Brian Jay
15-418
Final Project Report

Summary

I will implement the Viola-Jones facial detection algorithm using several different technologies, including OpenMP, pthreads, and CUDA on an NVIDIA GeForce GTX 1080 found in the Gates machines. I will compare each implementation to a baseline C++ implementation and OpenCV's provided implementation that both run on a standard CPU.

Background

GPU's are very valuable in image processing problems because of the inherent parallelism found in many common image processing problems. Viola Jones Object Detection works by finding Haar-like features corresponding a target image. Haar-like Features are composed of rectangular regions aligned in patterns to find edges, lines, and other primitive features. When these features are convolved over an image, they generate a large response over their represented feature.

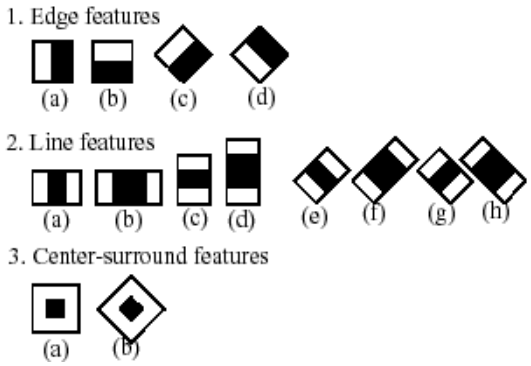


Figure 1: Examples of Haar-like Features

The algorithm works in 2 parts. First is a training step to produce a cascade classifier of features. The cascade is trained with two sets of data: a positive set containing the images with the target object for detection and a negative set containing images without the target object for detection. The cascade will determine which features are common in the positive set. For example, a positive set of human faces should have a distinctive pattern of edge and line responses corresponding to the eyes, nose, and forehead.

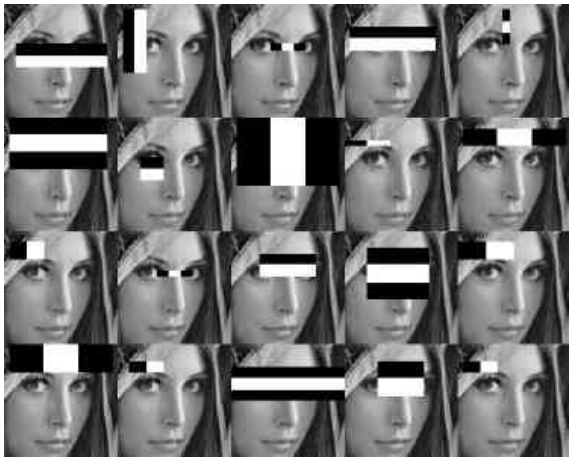


Figure 2: Viola Jones can be used for facial detection. Eyes, noses, and foreheads can be identified by a series of Haar-like Features.

Upon conclusion of training, the detector will determine a set of features able to differentiate the positive and negative training sets. Since training is a one-time process and cascades for detected faces have been published in libraries like OpenCV, I will use OpenCV's provided data and instead focus my work on parallelizing the image processing across the resources of a GPU.

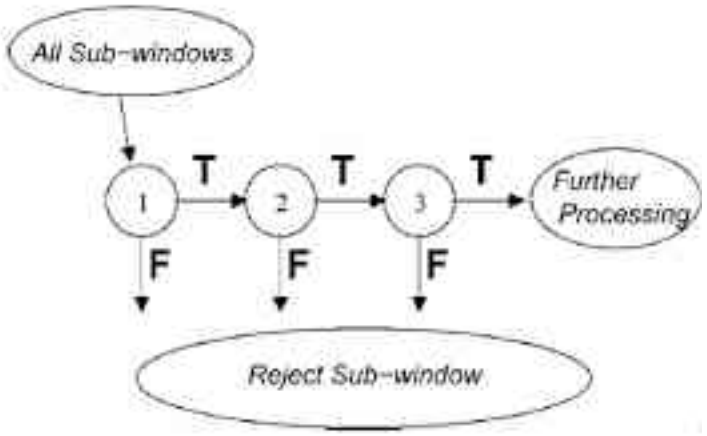


Figure 3: A cascade classifier containing three stages. Each stage will search regions of an image for one or more features. If the region does not contain appropriate feature responses, the possibility of the region containing the target object is rejected.

