Parallelized Image Stitching

Adithi Rao, Vaishnavi Mantha

Implementation can be found at our GitHub repository.

Summary

We implemented an image stitcher for png images using OpenMP and enhanced sections of the pipeline with CUDA. Our initial goal was to have this work on videos, but due to issues with OpenCV, we had to manually process images on our own and therefore were not able to process videos. However, our algorithm is easily amenable to working with videos. We achieved a speedup of about 5x on 8 cores over our sequential implementation on average for any number and any size of input images fed into the program.

Background

The goal of our project was to build an image stitcher that takes input a sequence of rotation variant images and output a panoramic image. The image stitching pipeline can be broken down into the following stages: keypoint detection, keypoint matching, homography calculation via RANSAC, image warping, image placement onto final panorama.

The image stitching process can be understood by looking at pairs of images.
Keypoint detection is done using the SIFT (scale-invariant feature transform) algorithm. We then match the keypoints using a brute force comparison with using the Euclidean distance as a matching score. RANSAC (random sample consensus) is an iterative algorithm to find the homography that best models the necessary image warp. The algorithm randomly samples points required to fit the model from the list of key points from one image, solves for the model parameters (coefficients create the homography matrix), and scores the fraction of inliers within a set threshold of the model using the respective matched keypoints from the second image. This procedure is repeated for a number of iterations and the homography with the highest score is chosen. A homography is a projective transform matrix that determines how to transform a given image to best stitch with its adjacent input image. Image warping consists of multiplying the image channels by the best homography matrix. Image placement consists of computing the offset with respect to the panorama, copying the pixels over, and performing basic linear interpolation for a smoother looking image.

Our algorithm is extended to multi-image input by maintaining a left to right order while computing pairwise image matches, and while composing the found homography matrices for final panorama placement. Note that at this point, our program only works for images fed in such that the movement is left to right and will break/produce nonsensical outputs for any other image sequence.

The key data structures used per input image were three 1D arrays representing the color channels (R, G, B). Each input image also maintained a list of keypoints detected - to be able to

find matches between adjacent input images. Part of our pipeline also computes the final dimensions of the panorama and creates three arrays (for each color channel) to store the image. Key operations were performed on each of these structures. The keypoint lists were transformed into matrices and multiplied with candidate homographies. The image channel matrices are multiplied with the best found homography matrix to find the new image coordinates with respect to the final panorama. Pixels in the final panorama also undergo bitwise OR and interpolation operations.

Upon instrumenting our working sequential code, we found that the keypoint detection, RANSAC iterations, and image warping, and image placement were the expensive parts of the image stitching pipeline. The image-stitcher extends to multiple images so image-level parallelism was easy to exploit. The RANSAC algorithm has many iterations of the same computation, so iteration-based parallelism was used here. Image warping and image placement operations had many pixel based operations, so pixel-based parallelism was applied here.

The dependencies can be broken down into these stages: keypoint detection, keypoint matching, homography calculation via RANSAC, image warping, image placement onto final panorama. Different parts of this pipeline involve different types of parallelism - image based, iteration based, pixel based. Therefore there is some, but limited data-parallelism (we will go into more detail in the next section). SIMD execution does not really fit with our project because all of our computations require a lot of if statements/divergent execution, and we use a lot of higher level data structures that are hard to convert to arrays to use for SIMD, so we decided not to go with that approach.

Approach

This section will discuss implementation details and our parallelization strategies. Our final code for all versions of our approach is available on our GitHub repository.

Our image stitcher algorithm was parallelized using OpenMP and CUDA on the GHC GPUs.

