Accelerating Feature Extraction and Image Stitching Algorithm Using Nvidia CUDA

***A project presented to National Science Foundation for Undergraduate Research**

John Korah

Department of Computer Science
California Polytechnic State University
Pomona, CA, USA
jkorah@cpp.edu

Rick Ramirez

Department of Computer Science
California Polytechnic State University
Pomona, CA, USA
rickramirez@cpp.edu

Du D. Le

Department of Mathematics and Computer Science
Mount San Antonio College
Walnut, CA, USA
dlee312@student.mtsac.edu

Yuqi Chen

Department of Mathematics and Computer Science
Mount San Antonio College
Walnut, CA, USA
ychen453@student.mtsac.edu

***Abstract*— This report introduces an improvement to the feature detection and image stitching algorithm by utilizing parallel computing on the Central Processing Unit (CPU) through MPI, and accelerating the process further using Nvidia CUDA on the GPU. In this report, we present a method for computing image features on GpuMat objects, which can serve as an alternative to the traditional detectAndCompute method in the standard OpenCV library. Our approach makes use of the GPU in four key steps: feature detection, feature matching, image warping, and blending/compositing. The resulting images maintain the same quality as those produced by CPU-based methods, but with significantly improved processing speed, decreasing runtime and cost, enabling the potential for real-time mapping applications.**

***Index Terms*— ORB, SIFT, CUDA, OpenMP, MPI, real-time, node**

I. Introduction

Accelerating image-stitching processes has long been a significant problem in the field of computer vision. Real-time image processing is crucial in numerous applications, particularly during natural disasters where rapid changes in the landscape, collapsing buildings, and outdated maps necessitate immediate and accurate updates. By enhancing these algorithms to perform in real-time, we can significantly improve responsiveness in such critical situations.

Image stitching requires feature detection algorithms. The first widely-used feature detection algorithm was the Harris Corner Detector [1], introduced in 1988 by Chris Harris and Mike Stephens. Later on, in 1999, the Scale-invariant feature transform (SIFT) algorithm [2] by David G. and Lowe revolutionized the field of computer vision by providing a reliable method for feature detection and description. It builds upon the Harris Corner Detector but adds many key innovations to improve robustness and functionality. The SIFT algorithm is still widely used today, and most new methods in feature detection, either through traditional means, or neural networks, use SIFT as their benchmark. Although SIFT is a reliable method for such tasks, it is also very computationally expensive. Therefore, many more methods throughout the years have been developed to address that issue, and in 2011, Oriented FAST and Rotated BRIEF [3] (ORB) was introduced. After multiple benchmark tests, ORB has been proven to have better overall performance with quick computing and is demonstrated to be robust to light and rotational shifts.

This paper explores the acceleration of image stitching using the ORB algorithm in conjunction with GPU (Graphics Processing Unit) technology, demonstrating substantial improvements in processing speed and efficiency. This project serves as a successor to our mentor's previous project [4] regarding image stitching via CPU and parallel computing. The project mainly focuses on images captured by Unmanned Aerial Vehicles (UAV), the image patch is then sent to a central computer with an Nvidia graphic card for image processing. The map generation process involves processing a pair of images, then the algorithm will detect, and match each of the key points from one image to another. The author of the original paper, our mentor also designed an algorithm to process the image patch using multi-core CPUs, which are available on most modern computers. Our mentor also addresses the issue of increasing data overhead by employing distributed and shared memory architectures to reduce computation time. Our project will use all of the techniques available above, alongside the GPU, and the goal is to generate reliable maps and accelerate image stitching algorithms even further by utilizing multi-core, multi-threads, and GPU computation.

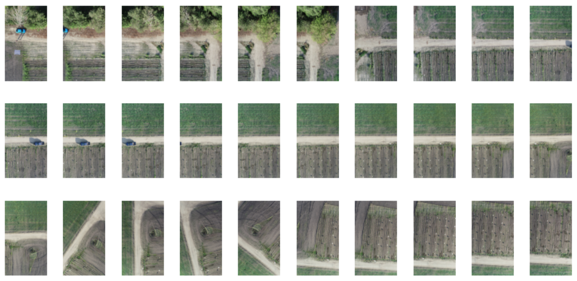


Fig. 1. Patch of 30 images captured by UAV

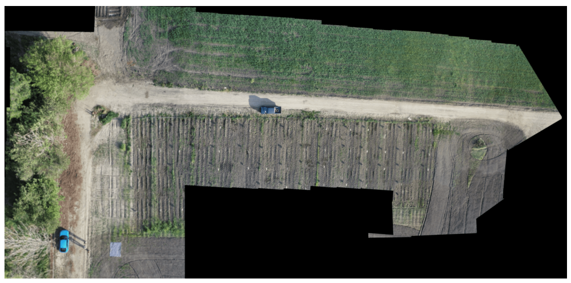


Fig. 2. Final Map Obtain from Image Stitching

For this project, we will be using the Open Source Computer Vision Library (OpenCV), and to maximize the performance of our project, we will use a low-level programming language, namely C++. OpenCV, in addition to its standard library, provides an extension called the OpenCV Extra Module. This extension enables users to harness the power of GPUs through the NVIDIA toolkit and deep neural networks (DNN), significantly enhancing processing capabilities. For benchmarking purposes, we will use a patch of sixty-four 4k images (2160×3840 px), and the final goal is to accelerate the run time, increase the speedup, and decrease the resource cost as much as possible.

