization strategies using HIP:

1. Shared Memory Tiling: We implemented a tiled approach using shared memory in the `compute_gradients_shared` kernel. Each block loads a tile of size `BLOCKSIZE + 2` to include halo regions, significantly reducing global memory accesses.
2. Stream Processing: We utilized HIP streams to enable concurrent execution across different octaves of the pyramid. Each octave runs in its own stream (`streams[i]`), allowing overlap of memory transfers and kernel execution between octaves.
3. Memory Access Pattern: We optimized memory access by carefully loading the central pixels and halo regions into shared memory tiles, ensuring coalesced memory access patterns where possible.
4. Asynchronous Operations: We leveraged asynchronous memory transfers (`hipMemcpyAsync()`) for both host-to-device and device-to-host transfers, overlapping data movement with computation.
5. Thread Block Organization: Using a block size of 32×32 threads, we balanced occupancy and shared memory usage while handling edge cases properly through boundary checks.

This GPU implementation significantly outperforms the CPU version by processing multiple pixels in parallel and efficiently managing memory hierarchy through shared memory usage and asynchronous operations.

- **Image saving of matched features**

  We accelerated the image saving process by parallelizing the data transformation required before saving to JPG format. Here's how we optimized it:

  1. Memory Layout Transformation: We implemented a GPU kernel `transformImageDataKernel()` that handles two critical operations in parallel:
     - Converting from floating-point (0.0-1.0) to byte format (0-255)
     - Reorganizing data from channel-major to pixel-major format for JPG encoding
  2. Thread Organization: We used a 2D thread block structure (16×16 threads) to map directly to the 2D image structure, where each thread processes one pixel location across all channels. This ensures efficient memory access patterns.
  3. Memory Operations:
     - Used `__restrict__` keyword to hint at non-overlapping memory access
     - Implemented efficient memory layout transformation by computing source and destination indices for each channel
     - Used atomic operations implicitly through the channel loop to avoid write conflicts

4. Data Transfer Pipeline:
    - Allocated device memory for both float input and byte output
    - Transferred input data to GPU
    - Processed data in parallel on GPU
    - Transferred transformed data back to CPU for final JPG encoding using `stbi`

    This GPU implementation significantly speeds up the image saving process by parallelizing the data transformation step, which is particularly beneficial for large images with multiple channels.

2. OpenMP

We also implement OpenMP parallelizing techniques on tasks that are less suitable or too complex for GPU implementation, including:

- **Parallelized Loops for Keypoint Processing**
    - We introduced `#pragma omp parallel for` to distribute the iteration of keypoints across multiple threads. Each thread calculates the orientation of a subset of keypoints independently.
    - To ensure safe updates to shared resources (e.g., appending results to a global keypoint list), we added a **critical section** (`#pragma omp critical`), preventing race conditions during concurrent writes.
- **Parallelization in Feature Matching**
    - We utilized `#pragma omp parallel` to split the matching process across threads, allowing each thread to handle a subset of keypoints from the first image.
    - To minimize contention, each thread stores its results in a local data structure. At the end of the computation, we added a `#pragma omp critical` section to safely merge these local results into the global match list.

3. Multi-core GPU utilization by pthread

We further utilized pthreads to enable multi-core GPU usage by asynchronously processing multiple images on different GPUs. Below are the enhancements we added to achieve this:

1. **Thread Creation for Parallel Processing**
    - We created two separate threads using `pthread_create()` and assigned each thread to process a different image. Each thread receives a unique GPU ID, ensuring that the computations for the two images are executed on separate GPUs simultaneously.
2. **Thread Data Struct for GPU Assignment**
    - We defined a custom `ThreadData` structure to encapsulate the data required by each thread, including a pointer to the image, the keypoints list

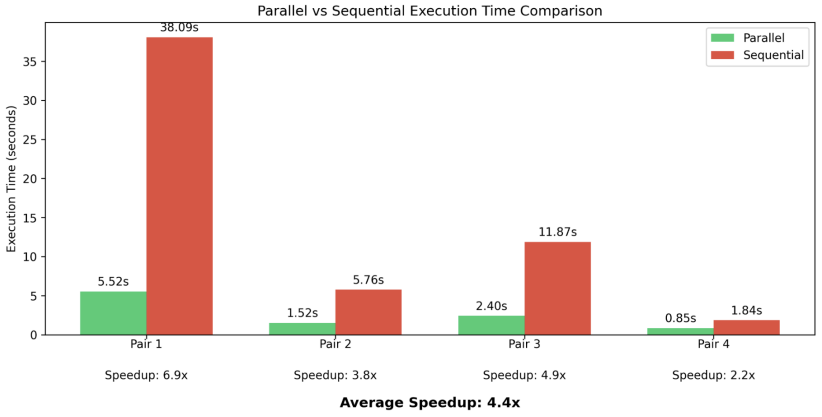
for storing results, and the GPU ID to be used for computation. This ensures that each thread operates on isolated data while leveraging specific GPU resources.

3. **Thread Synchronization**
    - To ensure the threads complete their tasks before proceeding, we incorporated `pthread_join()` to synchronize the threads. This guarantees that the keypoints for both images are fully computed before continuing with subsequent operations.

