CUDA-accelerated feature matching for image stitching

Jia-Shen Boon
University of Wisconsin-Madison
boon@cs.wisc.edu

Abstract

Image stitching is an algorithmic pipeline that takes as input an unordered set of images and combines a subset of the input into one continuous canvas. It has applications in a wide variety of fields, including mobile phones, agriculture, military surveillance and medical imaging. For large problem sizes, the compute time of the pipeline is dominated by a step known as "feature matching". How to bring this time down is still an area of active research. In this paper, we speed up feature matching by implementing it on a CUDA-enabled GPU. We present performance results and discuss future plans.

1. Introduction

Image stitching (a.k.a. panorama stitching, image mosaicing) is an algorithmic pipeline that takes as input an unordered set of images and combines a subset of the input into one continuous canvas. It has applications in a wide variety of fields, including mobile phones, agriculture, military surveillance and medical imaging.

2. Background

Our lab applies image stitching to the task of crop monitoring. We have developed unmanned aerial vehicles (commonly known as "drones"), each equipped with a downward-facing camera. See Figure 3. The UAV autonomously images a field and the images are stitched together offboard (see Figure 1). We have also developed image processing methods to identify individual crops. In short, farmers can remotely monitor the health of individual crops over time.

2.1. Image stitching pipeline

A generic, automatic image stitching pipeline is given in Figure 2.

To set up the backdrop of feature matching, we now briefly describe the steps upstream from it in the pipeline. First, given m images, a variable number (n_i) of keypoints

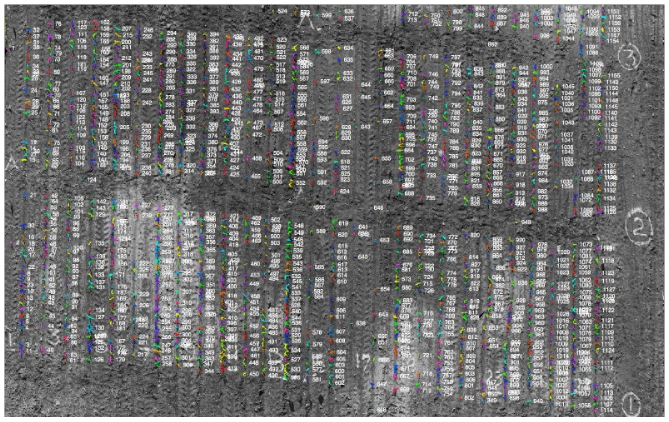
Figure 1: 4800 images of a field stitched together by our stitching algorithm. Every crop is individually identified.

is detected in every image, for $i = 1, \dots, m$. A keypoint is an x-y image coordinate where something interesting is happening. Next, the region around every keypoint is described by a k-dimensional vector known as a descriptor, where k is a constant. The choice of detector and descriptor is application-dependent. In short, the input to a feature matcher is a $n \times k$ matrix of floating point values, which is a vertical stack of $n_i \times k$ matrices, for $i = 1, \dots, m$ and $n = \sum_{i=1}^{m} n_i$.

The task of a feature matcher is then to find the correspondence between every descriptor j and every image i, for $j = 1, \dots, n$ and $i = 1, \dots, m$. Hence, the output is a $m \times n$ matrix of integer values. The purpose of matching features is to determine which pairs of images overlap, and compute the transformation between such pairs. These are the steps downstream from feature matching.

2.2. Typical numbers

Here are the typical numbers in our crop monitoring application, within an order of magnitude. The number of images to be stitched, $m \approx 1000$; the number of descriptors per image, $n_i \approx 500$; the size of one descriptor, $k \approx 32, 64$. This also means that the total number of features, $n \approx 5 \times 10^5$.

2.3. Description of task

Here we refine the task of feature matching. Specifically, we define how we determine descriptor-to-descriptor correspondence, if it exists at all.

As aforementioned, the subtask of feature matching is to find the correspondence between a descriptor j and an image i for all $j \in [1, n], i \in [1, m]$. This is equivalent to finding the correspondence between a descriptor and n_i other descriptors. The output of this subtask is the index of one of these n_i descriptors, or report that no correspondence exists. Determining correspondence is to some extent based on heuristics, and we use the same heuristic as is found in OpenCV. The heuristic that we use is as follow:

Given: query descriptor q, n_i descriptors $X = \{x^{(1)}, x^{(2)}, \ldots, x^{(n_i)}\}$ and threshold value T.

