Fast Panorama Stitching

Introduction

Taking panoramic pictures has become a common scenario and is included in most smartphones' and tablets' native camera applications. Panorama stitching applications work by taking multiple images, algorithmically matching features between images, and then blending them together. Most manufacturers use their own internal methods for stitching that are very fast. There are also a few open source alternatives.

For more information about how to implement panorama stitching as well as a novel dual camera approach for taking 360 panoramas, please see my previous post here: http://software.intel.com/en-us/articles/dual-camera-360-panorama-application. In this paper we will do a brief comparison between two popular libraries, then go into detail on creating an application that can stitch images together quickly.

OpenCV* and PanoTools*

I tested two of the most popular open source stitching libraries: OpenCV and PanoTools. I initially started working with PanoTools—a mature stitching library available on Windows*, Mac OS*, and Linux*. It offers many advanced features and consistent quality. The second library I looked at is OpenCV. OpenCV is a very large project consisting of many different image manipulation libraries and has a massive user base. It is available for Windows, Mac OS, Linux, Android*, and iOS*. Both of these libraries come with sample stitching applications. The sample application with PanoTools completed our workload in 1:44. The sample application with OpenCV completed in 2:16. Although PanoTools was initially faster, we chose to use the OpenCV sample as our starting point due to its large user base and availability on mobile platforms.

Overview of Initial Application and Test Scenario

We will be using OpenCV's sample application "cpp-example-stitching_detailed" as a starting point. The application goes through the stitching pipeline, which consists of multiple distinct stages. Briefly, these stages are:

1. Import images
2. Find features
3. Pairwise matching
4. Warping images
5. Compositing
6. Blending

For testing, we used a tablet with an Intel® Atom™ quad-core SoC Z3770 with 2GB of RAM running Windows 8.1. Our workload consisted of stitching together 16 1280x720 resolution images.

Multithreading Feature Finding Using OpenMP*

Most of the stages in the pipeline consist of repeated work that is done on images that are not dependent on each other. This makes these stages good candidates for multithreading. All of these stages use a "for" loop, which makes it very easy for us to use OpenMP to parallelize these blocks of code.

The first stage we will parallelize is feature finding. First add the OpenMP compiler directive above the for loop:

```
#pragma omp parallel for
for (int i = 0; i < num_images; ++i)
```

The loop will now execute multithreaded; however, in the loop we are setting the values for the variables "full_img" and "img". This will cause a race condition and will affect our output. The easiest way to solve this problem is to convert the variables into vectors. We should take these variable declarations:

```
Mat full_img, img;
```

and change them to:

```
vector<Mat> full_img(num_images);
vector<Mat> img(num_images);
```

Now within the loop, we will change each occurrence of each variable to its new name.

`full_img` becomes `full_img[i]`

`img` becomes `img[i]`

The content loaded in full_img and img is used later within the application, so to save time we will not release memory. Remove these lines:

```
full_img.release();
img.release();
```

Then we can remove this line from the composting stage:

```
full_img = imread(img_names[img_idx]);
```

full_img is referred to again during scaling within the composting loop. We will change the variable names again:

`full_img` becomes `full_img[img_idx]`

`img` becomes `img[img_idx]`

Now the first loop is parallel. Next, we will parallelize the warping loop. First, we can add the compiler directive to make the loop parallel:

```
        #pragma omp parallel for
        for (int i = 0; i < num_images; ++i)
```

This is all that is needed to make the loop parallel; however, we can optimize this section a bit more. There is a second for loop directly after the first one. We can move the work from it into the first loop to reduce the number of threads launched. Move this line into the first for loop:

```
        images_warped[i].convertTo(images_warped_f[i], CV_32F);
```

We must also move the variable definition for images_warped_f to above the first for loop:

```
vector<Mat> images_warped_f(num_images);
```

Now we can parallelize the composting loop. Add the compiler directive in front of the for loop:

```
        #pragma omp parallel for
        for (int img_idx = 0; img_idx < num_images; ++img_idx)
```

Now the third loop is parallelized. After these changes we were able to run our workload in 2:08, an 8 second decrease.