II. Literature Review

A. Classical Feature Detection Methods

1) ORB (Oriented FAST and Rotated BRIEF): ORB [3] is an efficient and low-cost image feature-matching algorithm that combines and optimizes FAST and BRIEF algorithms. This algorithm not only uses FAST's unique way of finding corner points to find key points but also uses the centroid method and Gaussian pyramid to improve the robustness of the algorithm.

FAST, short for "Features from Accelerated Segment Test," is a widely recognized corner detection algorithm introduced by Edward Rosten and Tom Drumm in their paper "Machine Learning for High-speed Corner Detection." [5] FAST identifies corners by examining a circular region around a pixel to determine if the pixel qualifies as a corner. The ORB (Oriented FAST and Rotated BRIEF) algorithm incorporates FAST-9, which uses a circle with a radius of 9 pixels to optimize performance. However, because the original FAST algorithm does not account for orientation, ORB enhances it by assigning an orientation to each detected keypoint, thereby making the algorithm rotation-invariant. This orientation is calculated using the intensity centroid method, which leverages the moments of the patch around the keypoint, defined as:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \tag{1}$$

These moments are used to determine the dominant orientation of the keypoint. The centroid can be then computed by

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \tag{2}$$

By optimizing the BRIEF (Binary Robust Independent Elementary Features) algorithm, binary descriptors are generated by simple pixel intensity comparison, and Angle invariance is recognized by shifting the descriptors in the main direction. The ORB algorithm is mainly used because of its accuracy of feature detection and low equipment requirements, which will make it suitable for image capture using UAVs and other low-cost devices in the future.

2) SIFT and PopSIFT: a Faithful SIFT Implementation For Real-time Applications: SIFT [2] is one of the most well-known and reliable feature detectors in the field of computer vision to extract invariant features from an image. SIFT primarily works based on the Difference of Gaussian (D.o.G), defined as

$$D(\mathbf{x},\sigma) = \frac{1}{2\pi\sigma} \left[\frac{1}{k} \exp\left(-\frac{\|\mathbf{x}\|^2}{2(k\sigma)^2} \right) - \exp\left(-\frac{\|\mathbf{x}\|^2}{2\sigma^2} \right) \right]. \tag{3}$$

Since

$$D(\mathbf{0},\sigma) = \frac{1}{2\pi k\sigma} < \frac{1}{\sqrt{2}} D_{max} \quad \text{and} \quad D(\infty,\sigma) = 0, \tag{4}$$

the D.o.G is a bandpass filter, which means that it only lets selected frequencies through. One characteristic of the band-pass filter is that it lets frequencies greater than $\frac{1}{\sqrt{2}} D_{max}$ through and blocks low frequencies. This means that the frequencies that it passes tend to be the areas of high contrast on the image, which are usually edges (In computer vision, edges are abrupt changes in intensity, discontinuity in image brightness, or contrast).

The algorithm then uses the Gaussian pyramid to compare the respective pixel with its surrounding 26 other neighbor pixels to look for the extrema of the image and improve the accuracy by the Taylor series method. SIFT then uses a straightforward approach for assigning orientations. For each image sample, $L(x,y)$, we compute the gradient magnitude $M(x,y)$ and orientation angle $\theta(x,y)$ using differences of pixels:

$$\begin{split} M(x,y) = [(L(x+1,y) - L(x-1,y))^2 + \\ (L(x,y+1) - L(x,y-1))^2]^{1/2} \end{split} \tag{5}$$

and,

$$\theta(x,y) = \tan^{-1}\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)} \right). \tag{6}$$

The final output is an n-dimensional feature vector whose elements are invariant feature descriptors. SIFT, although a reliable method of extracting image features, is also a very computationally costly method, and therefore may not be suitable for real-time applications. Authors Griwodz, Calvet, and Halvorsen proposed a Python library called PopSIFT [6] in 2018, which serves as a CUDA implementation of the SIFT algorithm. The method involves utilizing the power of CUDA by uploading images to the GPU, which greatly reduces runtime for real-time image processing.

B. Overview of General GPU Architecture and Nvidia's Compute Unified Device Architecture (CUDA)

1) General GPU Architecture: Unlike a Central Processing Unit (CPU), which typically has only a few physical cores, a Graphics Processing Unit (GPU) can contain thousands or even tens of thousands of cores. Although GPU cores are designed for simple calculations and are less versatile than CPU cores, the GPU's key advantage lies in its ability to perform large-scale parallel computations. Take a look at an image, for instance:

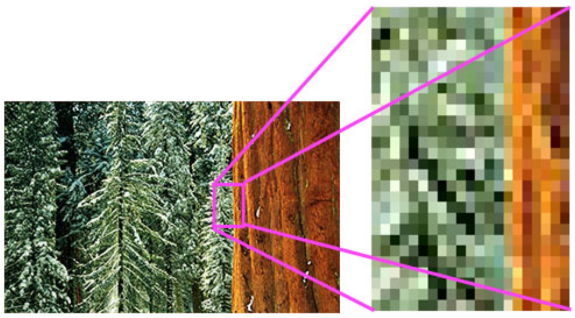


Fig. 3. Pixel Zooming, Image by Julie Waterhouse Photography

Images are composed of millions of pixels. When processing an image, the GPU cannot handle pixels one by one (it shouldn't!), as doing so would leave many cores idle, leading to a significant waste of resources. Instead, a key principle of GPU operation is parallelism: whichever core is available first takes on the computation, ensuring that all cores are utilized efficiently. All computations are independent of each other, and finally, we only need to combine them to output the final input image.

