Final Project Report – ParaBurst

This report outlines the results, details and findings of our final project – ParaBurst. It is an application that analyzes photos taken in a burst mode and returns the best photo based on eye state detection using highly parallel algorithms.

We managed to implement a parallel version of the Viola-Jones algorithm for (face detection) optimized to run GPU. Further, a simple algorithm was used to detect the state of eyes (open or closed) and return the best image in a burst.

Section A – Viola-Jones algorithm for Face Detection and its Parallelization

i. A brief overview of the Viola-Jones Algorithm

Given below is a diagram giving the outline of the Viola-Jones Algorithm:

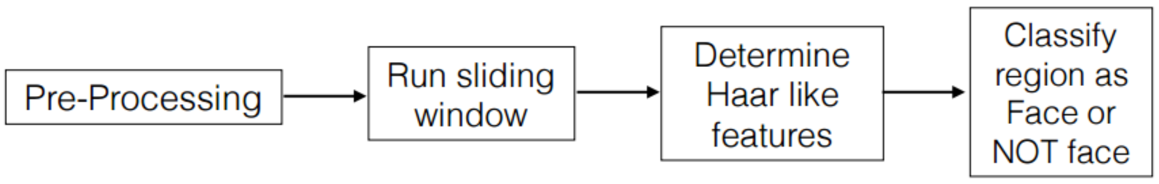
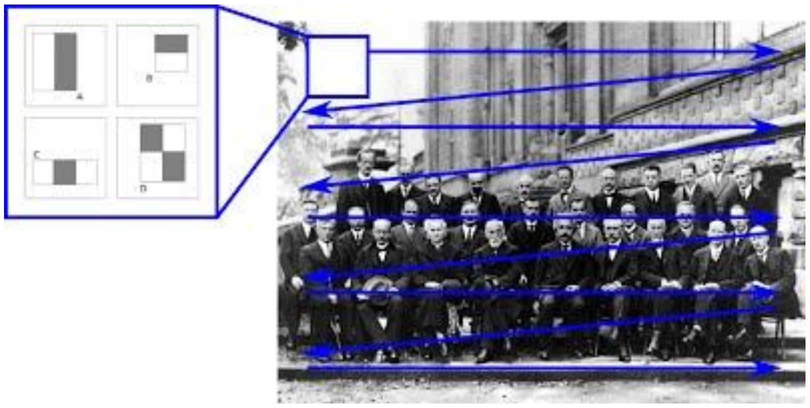

Pre-Processing
Run sliding window
Determine Haar like features
Classify region as Face or NOT face

Figure 1: Outline of the Viola Jones Algorithm

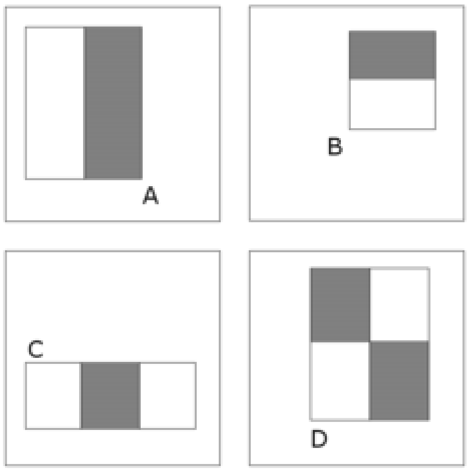
- **Pre-processing**: The preprocessing phase sets up an image for running the classifiers. This stage includes parsing the image, converting it to Grayscale and finally down-sampling the image for face detection.

- **Running the Sliding window:** The sliding window is a fixed window of size 24x24 that traverses the image pixel by pixel and runs certain filters on the image area it covers. These filters reveal some features which can later be used to classify the region as a face or non-face.



