UCSC AM 148: GPU Programming Final Project Write Up

Colin Chen coachen@ucsc.edu

No Institute Given

Abstract. My attempt at implementing parallel stereo vision feature matching with CUDA.

1 Introduction to the Problem

1.1 The Importance of Depth Perception

An important problem in computer vision is recovering depth from an image. This is impossible with a single image but if a second image is introduced, it becomes possible. Depth is valuable for fields like robotics where it can be used to help build a map of a robot's surroundings and localize the robot in its environment. It is invaluable for autonomous vehicles and obstacle avoidance. While there are many kinds of sensors that can determine depth, there are a few features of cameras that make this system appealing.

1.2 Pros and Cons of a Camera-Based Solution

One advantage of a camera based solution is that cameras are passive sensors. This means they only observe their surrounds, they do not project their own light or sound to make a measurement. This means that adding more cameras to the system will not create interference, and in fact, can help increase the accuracy of the system in ways I will discuss later.

One disadvantage of a camera based solution is that computing depth is computationally expensive. I will discuss the intricacies of such an algorithm later, but it is difficult to make these computations in real time. This is a limitation when such a system would like to be employed in a field such as robotics or autonomous vehicles where real-time sensing is a necessity.

1.3 How Does Stereo Vision Work

To determine the depth of a pixel with two images, we must know two things. One is the relative pose of each camera to each other. In other words, we must know how one camera is positioned in relation to the other. While there are techniques where this pose is not known and is instead estimated, we must have some idea of the relative poses to continue. The second thing we need to know is which pixels in one camera match to the pixels of the other camera. In other words,

we need to know which pixels in each camera see the same point in the world. We use this to determine the difference in pixel coordinates of matching pixels. We call the difference in pixel value from one camera to the other disparity. If we know the disparity, we can reason an inverse relationship of depth to pixel disparity. In other words, if two pixels have similar pixel coordinates in both cameras (low disparity), we can reason that this pixel is relatively far away from the cameras while two pixels that have greatly different pixel coordinates (high disparity) map to a point relatively close to the cameras.

Both of these problems are not trivial but there are relatively standard approaches in the literature to create a stereo vision system. For the sake of this project, we will assume that our cameras are perfectly calibrated and all we need to determine is each pixel's disparity.

1.4 Determining Pixel Disparity

Determining pixel disparity is not a trivial task. To determine pixel disparity, for each pixel in one image, we must find a matching pixel in the other image. In other words, we must determine which pixels see the same point in space. A naive way to approach this problem might be as follows:

Algorithm 1: Naive Pixel Matching

```
Result: Write here the result
for Row in Left Image do
    for Column in Left Image do
        for Row in Right Image do
            for Column in Right Image do
                find minimum difference between pixels;
            end
        end
    end
end
```

This algorithm takes each pixel in the left image, then searches the entire right image for a good match. Not only is algorithm dreadfully slow, clocking it at $O(n^2m^2)$, with images of n by m pixels, it is prone to inaccuracy do to noise.

We can improve on this algorithm by employing the epipolar constraint.

1.5 The Epipolar Constraint

To improve our matching algorithm, we can employ the epiploar constraint. Epiploar Geometry deals with relating two cameras to each other. The epipolar constraint determines that given a pixel in one image, we can determine a line of pixels in the other image that might be matches for the pixel. Not only does this reduces our search space for each pixel from the entire other image to a single row of pixels in the other image, it determines that pixels NOT on this line can NOT be matches for our original pixel, even if they share the exact same color. Our new algorithm looks something like this.

Algorithm 2: Pixel Matching with the Epipolar Constraint

```
Result: Write here the result
for Row in Left Image do
    for Column in Left Image do
        for Pixel in the corresponding epiploar line do
            find the minimum difference between pixels;
        end
    end
end
```

This algorithm has improved our run time from $O(n^2m^2)$ to $O(n^2m)$ and increased the accuracy of our matching algorithm by removing pixels that are physically impossible to match with the original pixel. Determining epipolar lines as well as rectifying the images so that corresponding epipolar lines are parallel is not trivial. For this project, I will assume that images have already been rectified and focus on matching for the simplest case where corresponding rows in each image are corresponding epipolar lines. Despite this simplification, there still exists a problem of noise.