Let's say we have 100 tasks for a GPU to compute, with each task represented as a thread. GPUs are designed to handle thousands or even millions of threads simultaneously, but they don't assign one thread to each core directly. Instead, GPU cores are organized into groups called "warps," typically consisting of 32 threads that execute the same instruction simultaneously. These warps are managed by units within the GPU called Streaming Multiprocessors (SMs).

If the number of tasks increases to 1 million, the GPU doesn't need 1 million cores to process them. Instead, it groups the threads into blocks, and these blocks are distributed among the SMs. Each SM contains multiple cores that work together to execute the warps of threads. When an SM processes a block, it schedules the warps dynamically, keeping all its cores busy and switching between warps to hide any delays, such as waiting for memory access.

For example, with 1,000 cores organized into several SMs, each SM might handle many blocks of threads. Even if a single block contains 1,000 threads, the SM can manage them efficiently by processing them in warps and switching between them as needed. This hierarchical structure allows the GPU to handle a vast number of threads, ensuring that even with millions of tasks, the cores are utilized efficiently to maximize parallel computation.

2) Compute Unified Device Architecture (CUDA): CUDA is a software developed by Nvidia Corporation used to accelerate parallel computing [7]. In simple terms, when given a task, the CPU sends the instruction to the GPU, the GPU will then process the task, and send it back to the CPU. However, this mere process alone could result in unwanted latency.

Instead of the CPU merely sending instructions and waiting, it transfers data to the GPU for processing. The GPU, utilizing CUDA, processes this data in parallel across multiple GPU devices. Once processing is finished, the data is returned to the CPU for use by the application. This parallel processing is fundamental to understanding CUDA software. The advantage of this software is the crucial reason why we must use Nvidia GPUs, but not of any other brands.

III. Methodology

A. Introduction

The code base for this experiment comes from our mentor's previous project. In his experiments, ORB's OpenCV 4.5.5 was used to implement feature detection and image stitching on high-resolution images captured by UAVs, thereby efficiently generating real-time panoramas. There is a section on ORB acceleration in OpenCV, which will be discussed in detail later.

The purpose of this experiment is to use OpenCV's ORB algorithm and CUDA acceleration algorithms on the basis of our mentor's research to improve the efficiency of image processing further. We selected ORB due to its effectiveness as a feature extraction algorithm available with CUDA in the extended OpenCV library, which is compatible with C++. In contrast, alternative methods discussed in the literature review, such as PopSIFT and neural-network-based techniques, are limited to Python. While Python offers ease of use, it generally exhibits lower performance compared to C++.

B. *Image Stitching process*

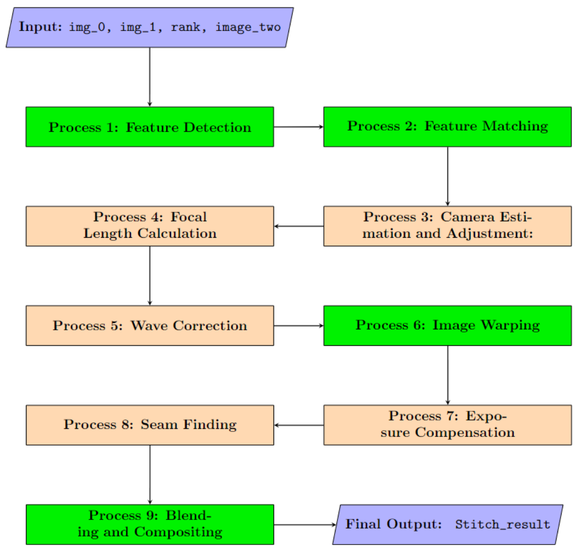

Input: img_0, img_1, rank, image_two
Process 1: Feature Detection
Process 2: Feature Matching
Process 4: Focal Length Calculation
Process 3: Camera Estimation and Adjustment:
Process 5: Wave Correction
Process 6: Image Warping
Process 8: Seam Finding
Process 7: Exposure Compensation
Process 9: Blending and Compositing
Final Output: Stitch_result

Fig. 4. Image Stitching Workflow, where green are steps that use CUDA

1) Basic Workflow: The diagram above shows the algorithm workflow and the green boxes are the steps that utilize CUDA. The steps of image processing are mainly divided into four parts: feature detection, feature matching, image distortion, and image synthesis. The feature detection is mainly implemented by ORB (Oriented FAST and Rotated BRIEF) algorithm, which has high stability and efficiency. First, the feature points of two images with spatial order are detected, and then the feature points with high similarity in the two images are matched by a matcher. During the matching process, there may be some undesirable feature pairs (such as outliers that are too high or matching lines that are wrong), while a good match should be parallel and nearly horizontal. The Random Sample Consensus (RANSAC) algorithm can be used to filter out these undesirable feature pairs, and only the parallel horizontal matching lines are retained.