<u>Figure 2:</u> The sliding window

- **Haar Filters:** The sliding window applies multiple filters called to the image region it bounds. These filters are called Haar Filters and reveal horizontal and vertical features in the image. A Haar-like feature considers adjacent rectangular regions at a specific location in a detection window, sums up the pixel intensities in each region and calculates the difference between these sums. This difference is then used to categorize subsections of an image.



<u>Figure 3:</u> A few Haar Filters

- **Cascade Classifier:** A bunch of Haar-like features are compiled into a single large classifier, which is applied to each sub-window. At each step, the algorithm classifies a sub-window using a specific feature. If the sub-window meets the criteria then the algorithm proceeds with applying other features, otherwise, the sub-window is rejected. Intuitively, at each stage, the algorithm compares the values against tested classifiers and asks *Does this feature resemble a facial feature*? Earlier stages are easier whereas later stages extremely computationally intensive and more difficult to meet the criteria.

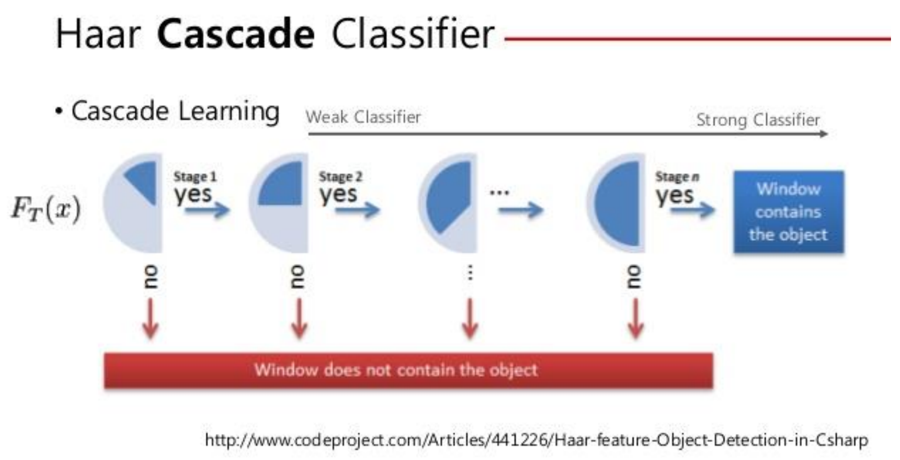

Haar Cascade Classifier
• Cascade Learning
Weak Classifier
Strong Classifier
$F_T(x)$
Stage 1
yes
Stage 2
yes
...
Stage n
yes
Window contains the object
no
no
no
Window does not contain the object
http://www.codeproject.com/Articles/441226/Haar-feature-Object-Detection-in-Csharp

Figure 4: The Haar Cascade Classifier

ii. Approaches to Viola Jones Alorithm

a) Sequential Approach

The sequential approach involved writing the Viola-Jones Face detection algorithm. In order to analyze the best-case performance of the program on a single CPU, we made a few optimizations without using parallel primitives. These optimizations proved to be helpful in the later stages of GPU-based development as well.

- Instead of using the standard Haar classifier that consists of a large number of cascade classifications, we used a small yet tested subset of the classifiers. This ensured that for any given subwindow, our Haar cascade classifier involved a much smaller hierarchy.

- An analysis of our CPU code showed that a large percentage of overall time was spent on computing sum of pixel intensities for each sub window. After some research, we found a better way is to compute the integral image (Described later).

b) Naïve Parallization using OpenMP

We added OpenMP primitives to speed up our program but the performance gain was not a lot. The primary reason is the dependencies between stages in a cascade classifier - the algorithm proceeds to the second stage if and only if the first classifier's result is a success. Furthermore, the computations involved at each stage are not same and in fact, they differ by a high margin between the first few stages and the stages applied towards the later stages. We then decided that CUDA might be the best approach.

c) Using CUDA

The algorithm comprises of computing hundreds of features over a lot of **independent** sub windows, which is characteristic of a SIMD application. In order to implement SIMD and utilize the GPU in parallel, used NVIDIA's CUDA library.