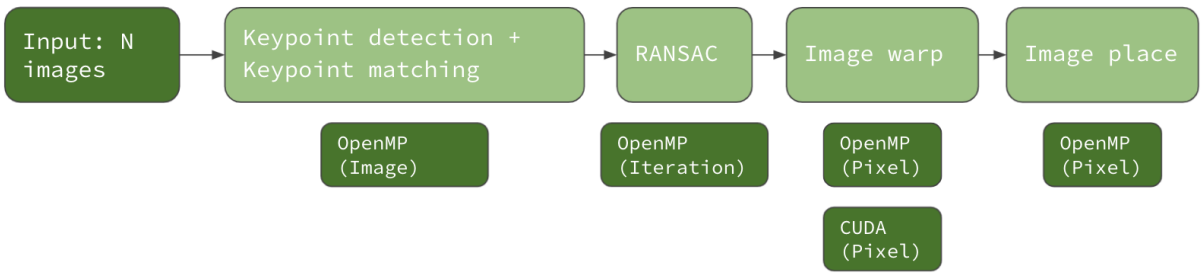

Figure 1: Image stitcher pipeline

Sequential Implementation

We coded our sequential image stitcher in C++. We used an existing open source sequential implementation for the SIFT algorithm (key point detection and matching) [1]. The remaining phases of the image stitching pipeline were programmed from scratch. Implementing most of the pipeline ourselves helped us familiarize ourselves with the data structures and computation being done, which made it easier for us to exploit different forms of parallelism.

Our initial sequential implementation was written using OpenCV functions for reading/writing images and image transforms. However, we had a lot of trouble transferring this version onto a GPU. Upon trying various GPUs (Nvidia Jetson TK1, GHC cluster, Latedays machines, Nvidia core machines), ran into issues with installing OpenCV and then linking both OpenCV and CUDA to our program. After spending over a week trying to set up our GPU, we decided to invest the time to rewrite all the OpenCV primitive functions ourselves. This caused us to sacrifice a little bit on the quality of our output image depending on the homography computed, since we do not use as fancy interpolation techniques as OpenCV does in warpPerspective.

OpenMP

The nature of the image stitching algorithm made it very amenable to OpenMP parallelism. We explored pixel based, image based, and iteration based parallelism. Of these, we preferred to exploit pixel-based because there are magnitudes more pixels than images/iterations.

We parallelize over the for loop that iterates over the images around the SIFT image feature detection algorithm since we needed to run it on each of the images. We made a serious effort into parallelizing the image SIFT algorithm itself but ran into several problems, including not fully understanding the algorithm that was used in ezSIFT and also not being able to parallelize it without dramatically changing the way the algorithm was written. Since this required a lot of knowledge of the computer vision components of feature detection and was not something we were very familiar with, we decided to focus our parallelization efforts on other parts.One more simple avenue of parallelism that we explored was with respect to the for loop over match_keypoints. We did not try to parallelize match_keypoints itself, because the time that it takes is quite trivial (usually less than a second given any pair of images), so we decided to only parallelize over the images.

In RANSAC, we parallelize over the iterations. OpenMP applies very well to this function because each iteration takes a while and does several complicated operations which would be hard to parallelize by converting to low level CUDA semantics. Each iteration also takes around the same amount of time, leading to no real work imbalance. The RANSAC algorithm updates a max score as well as variables associated with the score to keep track of the best found homography. To be able to parallelize this, we needed to pre-allocate an array to store scores from each iteration. We then use OpenMP parallel reduce to find the max score, associated index, and recompute the best homography.

Often in sequential image stitching algorithms, the homography matrix is computed for each image one step at a time, and does not allow for any parallelism when computing the transformation for each image onto the final image space. However, by premultiplying and

storing all of the homographies all with respect to the base image before entering the loop for warpPerspective and placeImage, we were able to explore parallelism between the images. We tried various parallelization strategies over the section of code where we warpImage and placeImage. placeImage could not be parallelized over images because we need to maintain the order of images investigated to get properly interpolated results. We tried parallelizing over images for warpPerspective, but this gave us little to no speedup (especially with only N = 2, 3 images) simply for the reason that we were not utilizing all the available cores. Next, we instead tried parallelizing over the pixels of each image. We saw better speedup because of the large number of pixels and thus independent work that the cores could split up. In our final implementation, we do a pixel-based parallelism for each investigated image in both warpImage and placeImage.

In summary, we had to change a limited number of data structures and loop structure from our sequential code to parallelize with OpenMP. Specifically, we needed to add new data structures to our RANSAC algorithm to be able to reduce in parallel. The main difficulties with OpenMP was determining which loops we should parallelize over (image, iteration, pixels). We also struggled to parallelize operations that involved std::vector (dynamically sized data structure) operations in the ezSIFT code. This was because vector push_back can lead to reallocation of the underlying array. We looked into Intel's thread building blocks for dealing with concurrent vectors [3], but ultimately decided that the involved code portions did not take too long and because concurrent vectors take longer than O(1) for random accesses, we would end up paying the cost later.