Image distortion and compositing is the next step of the image process, the main purpose is to align and fuse two images. Although the exact implementation of these steps is not explained in detail here, their importance cannot be overlooked. Image distortion usually involves geometric transformations to correct for perspective differences between images, while image synthesis involves color adjustment and gap filling to ensure a seamless connection of the final stitched image. Methods to improve image processing speed In order to further improve the speed of image processing, our mentor used a distributed memory design to parallelize image processing. Distributed memory designs transfer information between multiple CPU cores through MPI (Message Passing Interface), with each or more CPU cores processing two images as a node. The greater the number of CPU cores, the greater the number of images that can be processed in parallel simultaneously, without the need to wait for earlier images in the list to complete processing before processing later images. For example, if one wishes to concatenate four images, one can use two CPU cores for parallel processing. When the two nodes are finished processing, the results are sent to the new node via MPI to process the two images processed by the previous node. This structure is similar to the tree structure and has been shown to increase the computation speed. However, one needs to note that given an unlimited amount of resources, it does not necessarily mean that the program can speed up infinitely, see Amdahl's law [8], which states that the potential speedup of a process due to parallelization is limited by the portion of the process that cannot be parallelized, which can be formalized by the equation

$$S = \frac{1}{(1-P)+\frac{P}{N}}, \tag{7}$$

where S is the speedup ratio, P is the proportion of the task that can be parallelized, N is the number of processors or cores, and $(1-P)$ is the proportion of the task that must be performed sequentially.

Another method to improve the efficiency of the algorithm is shared memory design, which is implemented through Open Multi-Processing (OpenMP). Shared memory designs also support parallel processing, allowing for multiple CPU cores per node to process images in distributed memory designs. Each image is divided into sub-sections of varying sizes according to the rules, and each CPU core acts as a thread to process the sub-sections of the image. This multithreaded image processing method has also proved to be effective in improving the speed of computation.

2) Optimization: The main change from the original paper is to convert the method of image feature extraction using CPU to GPU. We will add a function based on our mentor's original code that can detect image features using the GPU. We changed the finder of feature detection to use `cuda::ORB`, and based on the open source code of OpenCV CUDA, we used the corresponding CUDA ORB image feature detection algorithm to achieve the same purpose as its regular version. We call this new function

```
Mat image_stitch_cuda(Mat img_0, Mat
img_1, int rank, string image_two)
```

The algorithm is made more flexible by adding a function that uses CUDA to accelerate image stitching. Users can adjust the way the algorithm works according to the device they are using; Use CPU for image stitching, or GPU to speed up image stitching. Aside from the main stitching function, we also added another function called

```
void computeImageFeatures(Ptr<cuda::ORB>
orb, const cuda::GpuMat& img,
ImageFeatures& features)
```

to compute GPU images. This is an analogous version of the `detectAndCompute` function when using regular feature detector ORB.

IV. Results and Analysis

A. *Data collection and analysis*

We will use 64 4K images captured by a UAV (Unmanned Aerial Vehicle). The primary goal of this experiment is to utilize CUDA to accelerate the processing of high-resolution images, so we will use images with a resolution of 3840 × 2160 px.

CPU Implementation. We will evaluate the algorithm's performance under different conditions, including with and without CUDA acceleration. In the absence of CUDA acceleration, we will record the impact of distributed memory design (utilizing MPI) and shared memory design (utilizing OpenMP) on the algorithm's speed. Given our limited resources, we will test the processing speed using 1, 2, 4, 8, 16, and 32 CPU cores, as well as using 1, 2, 4, 8, 16, 24 threads, and 32 threads.

GPU Implementation. For the CUDA-accelerated version of the algorithm, the tests will concentrate exclusively on the distributed memory design. This focus is due to the current limitations of the experiment, which prevent us from utilizing the GPU implementation across multiple threads.

a) Detailed Experimental Steps::

- **Performance Testing without CUDA Acceleration**:
  - **Benchmark Test**: Record the processing speed of 64 images without any optimization, which will serve as the benchmark for the experiment.
  - **Single-Core Execution (Serial Processing)**: Test the processing speed of 64 images using a single CPU core.
  - **Distributed Memory Design (Parallel Processing)**: Test the processing speed using 1, 2, 4, 8, 16, and 32 CPU cores.
  - **Single-Core Execution and Shared Memory Design (Serial Processing)**: Test the processing speed using 1, 2, 4, 8, 16, 24, and 32 threads.
  - **Distributed Memory and Shared Memory Design (Parallel Processing)**: Test the processing speed using combinations such as 2 cores 2 threads, 2 cores 4 threads, 4 cores 2 threads, 4 cores 4 threads, etc.
- **Performance Testing with CUDA Acceleration**:
  - **Benchmark test**: The processing speed of 64 images under the optimized design of 1core and 1 threads with cuda was recorded as the second part experimental benchmark.
  - **Single-Core Execution (Serial Processing)**: Test the processing speed of 64 images using a single CPU core and GPU acceleration.
  - **Distributed Memory Design (Parallel Processing)**: Test the processing speed using 1, 2, 4, 8, 16, and 24 CPU cores with GPU acceleration.

Through these tests, we will analyze the impact of various conditions on algorithm performance, especially the effects of CUDA acceleration, distributed memory design, and shared memory design under different numbers of CPU cores and threads. The results will provide valuable insights into optimizing the algorithm's performance.