Optimizing Pairwise Matching Algorithm

Pairwise feature matching is implemented in such a way that it matches each image with every other image and results in O(n^2) scaling. This is unnecessary if we know the order that our images go in. We should be able to rewrite the algorithm so that each image is only compared against the adjacent images sequentially.

We can do this by changing this block:

```
        vector<MatchesInfo> pairwise_matches;
        BestOf2NearestMatcher matcher(try_gpu, match_conf);
        matcher(features, pairwise_matches);
        matcher.collectGarbage();
```

to this:

```
        vector<MatchesInfo> pairwise_matches;
        BestOf2NearestMatcher matcher(try_gpu, match_conf);
        Mat matchMask(features.size(),features.size(),CV_8U,Scalar(0));

                for (int i = 0; i < num_images -1; ++i)
                {
                        matchMask.at<char>(i,i+1) =1;
                }
                matcher(features, pairwise_matches,matchMask);
                matcher.collectGarbage();
```

This change brought our execution time down to 1:54, a 14 second decrease. Note that the images must be imported in sequential order.

Optimizing Parameters

There are a number of options available that will change the resolution at which we match and blend our images together. Due to our improved matching algorithm, we increased our tolerance for error and can lower some of these parameters to significantly decrease the amount of work we are doing.

We changed these default parameters:

```
	double work_megapix = 0.6;
	double seam_megapix = 0.1;
	float conf_thresh = 1.f;
	string warp_type = "spherical";
	int expos_comp_type = ExposureCompensator::GAIN_BLOCKS;
	string seam_find_type = "gc_color";
```

To this:

```
	double work_megapix = 0.08;
	double seam_megapix = 0.08;
	float conf_thresh = 0.5f;
	string warp_type = "cylindrical";
	int expos_comp_type = ExposureCompensator::GAIN;
	string seam_find_type = "dp_colorgrad";
```

After changing these parameters, our workload completed in 0:22, a decrease of 1:40. This decrease mostly comes from reducing work_megapix and seam_megapix. Time is reduced because we are now doing matching and stitching on very small images. This does decrease the amount of distinguishing features that can be found and matched, but due to our improved matching algorithm, we do not need to be as precise.

Removing Unnecessary Work

Within the compositing loop, there are two blocks of code that do not need to be repeated because we are using same sized images. These are for resizing mismatched images and initializing blending. These blocks of code can be moved directly in front of the compositing loop:

```
	if (!is_compose_scale_set)
	{
		…
	}

	if (blender.empty())
	{
		…
	}
```

Note that there is one line in the warping code where we should change full_img[img_idx] to full_img[0]. By making this change, we were able to complete our workload in 0:20, a decrease of 2 seconds.

Relocation of Feature Finding

We did make one more modification, but the details of implementing this improvement depend on the context of the stitching application. In our situation, we were building an application to capture images and then stitch the images immediately after all the images were captured. If your case is similar to this, it is possible to relocate the feature finding portion of the pipeline into the image acquisition stage of your application. To do this, you should run the feature finding algorithm immediately after acquiring the image, then save the data to be ready when needed. In our experiments this removed approximately 20% from the stitching time, which in this case brings our total stitching time down to 0:16.

Logging

Logging isn't enabled initially, but it is worth noting that turning on logging results in a performance decrease. We found that by enabling logging, there was a 10% increase in stitching time. It is important to turn logging off in the final version of the application.

Conclusion

With the popularity of panorama capture applications on mobile platforms it is important to have a fast, open source way to stitch images quickly. By decreasing the stitching time we are able to provide a quality experience to the end user. All of these modifications bring the total stitching time from 2:18 to 0:16, an 8.5x performance increase. This table shows the breakdown.

Modification	**Time Decrease**
Multithreading with OpenMP*	0:08
Pairwise Matching Algorithm	0:14
Optimize Initial Parameters	1:40
Remove Unnecessary Work	0:02
Feature Finding Relocation	0:04