Spring 2020 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Accelerating Image Stitching by Exploiting Data Parallelism

Prathamesh Patel

May 6, 2020

Abstract

Image processing workloads often show tremendous potential for parallelism due to the independent operations on each pixel even though traditionally, they have been implemented sequentially on CPUs. Through our project, we study the potential ways to exploit this parallelism and compare the results and benefits of each method. We have studied an Image stitching algorithm implementation called OpenPano on multicore CPUs and GPUs. While we witness a significant improvement with multicores, the improvement is not so pronounced with GPUs. We further analyze and discuss the reasons for such behavior. We realize that although the image stitching algorithm is inherently friendly to parallelism, we need to be careful with overheads of launching parallel tasks as they can lead to performance loss if used blindly.

Link to Final Project git repo: *https://euler.wacc.wisc.edu/iyelurwar/imagestitching.git*

Contents

1. General Information

In this important section, please provide only the following information, in bulleted form (four bullets) and in this order:

1. Your home department: CS
2. Current status: MS student
3. Individuals working on the Final Project (include yourself)
    - Prathamesh Patel
    - Rahul Singh
    - Ishan Yelurwar
4. Choose one of the following three statements (there should be only one statement here):
    - I am not interested in releasing my code as open source code.

2. Problem Statement

Default project: Image Stitching

Image stitching algorithms are widely used in generating a panoramic image from multiple non-panoramic images. In the past, these algorithms have been predominantly implemented sequentially on CPUs. But when the input size is large, a sequential operation can be very costly. However, as the operation on each pixel is independent, theoretically, it can be parallelized with exceedingly better results compared to sequential implementation. We planned to draw a comparison between the traditional sequential algorithm, a parallel implementation on multicore CPU, and a naive and tiled GPU implementation. We planned to use OpenCV implementation of this algorithm, for parallel implementation on CPUs, we planned to use OpenMP for a machine with 20 cores. For GPUs, the plan was to use the CUDA framework as the GPUs on Euler are from NVIDIA.

3. Solution Description

As mentioned in the previous section our initial plan was to use the OpenCV implementation of the image stitching algorithm, however, after facing infrastructural issues while creating the setup on Euler we decided to use the already existing image stitching program OpenPano [1]. OpenPano is already a highly optimized implementation which made our jobs challenging. Once the setup was done on Euler, we optimized the code first for multicores by using OpenMP [2] and later by using CUDA [3] for GPUs. Given the limited time frame of this project, we chose to implement a naive version of both optimizations and a tiled CUDA optimization using shared memory as the other advanced optimizations require a thorough understanding of the entire code base. We discuss the details in the following subsections:

a) OpenMP Optimizations:

In order to speed-up the application, we experimented with many OpenMP features to achieve better performance. We tried out nested OpenMP (OMP) loops to achieve more thread-level parallelism in the program. We experimented with different OpenMP scheduling schemes such as static and dynamic thread execution for many functions in the codebase. We also tried out the various configurations of OpenMP binding to exploit data affinity in the program. We tested the application with a different number of OpenMP threads by setting the environment at run-time in the scripts and choosing the number of threads with the optimal performance. We were able to measure the performance impact of the functions with a combination of self-created timer using Chrono Library and an in-built timer function included in the OpenPano Libraries.

b) CUDA Optimizations:

In order to include CUDA optimizations in the program, we had to set up the infrastructure for running OpenPano along with the CUDA module on Euler. To achieve that we linked the CUDA libraries by manually modifying the Makefile with CUDA Version 10.2.
For targeting specific functions with the highest potential for performance gain, we profiled the code at a coarse level with an in-built timer function in the OpenPano Libraries. We found that the functionalities of Gaussian Blurring and Keypoint detection using DoG Computations consumed a significant amount of execution time. Hence, we planned to target these features for optimizations with GPU computing.
In order to not restructure the codebase too much and to maintain simplicity, we used a wrapper function for each targeted feature, and we implemented the CUDA Kernel within the wrapper function. Initially, we tried to optimize the Gaussian Blurring feature with naive GPU implementation. But, since the performance impact was not significant, we optimized the function further with a Tiled GPU implementation using shared memory. For optimizing the DoG computation, we targeted two functionalities - Pixel difference operations and magnitude calculations as the features consuming significant amounts of time. To speed up the execution of these features, we implemented CUDA kernels for each of these functions.