CUDA

We tried CUDA implementations of functions in our pipeline that had pixel-based parallelism because with CUDA we can spawn more threads and take advantage of each pixel having its own independent competition. The functions we tried were RANSAC and warpPerspective. We were looking for signs of scalability, so based on our instrumentation and speedup calculations (some of which can be found in the Results section), our final implementation only keeps a CUDA implementation of warpPerspective.

Rewriting warpPerspective as a CUDA function was not a trivial change. warpPerspective involves multiplying the x, y coordinates of each pixel in the image by a 9x9 homography matrix. Our sequential and OpenMP versions used the Eigen library for easy linear algebra. While writing kernel functions however, we faced a lot of trouble trying to get Eigen matrices or Eigen functions to work on the GPU. We had to convert the Eigen::MatrixXd structures to char* arrays and manually keep track of which elements should be multiplied together, which is not very sustainable for doing more complicated matrix multiplications. kernelWarpPerspective we let each thread take on one pixel's computation and do the corresponding homography multiplication.

Something interesting we noticed was that the first CUDA operation performed always took around 0.67 seconds to perform. After some research, we found this is because the first CUDA runtime call initializes the CUDA sub-system and therefore takes some time to complete. When

dealing with small numbers of images, this warm up time makes a difference so we were able to get a small reduction in time by writing a dummyWarmup function that does a bogus cudaFree(0) and calling it in a parallel for that usually takes around 1.4 seconds. Thus we were able to hide the latency for CUDA warm up time.

We also tried parallelizing our RANSAC algorithm over the loop that happens for each iteration to count the number of inliers for a given homography matrix. This attempt did not yield any considerable speedup. We suspected that this is because only 5 arithmetic operations were being performed for a small sized matrix (in the 100s representing matched points rather than over all pixels which were in the 10000s for warpPerspective). Therefore, the loads and stores needed to initialize all of the CUDA arrays, doing the small number of multiplications on these matrices, and then transferring them back to the host memory was not worth it.

Ultimately, we found that the overall algorithm that we used is not super easily convertible to using CUDA unless we totally restructure the code and make insights that modify the algorithm but preserve correctness. This has been done in several papers but all implementations involved increasing the complexity of the algorithm/making several approximations based on a deep knowledge of Computer Vision.

Results

We tested our algorithm using images from a panorama image database [2]. The database provided multiple scenes each with sequential, single-axis rotation variant images. We used these images as input to our algorithm. We inspected the stitched panorama by eye for the sequential image and used that as a correctness benchmark for our parallel implementations. Here are some sample results from our program:

Figure 2: Panorama inputs- 3 color images

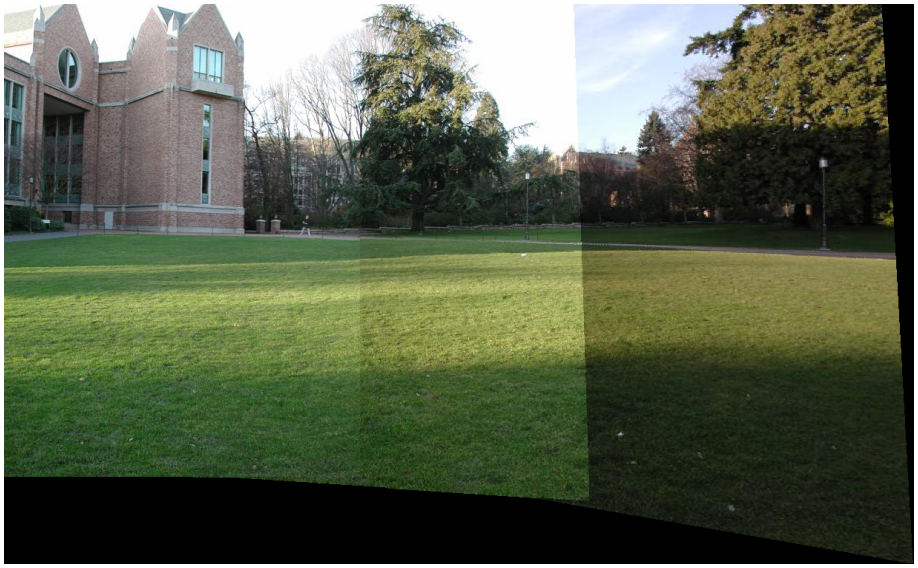

Figure 3: Panorama results with 3 color images above

Figure 4: Panorama inputs- 6 black and white images

Figure 5: Panorama output for the 6 images above

We measured the performance of our parallelization via speedup (sequential time / parallel time). Our sequential baseline is single-threaded CPU code. The final two parallel methods we ended up proceeding with to compare to our sequential version were a pure OpenMP version and a hybrid mix of CUDA and OpenMP. We decided to do this because most parts of our program could not be easily written with CUDA, so doing a normal comparison between just OpenMP and just CUDA would not be valuable. We analyzed each phase of our image stitching pipeline individually, since different parts are parallelized with different techniques/some parts could not be parallelized at all. Note that we use randomization in our RANSAC algorithm as well as the key point related functions, which could lead to slightly different times from the below times.