1. The simplest approach just involved parallelizing over all the stages of the cascade classifier. In the sequential version, the cascade classifier involved a dependent execution i.e. the second stage was applied only if the first was a success. On CUDA, we parallelized over all stages in a sub window. While this reduced dependencies, this caused a lot of unnecessary processing. In most cases, later stages were not even required as those regions were rejected by the first few stages. The face in an image is only a small part and by computing all the stages, we did not achieve the desired speedups. Furthermore, this only parallelized one part of the algorithm.

2. The second and **our current approach** parallelized over the sub windows. As a recap, the Viola-Jones algorithm has a moving sub window over the entire image, and all the classifications and computations are done within a given sub window at a specific time. The sequential version involved processing one sub window at a time, and then moving on to the next.

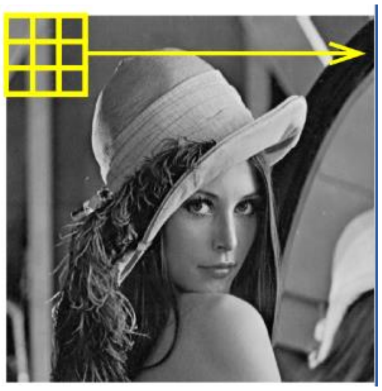


Figure 5: The Sliding Window

We divide the image into equal rectangular regions and each region gets mapped to a CUDA thread block, and each thread is responsible for one sub window. Note that we check the image size before dividing into regions since an extremely scaled down image doesn't warrant performing on further several small regions. After testing and analysis, we determined that 128 threads/block provides the optimum execution times. Since the Viola-Jones algorithm involves scaling the image down by a predetermined factor, the number of thread blocks required grow less as the image undergoes more scaling. Due to scaling, we also launch our CUDA kernel for each downsize of the image. Our findings showed that the smallest down sampled images take a very short span of time and do not affect overall time. Therefore, CUDA effectively helps us parallelize over the whole image using sub windows.

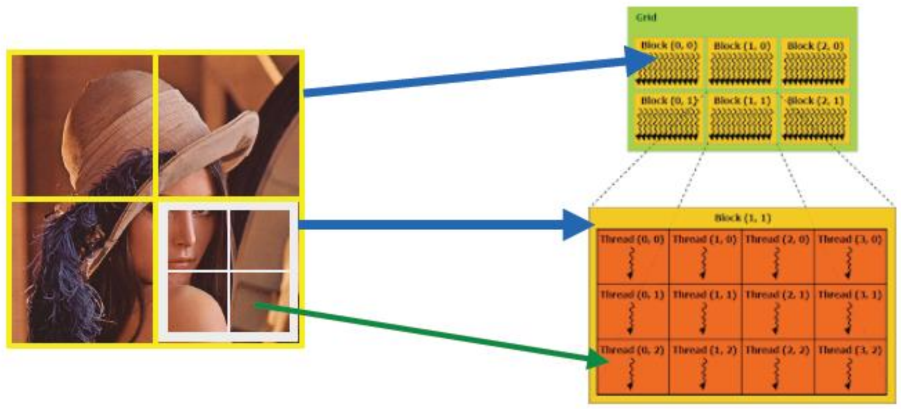


Figure 6: Sub-window Parallelization

This approach gave us a high speedup compared to previous implementations yet we found a few bottlenecks. Primarily, all the threads relied on Haar-like features, which were stored on the global memory. In other words, all threads compare their computed data against the Haar-like features to determine whether the feature corresponds to a face or not. Since all threads at least access the first feature from the global memory (some go on to access a lot more features if they are in the region comprising the face), the latency is quite high. To hide latency, we put the Haar-like features in the shared memory (100x lower memory latency[1]). As another memory optimization, we used memory coalescing to send the Haar-like features from global memory to shared memory by exploiting the fact that all features are stored in contiguous memory locations.
This allows us to achieve the optimum global memory bandwidth since all threads in a thread block are trying to load the Haar-like feature from consecutive memory locations in the global memory[2].