4. Overview of Results

Fig. 1 and Fig. 2 show the corresponding Inputs and Outputs of the program. In the example demonstrated, we have used 4 input images of flowers which are stitched together by the algorithm to create a single image.
We first profiled the entire code to see what the major bottlenecks in the code were. The basic algorithm can be broken in broad chunks where we shift our focus:

1. *Keypoint detection using Difference of Gaussians (DoG):* Image is resized and blurred multiple times to create a Gaussian Pyramid. This involves multiple computations on the image and has a potential for parallelism
2. *Pairwise Matching and Build trees:* Pairwise matching is done for images with each other and a matching confidence is maintained. A tree is constructed with edge values based on these matching confidence values, establishing the best matches between images. The tree implementation is trickier to parallelize however, matching can be parallelized.

3. *Estimate Camera:* Even after matching, images may have different focal lengths and rotation. This is adjusted by using camera estimation which is an iterative process with little scope of direct parallelization.
4. *Blending:* This is the final stage where the transformation takes place on the overlapping regions to create a seamless stitched image. Since the transformations are on granularity of pixels, there is a scope of parallelization.

Gaussian Blurring and DoG computation (Keypoint detection) were one of the major components in the execution of the stitching algorithm. Also, other major bottlenecks were the blending algorithm and the matching algorithm that matches every image to every other image. These two parts also scaled up with the increase in the number of input images.
We ran all our implementations on the following OpenPano test images:

1. CMU0 (38 images - 1300 x 867)
2. CMU1 (13 images - 1500 x 1112)
3. Flower (4 images - 1200 x 795)
4. Myself (12 images 1200 x 795)
5. NSH (36 images - 1300 x 867)
6. UAV (19 images - 2000 x 1500)
7. Zijing (12 images 1200 x 795)