1.6 Addressing Pixel Noise

To determine if a pixel is a good match, we compare the pixels' color or brightness. We assume that points in the world will be seen the same way from both cameras. This is problematic for a few reasons. Different materials may look different from different viewing angles which will change the way pixels read values from the world. Non-uniform lighting conditions may also cause the same point to appear a different color between the two cameras. Noise from the way the camera reads each point can also cause the same point to appear different.

One simple way to address this kind of noise is instead of searching for a single pixel, to search for a group of pixels. We can then compare two groups of pixels to make a more reasonable judgement on whether the pixels are a good match. One way to compare these pixels is by taking the Sum of Square Difference (SSD). This algorithm determines the difference of each pixel in the group, determines the square of each difference, and then takes the sum of those squares.

Our final algorithm looks like this:

Algorithm 3: Reduced Noise Pixel Matching with the Epipolar Constraint

```
Result: Write here the result
for Group of pixels in Row in Left Image do
    for Group of pixels in Column in Left Image do
        for Group of pixels in the corresponding epiploar line do
            find the minimum SSD between the two groups;
        end
    end
end
```

This algorithm still has a run time of $O(n^2m)$ and achieves better accuracy than previous iteration by reducing the impact of pixel noise. However, if ran sequentially as outline, this algorithm cannot be performed in real time especially with large images. To attempt to bring this algorithm to real time, I attempted to implement a parallelized version of this algorithm in CUDA.

2 My Attempted Solution

This algorithm embarrassingly parallel, that is each pixel disparity can be found independently without waiting for other disparities to be found. There are two steps I took to attempt to parallelize this algorithm although there are certainly other ways to improve this algorithm, and implementation which I will discuss later.

The first step is to calculate the pixels disparities between one block on the left image and the corresponding row in the right image in parallel. In the sequential algorithm, a single block must search through an entire row of potential matches in the other image to find the best match. This results in a single match taking $O(n)$ which must be performed for every single pixel in one image. While it is advantageous to be able to keep rolling minimum, the second step I attempted make this approach still slower.

The second step I took was to reduce the list SSDs to find the minimum value which is the best match. Using a parallelized reduce function nets us a single match at $O(n/\log(n))$ which is faster than than the original algorithm. This brings our total algorithm's run time to $O(n^2m/\log(n))$ which still isn't great but is faster nonetheless than the sequential approach.

2.1 Problems I Encountered

The main problem I encountered was working with the CImg library to read and create images. I ran into difficulty learning how CImg stores and writes to images which prevented me from implementing a sequential or parallelized version that created an accurate disparity map between the two images. I am confident that with more time, I could solve this roadblock and complete both the sequential parallelized versions of my stereo matching algorithm.

2.2 My Findings

As mentioned earlier, I was unable to produce any reasonable disparity images. Therefore, I cannot speak to the accuracy of this algorithm. The sequential and parallel algorithms may not actually produce identical disparity images because of the way the minimum SSD is found it each algorithm in the event that a identical match is found. In the sequential algorithm, the left-most identical block in the right image will be the disparity whereas in the parallel algorithm, the leftmost identical one is not guaranteed to be the selected disparity.

