15-418 Project Proposal: Parallel Edge Detector

Junye Chen (junyec), Zhan Dong (zhand)

1 URL

https://patrickzhandong.github.io/ParallelEdgeDetector/

2 Summary

For the final project, we want to write a parallel program for edge detection, a very important tool in the field of image processing and computer vision. We will use CUDA (possibly OpenMP), and try to have a good speedup against the benchmark in OpenCV.

3 Background

Edge detection is a very important topic in the field of computer vision and image processing. In many cases, we want to ignore the unnecessary features in the figure, and focus on the main features and changes of the image. In some other challenges like object classification, the outer edges of the object sometimes have greater importance over other features like colors and brightness. Edge detection is a popular tool that is often used to preprocess the image, that ideally can omit most of the features of the original graph, and label out the edges in the original graph.
In this project, we will mainly focus on Canny's edge detector, an algorithm proposed by John F. Canny in 1986. The algorithm is compute-intensive, and has multiple stages. A brief description of the algorithm is as follows: (cited from Wikipedia)

1. Apply Gaussian filter to smooth the image in order to remove the noise

2. Find the intensity gradients of the image

3. Apply non-maximum suppression to get rid of spurious response to edge detection

4. Apply double threshold to determine potential edges

5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

From the above description, we observe that the algorithm requires a lot of computation, and therefore have the potential to be accelerated through parallelism. We believe that this algorithm has a large potential for different levels of parallelism. A very important reason for this is that a large proportion of the computation involved in this algorithm is both intensive and independent. For example, the calculation of the intensity gradients is independent from other intensity gradients, and therefore can be calculated in a parallel way. Therefore, we expect a good speedup through using the parallelization strategies we have learnt in this course. Besides, since the information we need to determine if a pixel belong to an edge normally comes from nearby pixels, if we can find a good way to assign work to threads, we expect a relatively low overall communication cost.

4 Challenges

Some of the challenges we might face in this project are:

4.1 Cache Performance

In our computation, we will encounter memory access patterns that do not have good temporal localities. For example, if we store the image in row-major format, we might encounter severe cache misses during the calculation of some of the the numbers which require access to data in different columns. To alleviate these cache misses, we might come to a better way to store the data, or have some other approaches to improve cache hit rate.

4.2 Communication and Shared Space

Since we will be using CUDA for part of the parallelization, we also need to consider the communication. We observe from our previous assignment that a high communication cost will severely worsen the overall performance of the program. Therefore, we should come to good ways to do the communication, so that fewer elements need to be communicated. We should also think about what to be stored in the global space, and what should be stored in the shared space. A clever design of information saved in the shared space might also improve the performance.

4.3 Synchronization

Another possible challenge we might face is synchronization. Since the algorithm we are using is a multi-stage algorithm, we need to make sure that the partial result after each step is complete and correct. There also might be some synchronization required in each step, and since frequent synchronization usually results in bad overall performance, we need to design the algorithm so that synchronizations are only done when necessary.

5 Goals

5.1 PLAN TO ACHIEVE

We want to follow the Canny's edge detection algorithm, which mostly uses intensity gradient in the image to detect edges. We are going to follow the algorithm detailed in this paper (https://www.cse.unr.edu/ bebis/CS791E/Notes/EdgeDetection.pdf), and we will look for any other tutorials for clarifications. We have looked for some existing libraries of edge detection, and we decide to use OpenCV's implementation of edge detection as a benchmark for both accuracy and efficiency. We want to finish a functional sequential version and check correctness against the OpenCV implementation by the checkpoint. We will start parallelzing the algorithm with various techniques, such as OpenMP, CUDA. We expect to achieve significant amount of speedup against our sequential implementation first, because we should observe an obvious speedup when we parallelize the algorithm on GPU. Then we are going to compare our implementation to OpenCV's implementation. Since OpenCV has the option of choosing whether CUDA is used, we are going to compare both our CPU and GPU implementation against OpenCV's, and optimize our solution afterwards.

5.2 HOPE TO ACHIEVE

If we achieve good speedup well before deadline, we are going to explore anti-blurring, which is the restoration process of some blurred photos. We find this article very helpful in explaining the algorithm (http://yuzhikov.com/articles/BlurredImagesRestoration1.htm). We find Matlab has de-blurring methods in its library, and we will use that as our benchmark.

6 Demo

During poster session, we are going to demonstrate that our implementation correctly detects edges in some example images. We are also going to show OpenCV's result on these images, to show that our implementation gets the similar result. We are going to introduce the framework and techniques in our parallelization and show the speedup graph of our parallel implementation against our sequential implementation. We are also going to show the efficiency comparison between our implementation and OpenCV's implementation. We will possibly explain the flaws in our implementation if it doesn't achieve the speedup as we expect.

7 Platform Choice

Since OpenCV has C++ implementations, we are going to use C++ for our project so that we minimize any lurking variables that affect speedup. We are also going to use CUDA and OpenMP for good parallelization. CUDA is a very common choice for achieving good speedup when manipulating images. In HW2, CUDA is able to speed up the parallel rendering very well. Although the memory copying in CUDA is very costly, if we can finish most of the computation inside CUDA kernel, we should take the most advantage of CUDA. CUDA also offers shared memory, which is definitely a good way to improve locality when computing in large arrays.

8 Schedule

By November 8 Understand the algorithm of edge detection, start to write code for sequential implementation

By November 20 Finish sequential implementation, and already start to parallelize the algorithm.

By November 27 Finish the main skeleton of parallel implementation and should pass correctness checks. Compare the speedup against sequential benchmark and OpenCV benchmark (CPU and GPU).

By December 5 Wrap up the implementation. If the program achieves good speedup, start to explore anti-blurring. If not, analyze any potential problems (uneven distribution of work among threads, too much overhead of memory copying), and try to improve. Start the writeup

By December 12 Clean up code, and finish the writeup.

9 Reference

[**1**] https://en.wikipedia.org/wiki/Canny_edge_detector

[**2**] https://www.cse.unr.edu/ bebis/CS791E/Notes/EdgeDetection.pdf

[**3**] https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html