Criterion: Let i_1, d_1 be such that $x^{(i_1)}$ is the nearest neighbor of q in X with a distance from q of d_1. Let i_2, d_2 be such that $x^{(i_2)}$ is the second nearest neighbor of q in X with a distance from q of d_2. Then $x^{(i_1)}$ corresponds to q if $d_1/d_2 < T$. Otherwise q has no correspondence.

The above criterion is based on the following assumption. Correspondence to a descriptor is found when its nearest neighbor is much closer to it than all other neighbors. In other words, the descriptor's distance to its nearest neighbor is much smaller that its distance to its second nearest neighbor. Otherwise, if these two distances are similar, then correspondence is weak and we say that there is no correspondence to the query descriptor. We use Euclidean distance (a.k.a L_2 distance) throughout this paper.

It is worth noting that not only do other criteria to determine correspondence exist, even the formulation of the feature matching task in general can be done in different ways. See next section for another formulation. We choose the above formulation and criterion because these are employed in OpenCV and seem like a good starting point.

3. Related work

Image stitching is a well-studied problem in computer vision [6]. While the image stitching pipeline is elaborate (see Figure 2), its compute time is dominated by the feature matching step for large problem sizes due to its quadratic time complexity in a naive approach. Strategies to bring this compute time down can be classified into three broad categories: 1) use more efficient data structures and algorithms, 2) reduce the problem size with heuristics, 3) increase computational power with parallel hardware. In this paper, we explore the third strategy by implementing feature matching on a CUDA-enabled GPU.

Since feature matching can be seen as a variant of k nearest neighbors (kNN), many kNN data structures and algorithms directly apply here. This includes kd trees and its variants [4], 'slicing' [5], and locality sensitive hashing.

[1]http://docs.opencv.org/3.0-beta/modules/stitching/doc/introduction.html

(a)

(b)

Figure 3: UAVs, each equipped with a downward-facing camera. These UAVs take images that are inputs to the image stitching pipeline. (a) A 3DRobotics IRIS+ with a GoPro camera attached with a 200 mm lens. (b) A custom hexacopter.

Instead of finding correspondence between every descriptor and every image, [1] proposes to find the 4 nearest neighbors of every descriptor in the space of all descriptors, then each image is limited to being matched with at most 6 other images based on the number of correspondences found in the previous step. This allows them to build a single kd-tree, whereas in our approach we would need one kd-tree per image. Other possible heuristics include retaining just a subset of the input images when there is significant overlap between images, and limiting the number of descriptors per image.

There are many parallel implementations of feature matching and other steps of the stitching pipeline. OpenCV has implemented feature matching in both the Tegra multicore system-on-a-chip and CUDA-enabled GPUs. However, the CUDA implementation launches one kernel per image pair, which, depending on its execution configuration, suggests that there is further room for optimization. [3] also implemented a generic kNN solver on multiple GPUs. With regards to kd trees on a GPU, there has been work on minimizing execution divergence when running a tree-