1) Experimental setup : The experiment will be conducted on two different setups:

	First Setup	**Second Setup**
CPU	Intel Core i7-8750H	Intel Core i9-13900KF
GPU	NVidia GTX 1070 Max-Q	NVidia RTX 4060Ti
OpenCV Ver.	OpenCV 4.10.0	OpenCV 4.10.0
RAM	16GB	32GB
GPU Memory	8GB	16GB

TABLE I
SETUP OF DON (LEFT) AND YUQI (RIGHT)

The CPU and GPU used in the two experimental settings will be different, so there will be some differences in the data obtained. The CPU used in lab setup 1 has only 6 cores, so the data obtained using lab Setup 1 will only have data using 1, 2, 4, and at most 8 cores. We also use different running memory to compare whether the calculation results of the algorithm will be affected under different running memory.

2) Performance Metrics: Analytical Modeling of Parallel System: The number of processors is not directly proportional to the decrease in runtime but depends on which part of the program can be parallelized. As mentioned above, this is formulated by Amdahl's law [8],

$$S = \frac{1}{(1-P) + \frac{P}{N}}, \tag{8}$$

where S is the speedup ratio, P is the proportion of the task that can be parallelized, N is the number of processors or cores, and $(1-P)$ is the proportion of the task that must be performed sequentially.

Definition 1 (Serial and Parallel Runtime): The serial and parallel runtime of a program are defined as follows:

1) The Serial runtime of a program is the time between the beginning and the end of its execution on a sequential processor. We denote this by T_S.
2) Parallel runtime is the time from the moment the first processor begins its execution to the moment the last processor ends its execution. We denote this by T_P.

Definition 2 (Speedup): Speedup is defined as the ratio between the serial runtime to the time taken by parallel runtime,

$$S = \frac{T_S}{T_P} \tag{9}$$

The speedup of a parallel algorithm measures how much faster an algorithm is than its sequential counterpart.

Definition 3 (Cost): The cost of processing a program on a parallel system is defined as the product of run time and the number of processors,

$$C = T_P \cdot p, \tag{10}$$

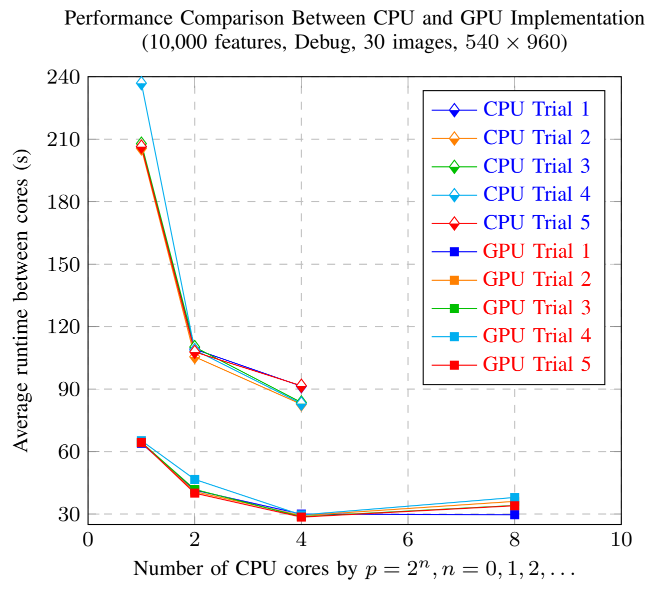
where p denotes the number of processors.

Definition 4 (Efficiency): The efficiency of a parallel program that uses p processors is defined by

$$E = \frac{T_S}{p \cdot T_P} = \frac{S}{p} \tag{11}$$

B. Runtime Comparison

1) CPU vs GPU, no OpenMP: First Setup



CPU Cores	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
1 Cores	63.98	64.32	64.45	65.23	64.43
2 Cores	41.48	40.73	41.82	46.64	40.04
4 Cores	29.97	29.26	28.73	29.59	28.53
8 Cores	29.68	36.08	34.12	37.96	33.99

TABLE II
GPU RUNTIME FOR FIRST SETUP

CPU Cores	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
1 Cores	206.30	205.59	207.84	236.89	206.44
2 Cores	109.16	105.50	110.18	108.78	107.85
4 Cores	91.39	82.71	83.57	83.04	91.60
8 Cores	Unavailable	Unavailable	Unavailable	Unavailable	Unavailable

TABLE III
CPU RUNTIME FOR FIRST SETUP

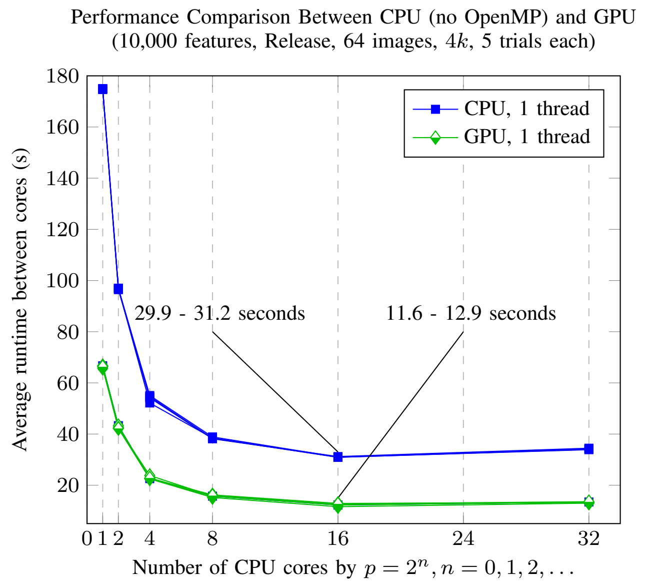
We initially ran our code on Setup 1. The original goal of this research is to be able to stitch 64 4K images, but due to the limited resources of the first setup, we had to downscale the image by 75% of their original quality to 540×960 px.