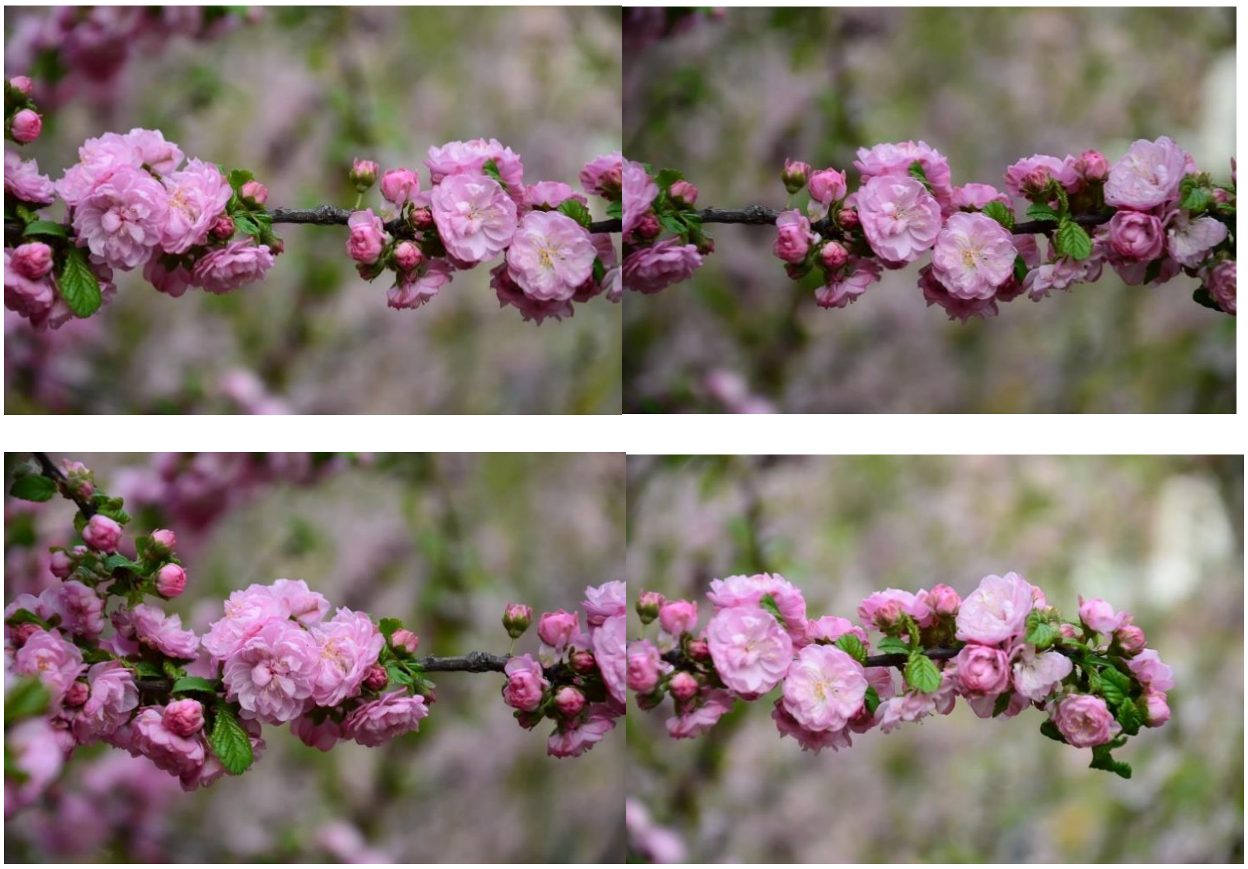

Fig.1. Flower Test Input (4 images)

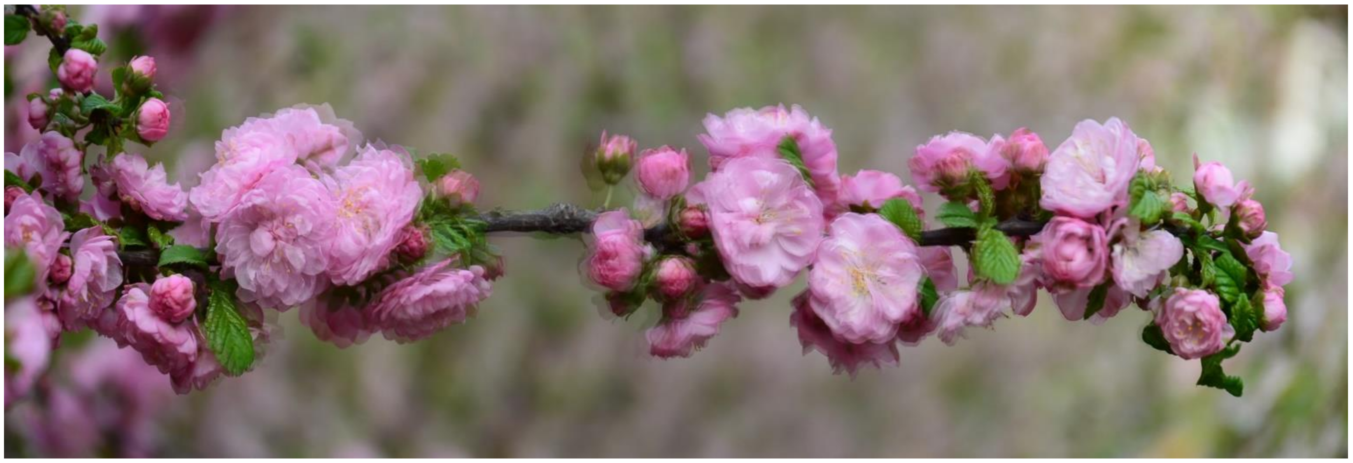

Fig.2. Stitched Output

The graph below shows the execution time for all the 3 versions. We break down the total execution time into different parts of the image stitching algorithm. The OpenMP version gave speedups ranging from 2.5x to 5x. The greater speedups were observed when the number of input images was greater as that increased the scope of parallelization.