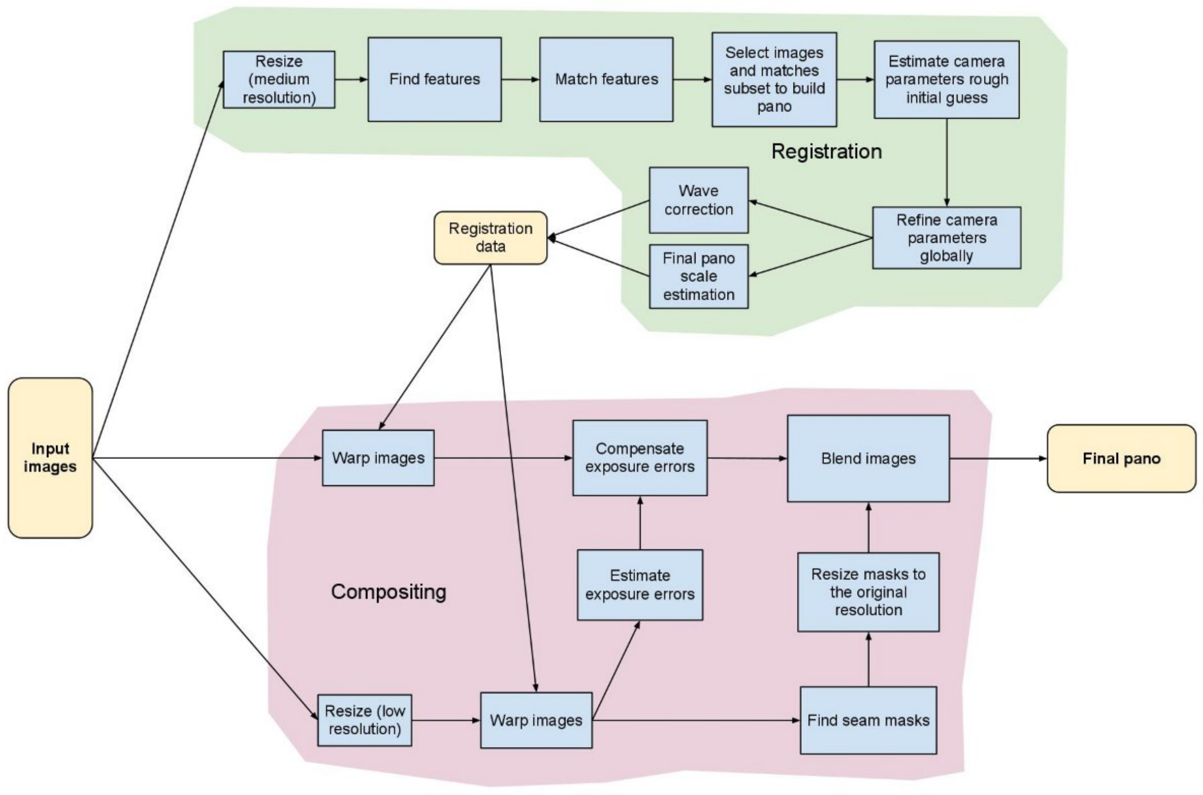
Resize (medium resolution)
Find features
Match features
Select images and matches subset to build pano
Estimate camera parameters rough initial guess
Registration
Wave correction
Registration data
Refine camera parameters globally
Final pano scale estimation
Input images
Warp images
Compensate exposure errors
Blend images
Final pano
Compositing
Estimate exposure errors
Resize masks to the original resolution
Resize (low resolution)
Warp images
Find seam masks

Figure 2: A generic image stitching pipeline[1]. In this paper, we optimize the upper box labeled 'Match features'.

traversal algorithm on a GPU [2].

4. Methodology

We use the following terms in this section. *Descriptors*, unless otherwise specified, is a $n \times k$ matrix of floats, where each row is a descriptor. A *distance matrix* S is a $n \times n$ matrix of floats where S_{ji} is the distance between the j-th and i-th row of the descriptors. A *correspondence matrix* C is a $m \times n$ matrix of integers where C_{ji} is the i-th descriptor's corresponding descriptor in image i, or -1 if no correspondence is found.

Every approach below builds on the findings of the previous one.

4.1. Naive approach

We first take a naive approach. A function computes the distance matrix from the descriptors, and another one computes the correspondence matrix from the distance matrix. Both CPU and GPU versions have been implemented and correctness is verified by looking at the root mean squared error between the GPU and CPU outputs. The root mean squared error between two 2D matrices A and B is given by $\sqrt{\sum_{i,j}(A_{ij} - B_{ij})^2}$.

This approach is easy to reason about and implement, but the distance matrix requires $O(n^2)$ space, which makes the approach infeasible for any real-world range of n. All approaches proposed in this paper take $O(n^2k)$ time.

4.2. Space-optimized approach

To remedy the above space complexity problem, the next approach directly computes the correspondence matrix from the descriptors. All values in the distance matrix are still computed in this approach, but they are assigned to temporary variables, thereby circumventing the $O(n^2)$ space complexity. This takes $O(nm)$ additional space, aside from the descriptors' $O(nk)$ space. This $O(nm)$ space complexity is optimal since that is exactly the space required for the output matrix. We cannot do better than this. We have also implemented this approach on the CPU and GPU.

The execution configuration is as follows. One kernel is launched per row of the correspondence matrix. Each thread in the kernel computes one element.

4.3. Occupancy-optimized approach

We use the Nvdia occupancy calculator to determine the block size that maximizes occupancy. The space-optimized approach above uses 21 registers, as stated by the compiler, and zero shared memory. The block size was previously set to the maximum of 1024 which gives an occupancy of 67%; the calculator reveals that setting the block size to 192 instead would maximize occupancy at 88%.

4.4. Memory-coalesced approach

The next optimization step addresses the memory access patterns of the threads in the same warp. Note that each 2D matrix is in fact implemented as a single contiguous block of memory in row-major order. For example, a 2×3 matrix A has $A[0], A[1], A[2]$ as the first row and $A[3], A[4], A[5]$ as the second row.