We observed that our implementation significantly outperformed the original CPU-based image stitching, reducing the runtime by a factor of three to four. However, as the number of used cores exceeded the available physical cores, the application began to slow down. After further investigation, we found that all components of the system were operating at their limits: the CPU was running at 100%, the RAM was fully utilized at 16GB, and the GPU memory was maxed out at 8GB. This likely explains the observed slowdown.

One interesting phenomenon we observed is that the GPU implementation was able to run up to 8 CPU cores, but the original implementation could only run up to 4 cores. One possible reason for this is that the GPU shared the resources with the CPU while performing the algorithm, which put less constraint on the CPU, and hence pushed the number of cores to the limit.

2) CPU vs GPU, no OpenMP: Second Setup.

Due to hardware limitations in the first setup, we switched to a second setup but encountered a similar issue with running out of memory, causing the process to fail after processing only a few 4K images. To address this, we upgraded the RAM in the second setup from 16GB to 32GB. After the upgrade, the application successfully stitched 64 4K images smoothly while maintaining the significant speedup provided by the CUDA framework. To further enhance performance, we compiled the application in release mode. To ensure the reliability of our results, we ran the application five times for each implementation, consistently observing a substantial reduction in runtime for the GPU implementation.



CPU Cores	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
1 Cores	177.44	174.90	175.02	174.80	174.63
2 Cores	96.45	96.95	97.01	96.68	96.44
4 Cores	55.00	52.15	54.58	55.05	54.18
8 Cores	36.86	38.45	38.31	38.87	38.15
16 Cores	29.28	30.92	30.92	31.01	31.20
32 Cores	33.71	34.00	33.90	34.25	34.46

TABLE IV
CPU RUNTIME FOR SECOND SETUP

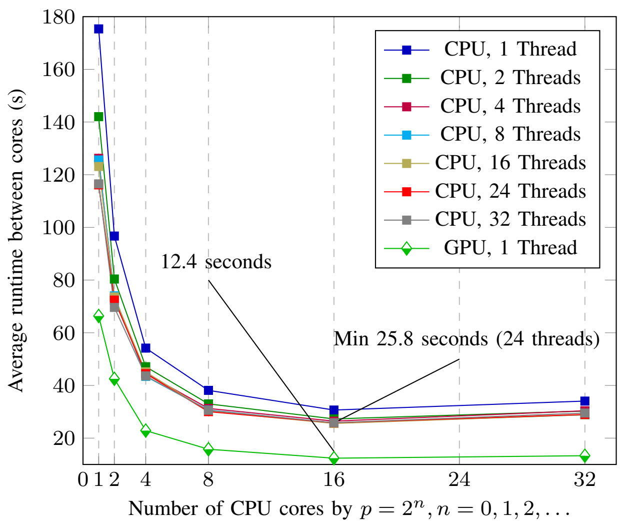
CPU Cores	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
1 Cores	66.53	66.67	66.63	65.72	66.31
2 Cores	42.62	43.26	42.22	42.29	42.19
4 Cores	22.47	22.58	22.62	23.79	22.83
8 Cores	15.20	15.73	16.23	15.77	15.98
16 Cores	11.61	12.27	12.88	12.44	12.71
32 Cores	12.99	13.44	13.21	13.38	13.56

TABLE V
GPU RUNTIME FOR SECOND SETUP

Similar to the previous setup, we observed a slight slowdown at 32 cores, which we suspected was due to the number of cores used exceeding the number of physical cores, forcing the computer to perform context-switching

3) CPU vs GPU, OpenMP: After verifying that the runtime follows a general pattern, we can now take the average runtime of the GPU and compare it with the CPU running on multiple threads (OpenMP).

Performance Comparison Between Different Threads, with GPU (10,000 features, Release, 64 images, $4k$)



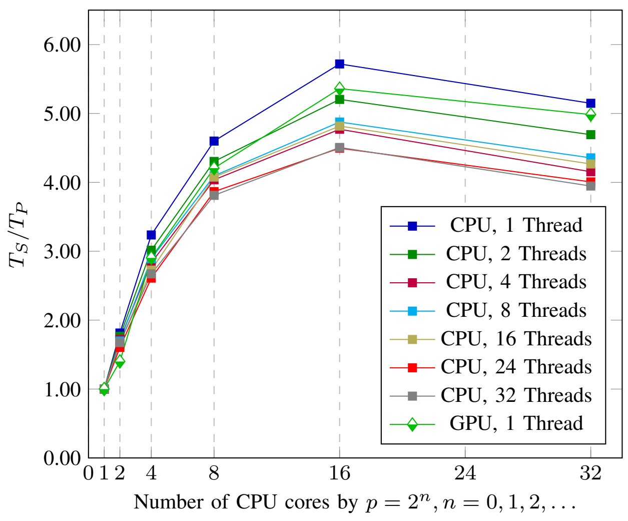
CPU Cores	**1**	**2**	**4**	**8**	**16**	**32**
1 Threads	175.36	96.71	54.18	38.13	30.66	34.06
2 Threads	142.03	80.40	47.11	33.00	27.29	30.27
4 Threads	126.22	73.31	44.57	31.27	26.46	30.38
8 Threads	125.60	74.01	43.43	30.67	25.76	28.84
16 Threads	123.12	73.58	45.16	30.20	25.56	28.86
24 Threads	116.14	72.39	44.50	30.04	25.83	28.99
32 Threads	116.53	69.61	43.62	30.57	25.86	29.53
GPU (Average)	66.37	42.52	22.86	15.78	12.38	13.32