Also, you can observe that the pairwise match, estimate camera, and blend operations scale as the number of input images to stitch together increases. The camera estimation implementation was a sequential RANSAC implementation, as a result, we were not able to parallelize much of the estimate camera portion of the algorithm.

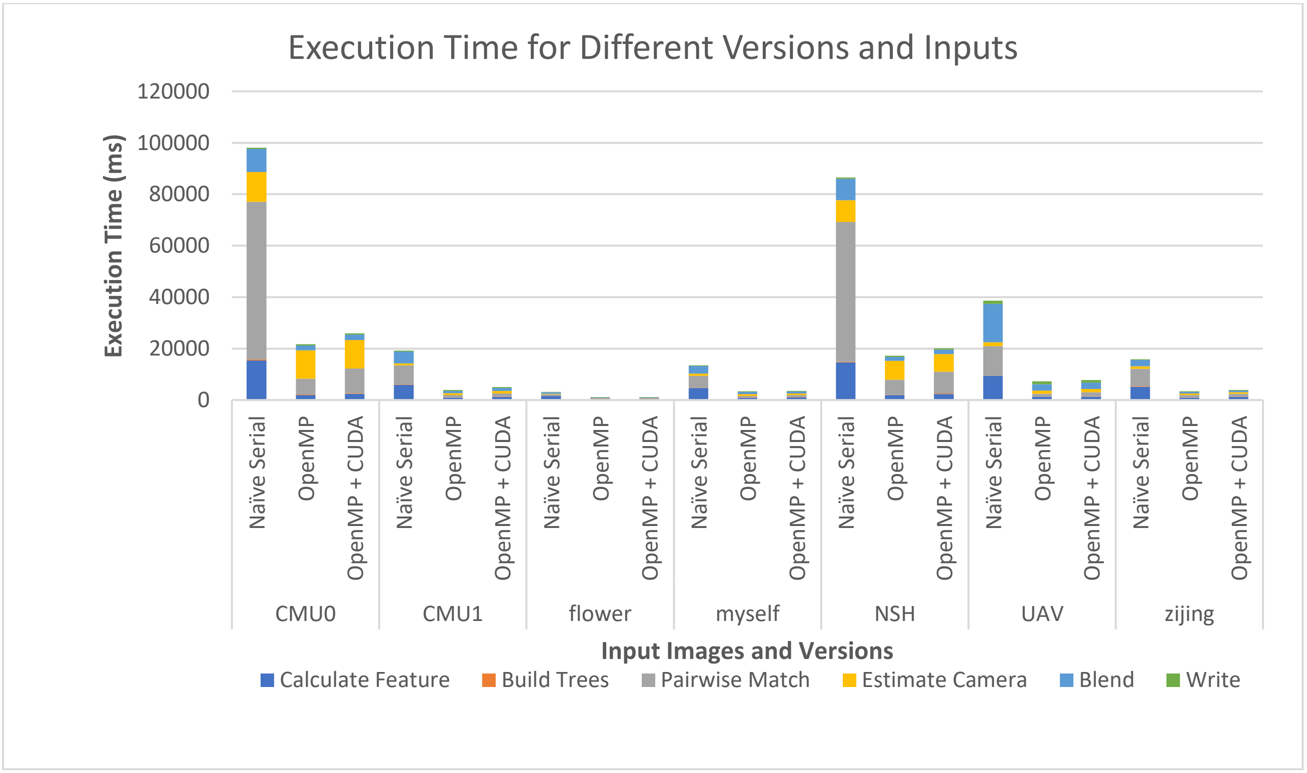

Fig.3. Execution Time for Input Images and Versions

Also, we can notice that the OpenMP+CUDA version performs almost similar or slightly worse in most of the cases. We tried to profile the individual CUDA kernels that we had optimized to find out the reason for this unexpected result.