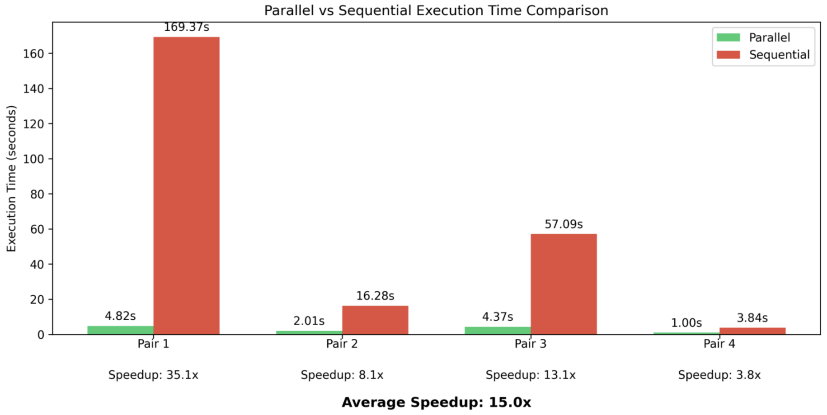
This implementation effectively distributes the workload across multiple GPUs by leveraging pthreads, enabling concurrent processing of images and achieving significant speedups in feature extraction tasks.

Evaluation

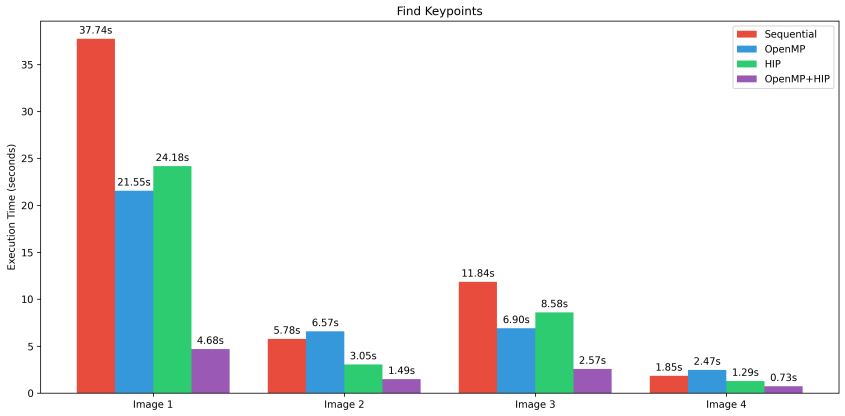
- **Sequential vs. Parallel:**
    - Find keypoints



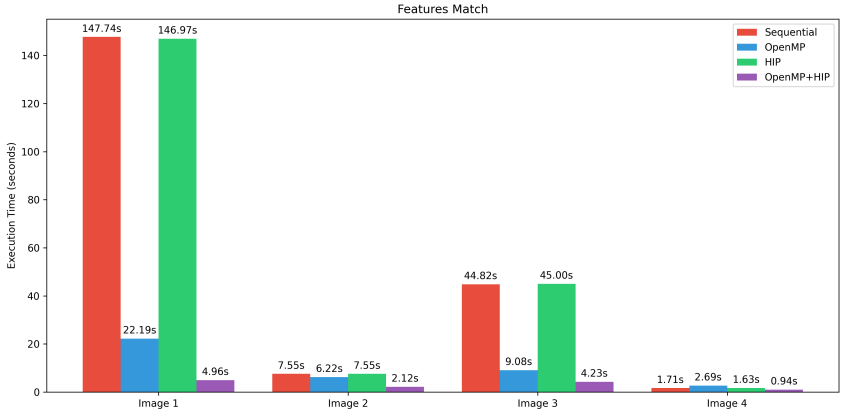
    - Match features



- **Sequential vs. OMP vs. HIP vs. Parallel:**
    - Find keypoints



- Match features



Analysis

- We profile the execution time of fully sequential vs. parallelized code (OpenMP + HIP) and found that it achieves up to 15x speedup while executing match features, indicating our success in parallelizing tasks on both GPU and CPU level.
- We further analyze the performance of the program that only use OpenMP or HIP independently, which means they do not fully utilize the parallelization resources. We can see that their performance is better than the sequential version, but are still slower than our hybrid-version. In particular, in match features task, if we do not implement OpenMP for parallelizing the process of finding keypoints pairs, the complexity will become $O(n^2)$, leading to the poor performance.

Conclusion

We have successfully reached 15 times more speedup through parallelization efforts. SIFT is a strongly parallelizable algorithm,. Although we have yet to fully discover its full speedup limits, it is notable from the evaluation results that further speedup on multiple parts with GPU acceleration is a direction that we can try in the future.

Reference

- SIFT paper: https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf
- SIFT repo: GitHub - dbarac/sift-cpp: A C++ SIFT implementation (Scale invariant feature transform)
- Image stitching : GitHub - ziqiguo/CS205-ImageStitching: Real Time Image Stitching
- Datasets : As-Projective-As-Possible Image Stitching with Moving DLT
- SURF : GitHub - abhinavgupta/SURF