[1] https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc
[2] http://stackoverflow.com/questions/5041328/cuda-coalesced-memory

d) Using OpenCV's GPU library

We implemented a final version to compare performace using OpenCV's GPU library with inbuilt CUDA bindings. This was only used as a reference implementation.

iii. A comparison of Results

We finally tested and compared our results for the various approaches we used.

Here is a quick comparison of the runtimes of the 3 versions on the same image:

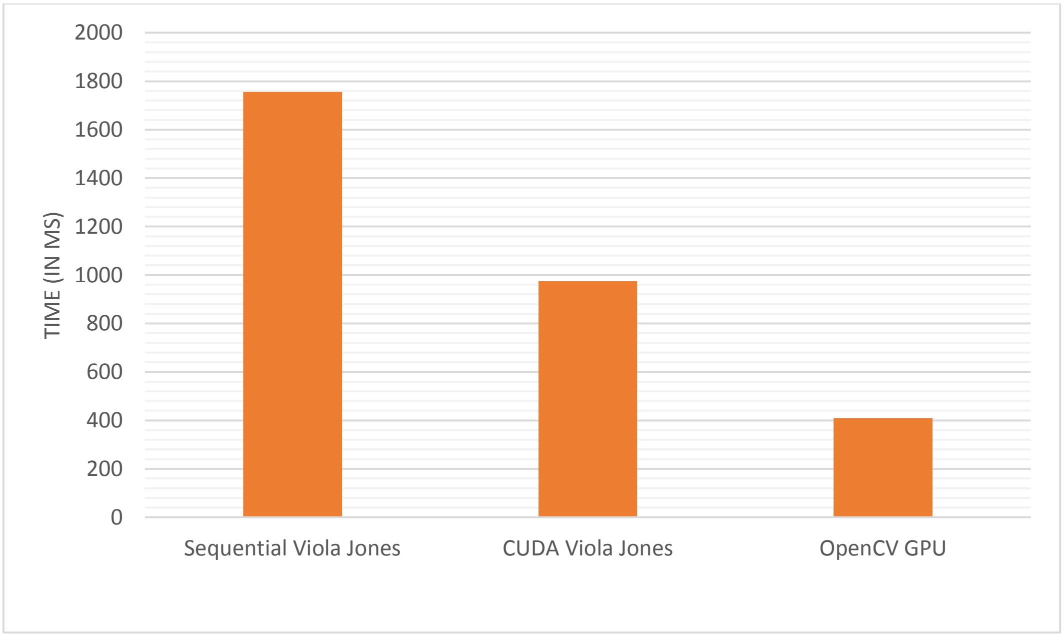


Figure 7: A comparison of the 3 implemented versions of Viola Jones

We observed a speedup of **1.8x** from the sequential version and approximately a **3.8x** speedup using the OpenCV GPU libraries.

Given below is a graph of the performance of the 3 algorithms on images with different number of Faces. As the number of faces increases the computation intensity also increases as we have to go through all stages of the classifier or multiple face regions.