The image to be processed is then divided up into regions, starting at size 20x20 and increasing in size. Each region of interest will be tested by the cascade classifier. If the region of interest contains the features to pass through the classifier, the Viola Jones Detector will draw a box around the region of interest to specify an area containing the target object. The following pseudo-code gives a high level overview of the process:

```
For size in scale:
      For each region in the image:
            For each stage in classifier:
                  For each feature in the stage:
                        Generate response
                        Accumulate response
                  If response < threshold:
                        Return Fail
            Return Pass
```

Approach

Using OpenCV as a reference, I set out to implement a version of the Viola-Jones Facial Detection Framework using several different platforms and technologies. I started with a baseline implementation in C++, using OpenCV's basic cvHaarCascadeClassifier and Mat data structures to store the cascade and image, respectively.

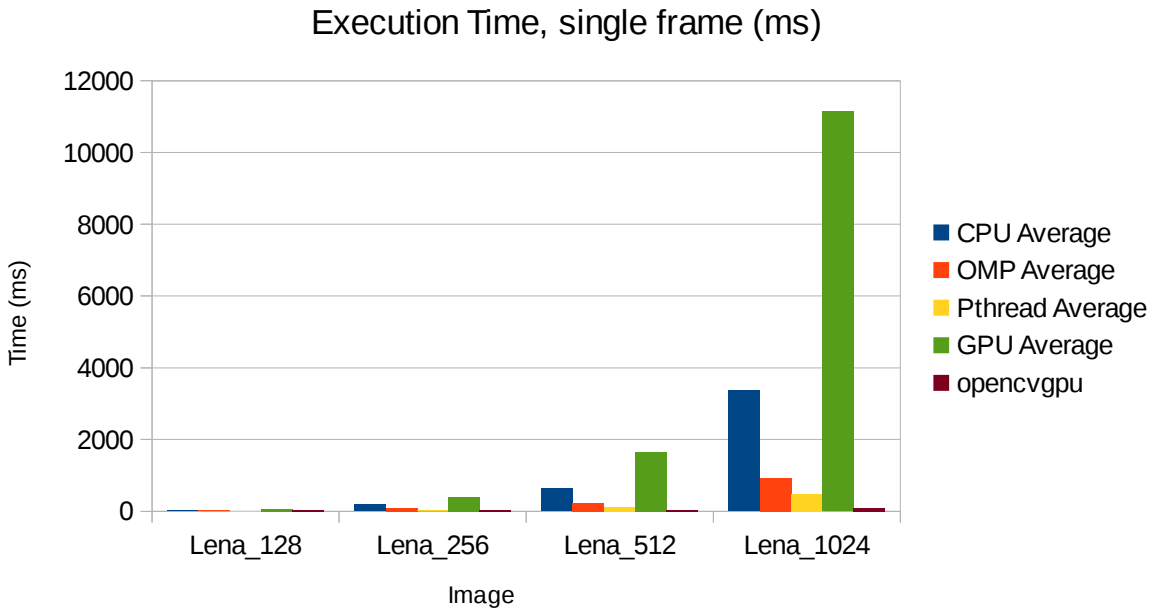
Image processing is inherently very parallel. Most image processing kernels rely on pixels in a localized region. As such, regions of the image can be easily mapped to threads and cores. I decided on using OpenMP, pthreads, and Cuda to develop different versions of the algorithm. The OpenMP implementation simply uses a #pragma omp parallel for schedule(dynamic) outside of the outermost loop. Given the nature of our workload, the dynamic scheduler is well worth any extra overhead it may incur. Very few regions of an image will probably contain a face, so the vast majority of sub windows will fail in the first few stages of the cascade. For those few sub windows that will process the entire cascade, it helps to utilize other threads available to take on some of the workload. The pthread implementation interleaves the assignment of regions to each of the 16 execution contexts on the Gates CPUs. I also experimented with using a chunked assignment of regions to threads, however I found no appreciable difference between the implementations. The CUDA implementation utilizes one thread per region of the image to process, with block sizes of 32x32 threads.