We will first analyze the pure OpenMP performance:

nThreads	Keypoint Detection	Keypoint Matching	RANSAC	Warp image	Place Image	Initial Read	Final Write	Overall
1	3.13652	0.0416449	1.37117	3.62162	0.954285	0.141808	0.181063	9.64614
2	1.88235	0.0419032	0.691802	1.909045	0.528091	0.0944583	0.181326	5.42507
4	1.32038	0.0416706	0.35752	0.977343	0.272061	0.0995449	0.18276	3.81305
8	1.32243	0.0420265	0.18507	0.497666	0.1411347	0.146532	0.181558	3.18624

Table 1: Pure OpenMP implementation times for 2 images

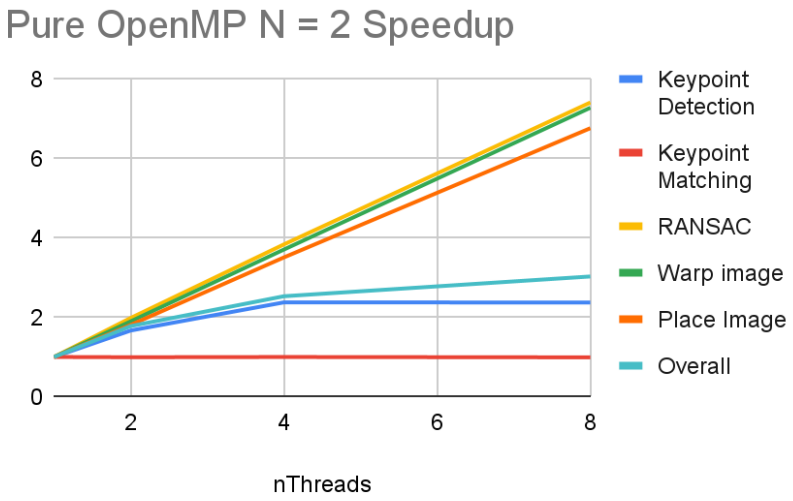

Figure 6: Pure OpenMP implementation speedups for 2 images

nThreads	Keypoint Detection	Keypoint Matching	RANSAC	Warp image	Place Image	Initial Read	Final Write	Overall
1	5.76212	0.490625	25.7317	7.2604	2.597736	0.216328	0.42741	43.6754
2	3.21454	0.336845	12.9704	3.780757	1.428123	0.23108	0.431056	23.605
4	1.382	0.293513	6.68384	1.933044	0.736537	0.230956	0.429275	13.1103
8	1.40229	0.2992	3.45678	0.993203	0.3909045	0.210709	0.434374	8.89626

Table 2: Pure OpenMP implementation times for 4 images

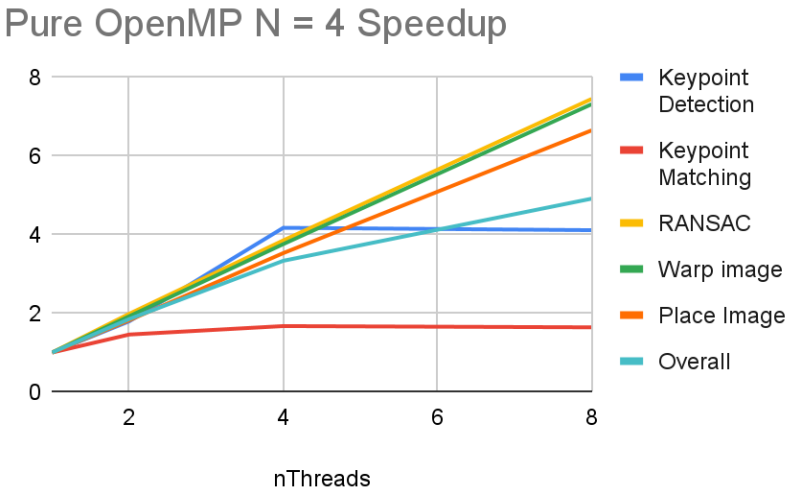

Figure 7: Pure OpenMP implementation speedups for 4 images

nThreads	Keypoint Detection	Keypoint Matching	RANSAC	Warp image	Place Image	Initial Read	Final Write	Overall