TABLE VI
RUNTIME COMPARISON BETWEEN DIFFERENT THREADS AND GPU

C. *Speedup, Cost, and Efficiency*

1) *Speedup:*

Speedup Comparison Between Different Threads and GPU (10,000 features, Release, 64 images, $4k$)



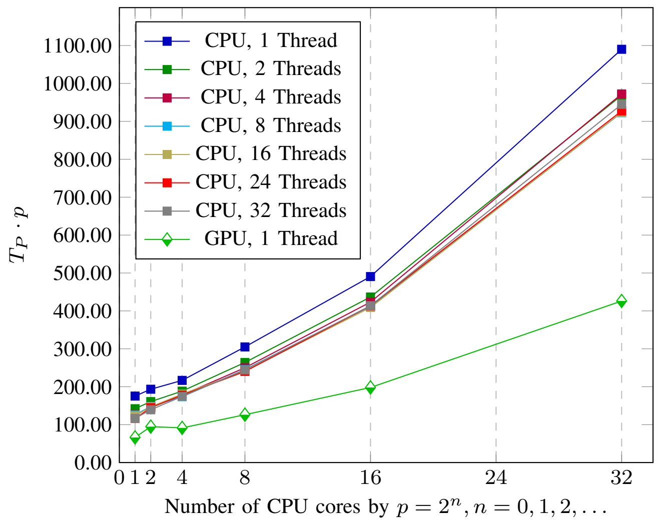
CPU Cores	**1**	**2**	**4**	**8**	**16**	**32**
1 Threads	1.00	1.81	3.24	4.60	5.72	5.15
2 Threads	1.00	1.77	3.01	4.30	5.20	4.69
4 Threads	1.00	1.72	2.83	4.04	4.77	4.15
8 Threads	1.00	1.70	2.89	4.09	4.88	4.36
16 Threads	1.00	1.67	2.73	4.08	4.82	4.27
24 Threads	1.00	1.60	2.61	3.87	4.50	4.01
32 Threads	1.00	1.67	2.67	3.81	4.51	3.95
GPU (Average)	1.00	1.40	2.90	4.21	5.36	4.98

TABLE VII
SPEEDUP COMPARISON BETWEEN CPU WITH DIFFERENT THREADS AND GPU

Even though the runtime has massively decreased under the usage of CUDA, we noticed that the speedup ratio does not stand out among the CPU with the OpenMP dataset, but it did, however, speedup in all scenarios of cores against CPU with 1 thread, which is reasonable since our GPU setup also uses only one thread of the CPU (no OpenMP available for GPU). We suspect the reason why the GPU doesn't speed up as fast is due to the uploading and downloading of images to the GPU, which costs much more resources due to the high quality of the image patch.

2) *Cost:*

Cost Comparison Between Different Threads and GPU (10,000 features, Release, 64 images, $4k$)

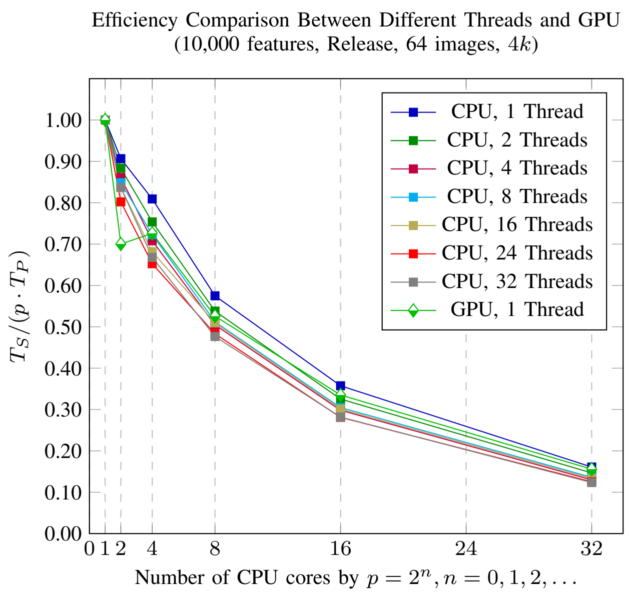


CPU Cores	**1**	**2**	**4**	**8**	**16**	**32**
1 Threads	175.36	193.43	216.72	305.03	490.61	1090.06
2 Threads	142.03	160.80	188.45	264.02	436.66	968.61
4 Threads	126.22	146.61	178.26	250.13	423.31	972.30
8 Threads	125.60	148.03	173.72	245.37	412.15	922.72
16 Threads	123.12	147.16	180.64	241.63	408.94	923.39
24 Threads	116.14	144.79	177.99	240.36	413.26	927.52
32 Threads	116.53	139.22	174.48	244.58	413.70	944.87
GPU (Average)	66.37	94.79	91.43	126.24	198.10	426.14

TABLE VIII
COST COMPARISON BETWEEN DIFFERENT THREADS AND GPU

Moreover, our method demonstrates superior cost-effectiveness compared to all previous implementations, even when utilizing only a single thread.