In the graph below, you can view the execution time for the CUDA Gaussian Blur kernel and the OpenMP implementation. In most of the cases the CUDA implementation outperforms the OpenMP implementation. The overall execution time still increases for the OpenMP+CUDA version due to the time involved in the communication between CPU and GPU (cudaMemcpy). We observed that more than 50% of the time was spent in the communication between CPU and GPU. This resulted in the OpenMP implementation performing better than most of the OpenMP+CUDA hybrid implementations.

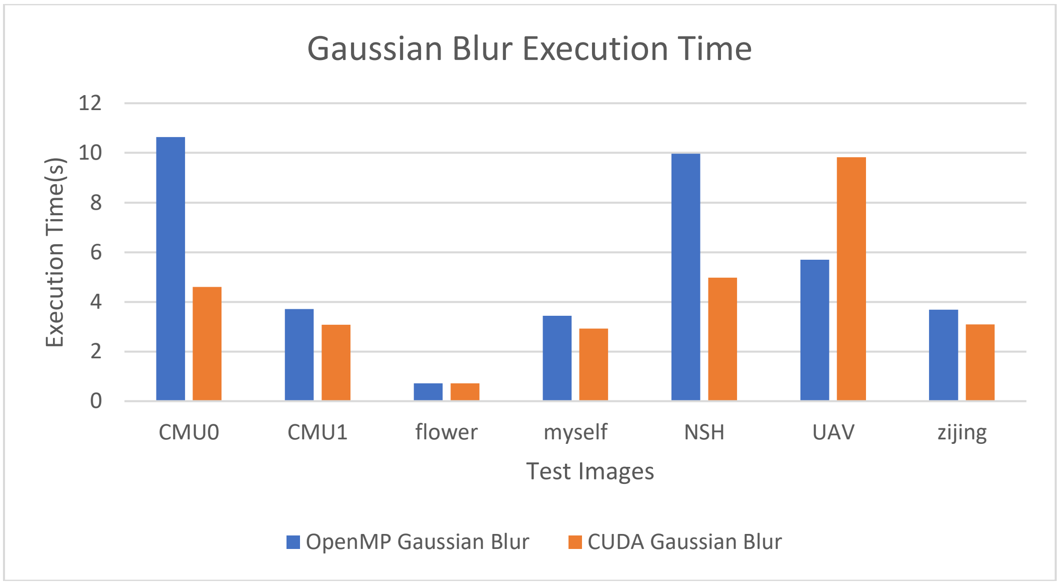

Fig. 4. Gaussian blur timing comparison between OMP and CUDA

In the graph below, you can observe that the DoG computation again performs better when implemented in CUDA but again due to the communication overheads we fail to see any significant benefits. Another reason why we don't see the desired speedups in the CUDA version could be due to the image resolutions that we had in the test images. We expect that CUDA implementation along with the communication overheads could outperform the OpenMP implementations at higher resolutions.