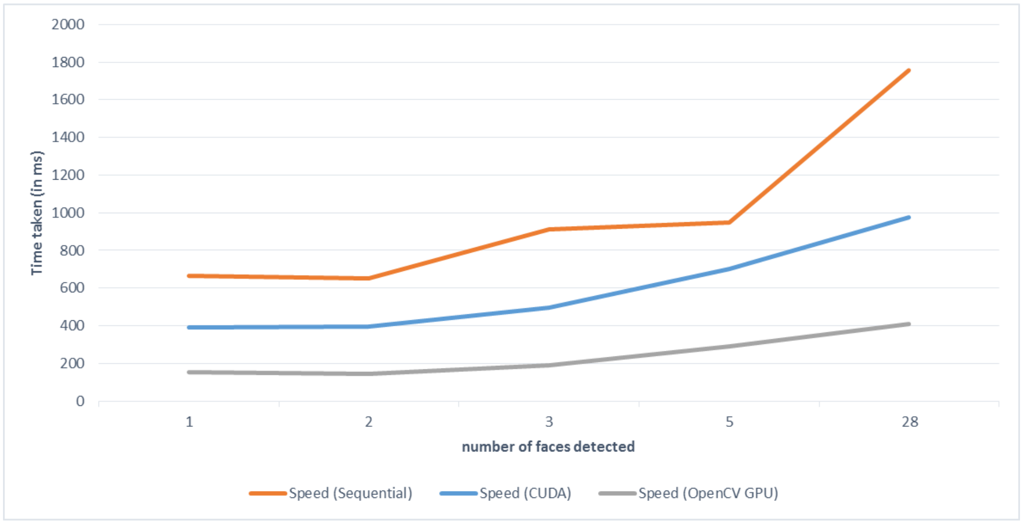


Figure 8: A comparison of the 3 implemented versions of Viola Jones

Even after trying multiple parallelization techniques, we could not beat the OpenCV GPU implementation. This is because:

- The openCV GPU library is highly optimized and performs the preprocessing stages much faster than our versions. These include – computing the integral image and downsampling the images.

- The OpenCV GPU library manages to change 2D representations of the image to one dimensional representations for GPU much faster and in a more optimized way than we do. It uses better indexing techniques than the one we used.

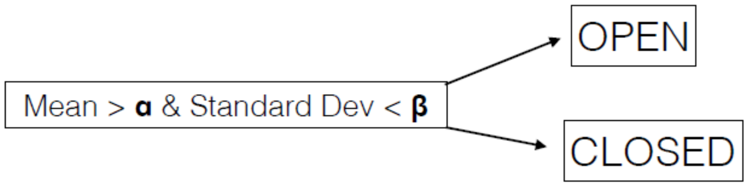
Another thing to note is that we only used a subset of haar classifier in our implementations and not the entire set. Hence the accuracy of the OpenCV GPU implementation was also much higher i.e. around 72% compared to 51% of our implementation when run on selected images.

<u>Section B – Open Eye State Detection</u>

i. The Algorithm

Once we managed to get the coordinates for Face and eyes from the Viola Jones algorithm, we used a simplistic approach to detect open eyes on multiple images. This approach was inspired a research paper[3].

The algorithm is simple for each eye region detected in the image, it calculates the mean and standard deviation of normalized pixel values. It then compares these values to certain threshold values and classifies it as "open" or "closed".



<u>Figure 9:</u> Eye State Detection Algorithm

We find the eye coordinates for one image in the burst.

Then we use these eye region co-ordinates for all images in the burst and classify eyes as open or closed. Finally the image with the most number of open eyes is returned as the BEST image.

ii. Attempts to Parallelize Eye State Detection

- **Naïve Parallelization**: Initially we attempted a simple parallelization approach using parallel for loops with Pragma OMP. However we noticed a significant slowdown of around 200-300ms when using parallized for loops attempting both dynamic and static scheduling. After further testing and reasoning we realized that we have at most 8-12 eye regions detected per image and computing the mean and standard deviation is a simple computation for eye regions not spanning more than 50x50 image regions. Hence the scheduling overhead on the CPU was compromising performance!
  Hence we decided to stick to the simplistic sequential approach.

[3] http://www.ajer.org/papers/v4(01)/F0401043048.pdf

- **Using the GPU:** Having already used GPU to speed up viola jones algorithm we considered parallelization using the GPU and cuda. However since the naïve parallel version showed no improvement, we reasoned that the overhead of communication with the GPU might once again compromise performance. And again as the eye regions were small it made sense to stick to the CPU for such simplistic computations.