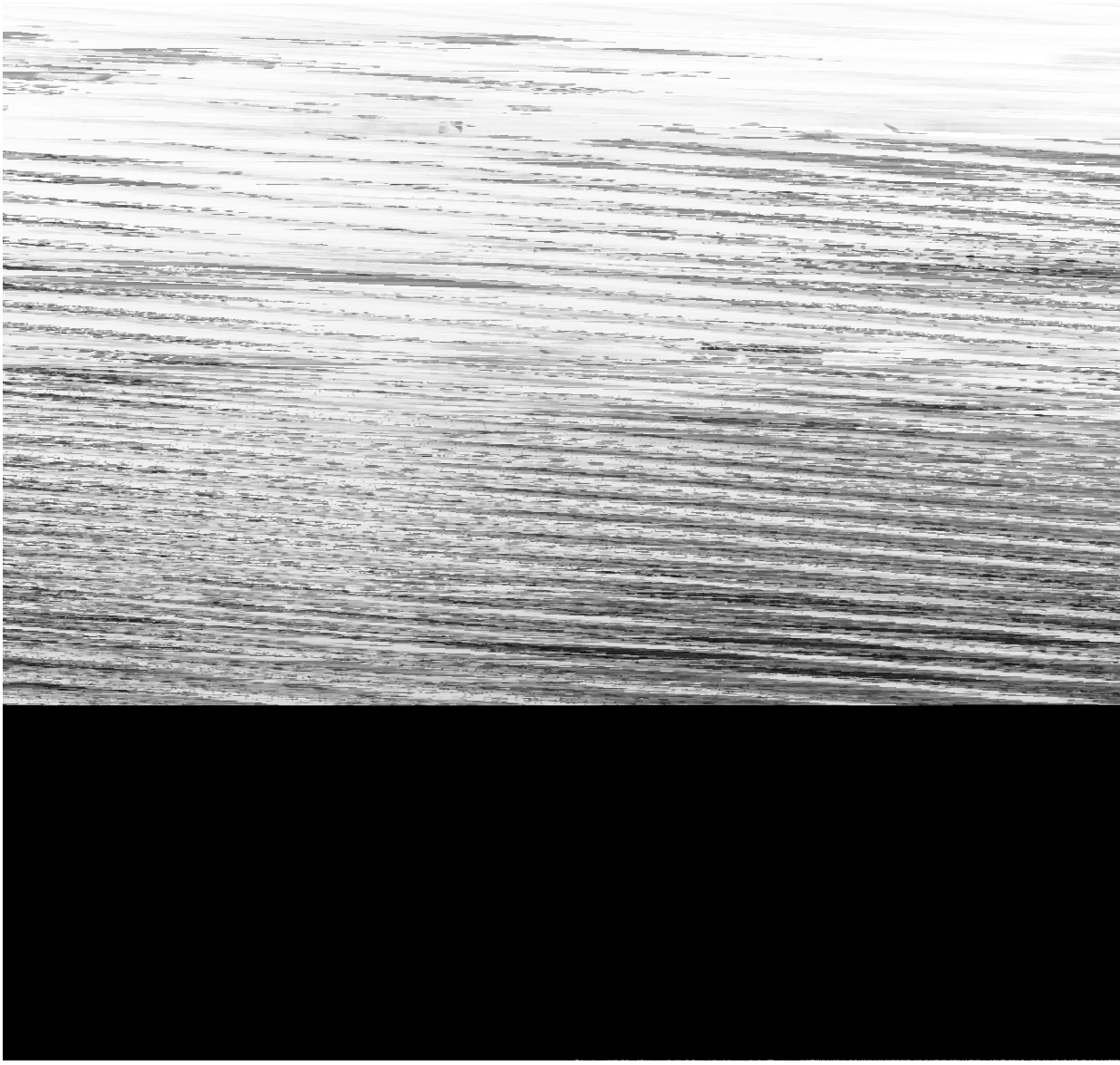
To ensure all the assumptions I made previously were met (cameras are calibrated, images are rectified, epipolar lines are mutually parallel), I used images from the Middlebury Stereo Vision 2014 Dataset [1]. These images were taken with precise calibration and labeled with a projected light patter to automatically determine depth. The image I used were 2988 by 2008 pixels and I used a matching block size of 64, which is fairly large matching block size. Typically, a larger matching block is more resilient to noise by the final disparity image is less precise. I cannot yet speak on this implementations accuracy.

To make the SSD calculation easier, I loaded the images as grayscale using Python and OpenCV.

In terms of run time, I measured the sequential algorithm at 42.12 seconds. I was unable to finish implementing the parallel algorithm to time it. I predict it would not have been real time but would have been several seconds faster than the sequential algorithm.

Below are two example images I tried to match along with the garbage disparity image.


RALEIGH



3 Conclusion

Even though I was unable to finish this project, I enjoyed working on it a lot. I will definitely be working on this more in the future and look forward to completing this project.

References

1. http://vision.middlebury.edu/stereo/data/scenes2014/