3) *Efficiency:*



CPU Cores	1	2	4	8	16	32
1 Threads	1.00	0.91	0.81	0.57	0.36	0.16
2 Threads	1.00	0.88	0.75	0.54	0.33	0.15
4 Threads	1.00	0.86	0.71	0.50	0.30	0.13
8 Threads	1.00	0.85	0.72	0.51	0.30	0.14
16 Threads	1.00	0.84	0.68	0.51	0.30	0.13
24 Threads	1.00	0.80	0.65	0.48	0.28	0.13
32 Threads	1.00	0.84	0.67	0.48	0.28	0.12
GPU (Average)	1.00	0.70	0.73	0.53	0.34	0.16

TABLE IX
EFFICIENCY COMPARISON BETWEEN DIFFERENT THREADS AND GPU

For efficiency, our algorithm is less efficient than the CPU, 1 thread setup, but is overall more efficient than other CPU setups as the number of cores increases. The efficiency of the GPU implementation eventually matches that of the CPU, 1 thread implementation as $p = 32$.

V. CONCLUSION

The Graphics Processing Unit has allowed large-scale parallel computation to be performed on intensive tasks such as image processing, and by utilizing this, alongside the CUDA platform provided by Nvidia, we were able to reduce significantly the runtime and resource cost while maintaining the same image quality as those produced by CPU-based methods. By integrating distributed memory design to run our application on multiple cores, we were able to speed up the process even further, stitching sixty-four, high resolution 4K images in less than 12 seconds, which is more than double the speed of that performed by the CPU at the same number of cores. We also found that our efficiency increases as the number of cores increases, and eventually asymptote the efficiency of the CPU implementation.

One drawback to this method is that it is a generally memory-costly method. However, if the user lacks a high-quality camera or needs to create a map quickly without focusing on fine details, the image patch can be downscaled to meet these requirements. This approach has been validated in the first setup, where the 4K patch was downscaled by 75%. This can be achieved easily by modifying a small part of the code base, which is not the main discussion of this report. Another issue with the 4K patch is that it was not able to stitch parts of the map where there are rotations due to memory issues. After careful inspection, we deduced that this is not a problem with our code, but due to memory issues, since if we stitch the same patch with downscaled quality, it was able to stitch the patch successfully without issue.

Integrating shared memory design for multi-threading with the GPU is still an open problem, and we plan to address this in future projects to accelerate our algorithm even further. The ultimate goal of this project is to process 64 frames of 4K images in one second. We also plan to resurvey our memory management in the code base to ensure that the program does not crash when the input is too large and to generate more consistent and reliable results. Another plan we have for the future is to run this application on a high-performance computer cluster (HPC), where the number of cores can go up to hundreds, or even thousands of cores. This can potentially reduce the runtime to only a few milliseconds, opening new frontiers to real-time image-mapping.

ACKNOWLEDGEMENT

We would like to acknowledge with our most sincere gratitude to the following individuals and organizations for their support and for providing priceless opportunities throughout this project.

First and foremost, we extend our heartfelt thanks to our advisors, Dr. John Korah and Rick Ramirez, for their unwavering guidance, technical expertise, and encouragement. Their dedication and countless hours of support each week have been instrumental in our learning and progress during this project.

We also wish to acknowledge the National Science Foundation and California Polytechnic State University for offering such an exceptional opportunity for us undergraduates, as well as for the financial support provided under Award #2024276.

We are profoundly grateful to Dr. Iraj Nejad, Dr. Alvin Kung, and Dr. Said Shokair for their efforts in bringing this program to Mt. San Antonio College, and for their detailed guidance, feedback, and invaluable advice. Their contributions have been crucial to the success of our research, and all of these would not have been possible without them.

Furthermore, we would like to express our gratitude to everyone who participated in the program. We appreciate your attentive listening to our presentations, your insightful questions, and your feedback on areas still needing improvement.

Finally, since this is a joint effort between us two, we would like to acknowledge each other. We are glad that we both work on this same project, and have each other's back, and we appreciate countless hours of troubleshooting issues,

discouragement from failures, and coming up with solutions to problems together.

References

[1] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[2] David G Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

[3] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011.

[4] Rick. Ramirez. Accelerated image stitching via parallel computing in unmanned aerial vehicle applications. *California Polytechnic State University*, pages 1–43, 2024.

[5] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV 2006*, pages 430–443, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Carsten Griwodz, Lilian Calvet, and Pål Halvorsen. Popsift: a faithful sift implementation for real-time applications. In *Proceedings of the 9th ACM Multimedia Systems Conference*, pages 415–420, 06 2018.

[7] Bhanu Priya Priya. Understanding nvidia cuda: The basics of gpu parallel computing. https://www.turing.com/kb/understanding-nvidia-cuda, 2024. Accessed: 2024-08-6.

[8] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS ’67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.