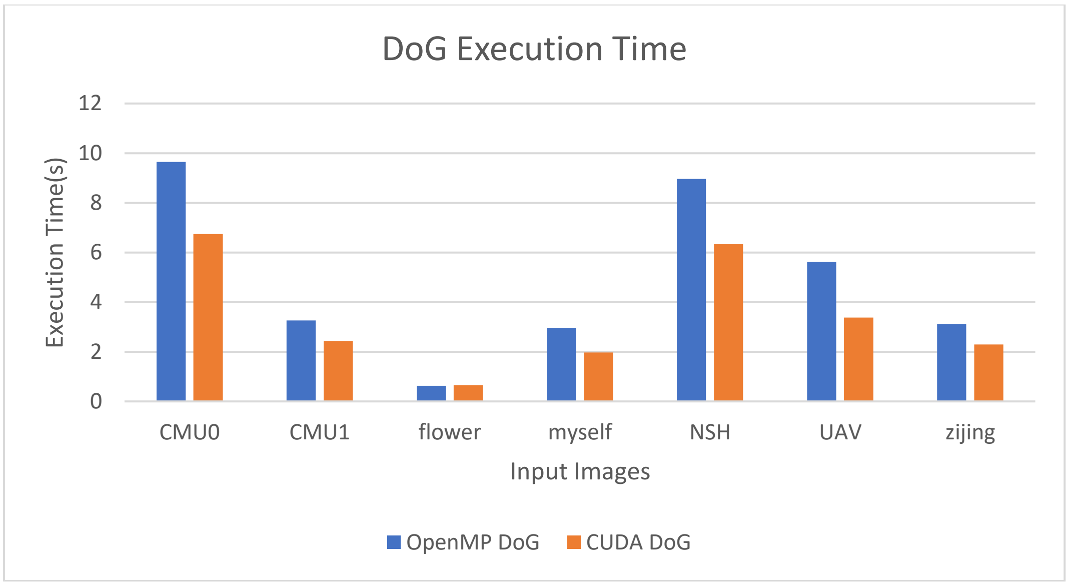

Fig. 5. DoG timing comparison between OMP and CUDA

5. Deliverables:

a. The final report on Canvas detailing our efforts with results and analysis of the project

b. Source code for sequential, parallel CPU and GPU versions through git repo with the following structure:

1. Serial implementation of OpenPano (OpenPano_v1_serial)
2. OpenMP implementation of OpenPano (OpenPano_v2_omp)
3. CUDA implementation of OpenPano (OpenPano_v3_cuda)
4. Script_runeuler (scripts to run the tests on any version)
5. tests/ (Input test cases)
6. outputs/ (Output of the test cases)

To run a specific implementation, enter the directory:

1. module load eigen (also load cuda if running the cuda version)
2. make -C src/
3. Use the scripts from the folder mentioned above (Script_runeuler)
4. This generates an output named out.jpg in src/ folder. (inside the specific implementation folder of OpenPano)
5. Timings can be checked in the output files generated

6. Conclusions and Future Work

From the above result and discussion, we conclude that the image stitching workloads have great potential for parallelization as anticipated. However, contrary to our beliefs the CUDA implementation doesn't provide a significant benefit, and in some cases, it even leads to a decline. As discussed, this is due to the overheads of data transfer before and after the kernels run the computation. From this observation, we conclude that the GPU implementation is not worth the effort and cost if the data is small (number of parallel threads is not high enough to mask the data traffic). In this regard, OpenMP proves to be very useful as it doesn't incur significant data traffic delays and provides a better speedup. However, there is potential for a much more detailed study by using advanced techniques of both OpenMP and CUDA. For the future work the areas that we can explore are:

a. Use streams to choreograph the data transfer to overlap computation with data transfer/ use Pinned memory (GPUs)
b. Use larger workloads so that we launch enough number of threads to mask data transfer (GPUs)
c. Explore the code base in detail for possible applications of OpenMP tasks and sections
d. With in-depth knowledge of code base, further optimizations with possible usage of CUB and Thrust Libraries can be explored.

Many of these implementations require a thorough understanding of the entire code base and the algorithm, which, given the time limit of this course were not feasible, but can be pursued in the future.

References

[1] https://github.com/ppwwyyxx/OpenPano
[2] https://www.openmp.org/
[3] https://developer.nvidia.com/cuda-zone