1	8.3322	0.651619	30.2694	10.78994	10.068056	0.289579	1.58794	67.4717
2	4.49222	0.331416	15.2671	5.615459	5.350732	0.307354	1.58588	37.3639
4	2.6106	0.292575	7.84499	2.883241	2.851867	0.296045	1.59227	23.6956
8	1.42941	0.297012	4.05825	1.478716	1.4935748	0.286788	1.59251	14.4287

Table 3: Pure OpenMP implementation times for 6 images

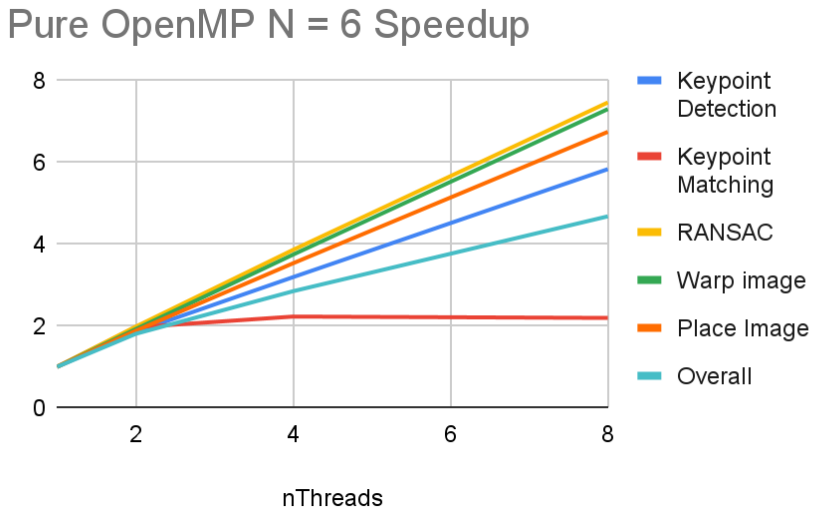

Figure 8: Pure OpenMP implementation speedups for 6 images

Figures 6, 7, and 8 show speedup graphs from our Pure OpenMP implementation. Our speedups across a varying number of cores for each phase of the image stitching pipeline have different slopes - based on the amount of independent tasks there are to be parallelized over which is what we parallelize over.

RANSAC, warp image, and place image all involve either iteration based or pixel based tasks, both of which are in very high quantity (1000+ tasks), and therefore we were able to get linear speedup as we increase the number of threads, since more threads just means more workers available to pick up the small fine-grained tasks. This is why we have straight lines for our speedups no matter how many images we are considering.

For keypoint detection, as the number of images increases, we get a more straightened out speedup graph. Varying workloads, in this case, number of images, exhibit different execution behavior. Keypoint detection was parallelized over images, so with fewer images (N = 2) there is not much work to distribute to nThreads > 2. With more images, however, there is more work to be distributed to worker cores, so we see more of a speedup for N = 6. For more images, we see expected behavior, that the parallel image stitcher takes less execution time with a higher number of cores.

For keypoint matching, we did not see much of a speedup. We can explain this because one iteration of running keypoint matches has a lot of sequential work that cannot be parallelized.

Note that each match is between a pair of images, so if we have N = 2 images, the associated for loop will only run one time. This explains why the speedup plot for N = 2 has the same runtime for a varying number of nThreads - there is only one iteration, and therefore there is no benefit from adding more threads because they will just be idle. A similar analysis was done to understand our N = 6 images results - we see a very slight speedup until around 6 threads, and see absolutely no benefits after since there are only 5 independent pieces of work to be done. Note that although we noticed that this part of the program can have a lot of idle threads, we are not able to move forward with the next step of the pipeline because all matches need to be computed before we start computing homography matrices and doing RANSAC. Another key thing to note is that the ezSift algorithm for matching could have been parallelized over each of the matches, but the way the code was written made it quite hard to parallelize without rewriting a lot of the logic. Moreover, since this part of the code did not take that much time, we decided to just go with an image based parallelism approach.