Results

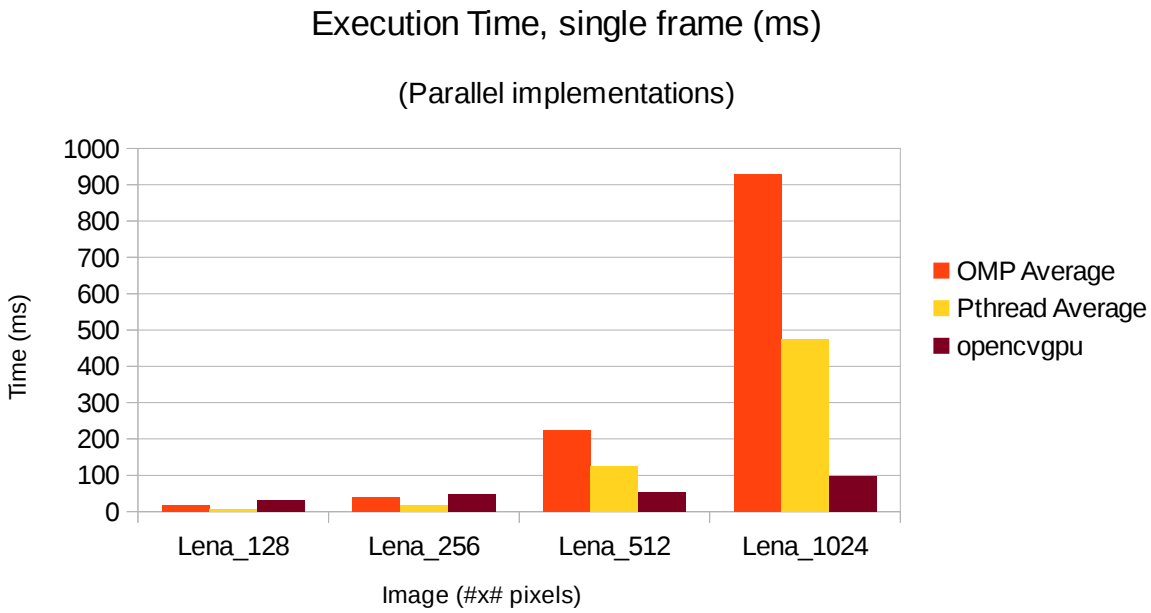
I ran each algorithm across a set of 8 pictures. 5 were scaled sizes of the classic Lena centerfold head shot, of sizes 128x128, 256x256, 512x512, 1024x1024, and 2048x2048. The

other three images consisted of multiple faces, and were included in the test set for correctness testing. The Viola-Jones algorithm has no problems detecting multiple faces in an image. The scaled images expose how each implementation of the algorithm performs on various image sizes, and provides insight into how each implementation scales with respect to image size.

The ultimate goal was to come up with an implementation that runs at close to 30 frames per second. Unfortunately, I was not able to implement a CUDA version that was able to meet the performance standards put forth in other research papers. However, I was able to match quite well with OpenCv's CPU based implementation of the object detection framework. In addition to using Cuda, I used OpenMP and pthreads on the GHC machines alongside a single threaded implementation to see what kind of speedup we can achieve on a traditional CPU.



My GPU implementation was not competitive with opencv's implementation due to a few key factors. My algorithm does not reuse data particularly well, namely each time a frame is processed, the integral image of that frame is calculated in full, stored back in main memory, then the cascade detection process begins. In opencv's implementation, each thread calculates the integral image for its respective sub-window, then processes the features while the integral image is still in cache, thereby exploiting the temporal locality of the integral image data. My implementation is therefore bandwidth bound and this drives down performance immensely as image size increases.



Taking a look at the CPU parallel implementations, at small image sizes the pthread implementation outperforms opencv's GPU implementation. Using 16 threads, one for each execution context on the GHC machines, the pthread code meets the hard real time requirement of 30 frames per second, achieving an average execution time of around 50 frames per second. Additionally, the OpenMP version of the algorithm performs comparably with opencv's GPU detection algorithm. Utilizing dynamic scheduling afforded around a 15% speedup over static scheduling, which is in keeping with the fact that thread divergence can seriously impact performance as discussed earlier.

At high image sizes, we can see via openCV's implementation that GPU's provide significant performance increases over traditional CPU's, but at lower resolutions and image sizes, the winner is not as clear. In cases of autonomous robots and other embedded, field deployed devices that cannot afford to power a GPU, a small image size can still be processed in real time on a CPU. As is evident by the second graph, both OpenMP and pthread implementations are marginally faster than opencv's GPU implementation. In these fields, as long as the frame to be processed is small enough, then real time processing is a feasible goal, and left as an exercise for the reader.

References

[1] - http://www.vision.caltech.edu/html-files/EE148-2005-Spring/pprs/viola04ijcv.pdf

[2] - https://pdfs.semanticscholar.org/df51/d088125c96c53bfb23c2fdfdc7977b58dc79.pdf?_ga=2.152929978.282291133.1494599203-300331682.1494599203

[3] - http://cseweb.ucsd.edu/~kastner/papers/fccm10-gpu_face_detection.pdf