The above implementation detail has significant impact on the performance of all the previous approaches. In particular, in each iteration, every thread in a warp accesses one element in the column of the $n \times k$ descriptor matrix D. For example, a warp may access $D[0:32][0]$, then $D[0:32][1]$ in the next iteration, then $D[0:32][2]$, etc. In other words, under the hood, each thread is accessing an address that is k elements (or `k*sizeof(float)` bytes) apart. D was initially chosen to be of size $n \times k$ so that each descriptor is a contiguous block of memory, but as shown above, this choice results in highly uncoalesced memory access.

In the memory-coalesced approach, we transpose the descriptor matrix to be a $k \times n$ matrix instead. Now, neighboring threads access neighbors in memory, at the expense of slightly more cumbersome pointer arithmetic in implementation. We verify the correctness of the implementation against the CPU space-optimized implementation.

5. Results

In the following experiments, we fix $k = 32$ and $n_i = 100$ for all images. m is the independent variable while time is the dependent variable. In the case of GPU performance, we use inclusive timing but we have observed that compute time dominates inclusive timings, meaning that exclusive and inclusive timings have negligible difference (less than 1%). Descriptors are randomly generated floats between -1 and 1.

Figure 4 shows the performance of three of our approaches. In all conditions, we set $m = 2^1, 2^2, 2^3, \ldots$ until memory cannot be allocated for some large m. As expected, the CPU-implemented space optimized approach takes time quadratic in n, as seen by the slope of 2. The GPU equivalent exhibits time complexity that is slightly worse than

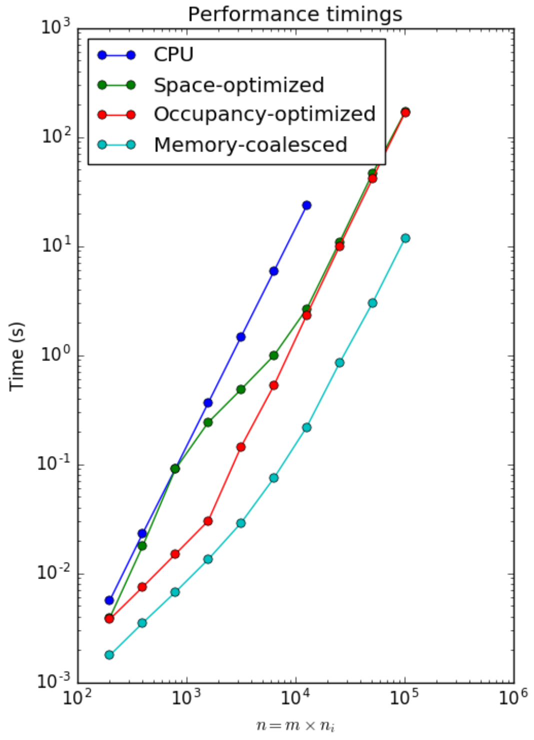

Figure 4: Performance results of some of our approaches, in a logarithmic scale. The plot has equal aspect ratio.

quadratic, given by a slope that is slightly greater than 2. The GPU equivalent outperforms the CPU for all m and up to 8.8 times when $m = 128$. Increasing occupancy improves performance slightly for large m, and significantly for small m. The former is further evidence that this task is compute-bound. The memory-coalesced approach drastically outperforms all other approaches, with 14.3 times better performance than the next best approach at $m = 1024$.

We exclude performance timings for the naive approach since the approach as no real-world value, as aforementioned.

6. Conclusion

We experiment with several approaches to speeding up feature matching with a GPU. Results look promising. In future work, we can look into using tree-like data structures which should improve time complexity to $O(n \log n)$.

Slides and source code for this paper are available online[2].

[2]`https://github.com/boonjiashen/stitching_with_cuda`

References

[1] M. Brown and D. G. Lowe. Automatic panoramic image stitching using invariant features. *International journal of computer vision*, 74(1):59–73, 2007.

[2] T. Karras. Thinking parallel, part ii: Tree traversal on the gpu. `http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-ii-tree-traversal-gpu/`. Accessed: 2015-12-21.

[3] K. Kato and T. Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 769–773. IEEE Computer Society, 2010.

[4] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2, 2009.

[5] S. Nene, S. K. Nayar, et al. A simple algorithm for nearest neighbor search in high dimensions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(9):989–1003, 1997.

[6] R. Szeliski. Image alignment and stitching: A tutorial. *Foundations and Trends® in Computer Graphics and Vision*, 2(1):1–104, 2006.