The initial input images reading and final panorama image writing took the same amount of time no matter how many threads we use, but these sections were not parts that could be parallelized.

Now we will discuss the CUDA + OpenMP based method that was our overall best version. Since we did not end up using CUDA for any function other than warpPerspective and used OpenMP in the same ways as we did in our pure OpenMP for everything else, we will simply discuss the times we were able to obtain for this single function.

Num Images	Sequential	Pure OpenMP (8 cores)	OpenMP + Cuda
2	3.62162	0.497666	0.00272612
4	7.2604	0.993203	0.00749263
6	10.78994	1.478716	0.02205128

Table 4: Warp Perspective times for different parallelization implementations

As you can see from the table we had a significant speedup for any given number of images once we changed our warpPerspective function to use CUDA. Note that the Pure OpenMP numbers are the best times we achieved with OpenMP, which was when we used the maximum number of cores available to us on the GHC machines which was 8. From this table, you can see that using CUDA gave a speedup of 489x over the sequential version and 67x over the OpenMP version for N = 8.

The limitations on our overall speedup ended up being the inherently sequentially independent parts of our pipeline. Though the speedup that we were able to achieve on warpPerspective was really exciting and prompted us to try and “cudafy” the other time consuming functions, the data structures we used and the complex logic of our code in most other functions made this quite hard to do. Specifically, CUDA was hard to implement for some of the ezSIFT functions because it would involve changing the algorithm significantly and completely eliminating the use of convenient data structures. An example of such a function is building the gaussian pyramids

for the SIFT detector. This section is costly and builds three pyramids (Difference of Gaussians, Gaussian pyramid gradient, rotation pyramids) that are all sequentially dependent on each other. We invested some time to understand how these pyramids are constructed to try and reorganize the code for parallelism but ultimately moved on. Another significant time sync in our most optimized version is RANSAC. This algorithm is iteration-based, so we could decrease the number of iterations for lower times, but a higher number of iterations probabilistically yields a better homography matrix. Parallelizing over more than the iterations for the RANSAC algorithm with CUDA is difficult because we would have to change our algorithm a lot to create more data parallel operations. The attempt we made to optimize a small loop inside a single RANSAC iteration showed little speedup, again proving that we would need to rework a lot of the algorithm.

The percentages for the different parts of our pipeline can be summarized roughly by the following figure, but note that they do vary drastically on what inputs are fed in:

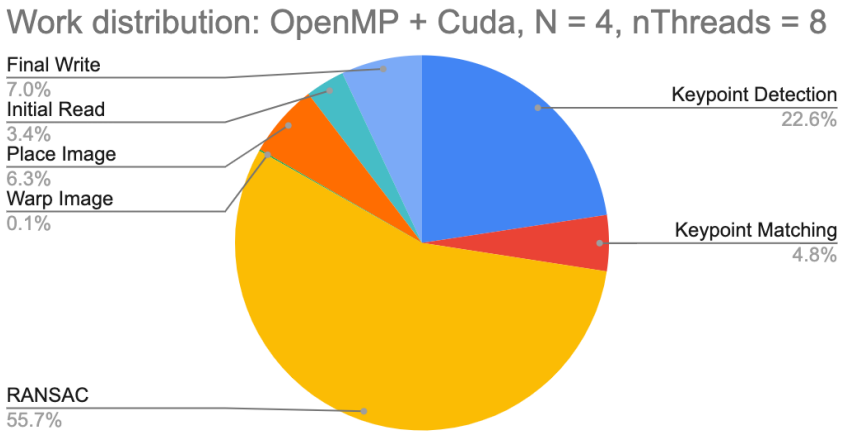

Figure 9: Workload distribution

Our workload was well suited for GPUs and given more time, we would attempt to understand a lot of the more complex algorithms and approximations so that we could use algorithms and data structures more suited for parallelization on GPUs. From our research though, transitioning all the code using these optimizations would be a serious time investment and was not feasible in the 4 week time period we had for the project.

References

1. ezSIFT Open Source sequential implementation: https://github.com/robertwgh/ezSIFT
2. Adobe Panoramas Dataset https://sourceforge.net/adobe/adobedatasets/panoramas/home/Home/
3. Intel Thread building blocks concurrent vectors https://www.threadingbuildingblocks.org/docs/doxygen/a00046.html

Division of Work

Equal work was performed by both project members.