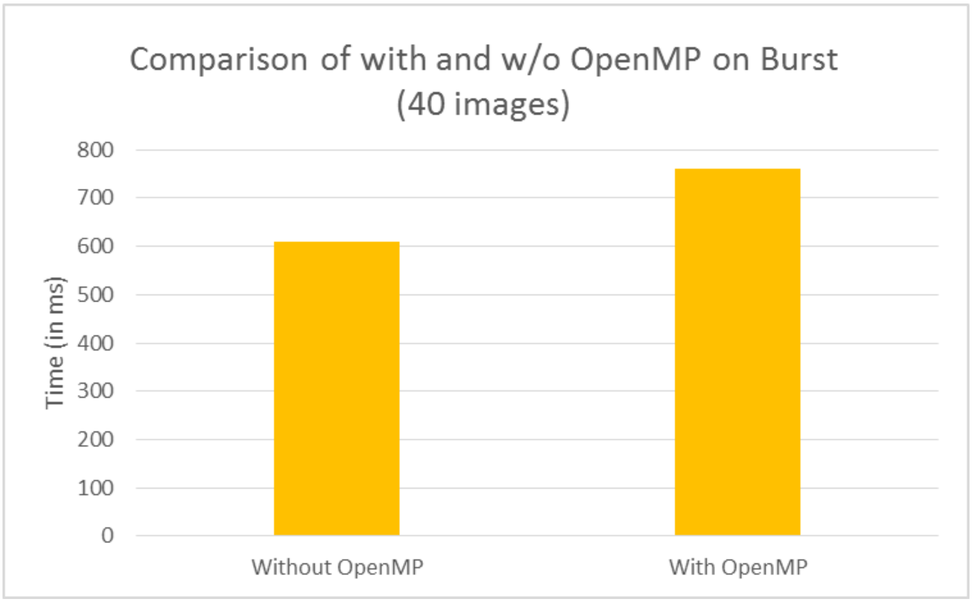


Figure 10: With v/s Without OpenMP (40 images)

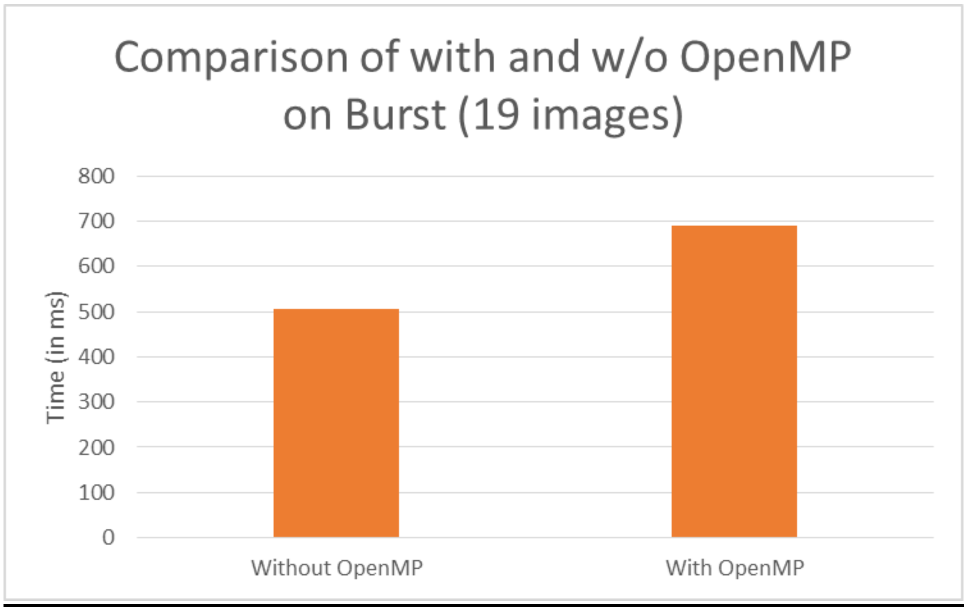


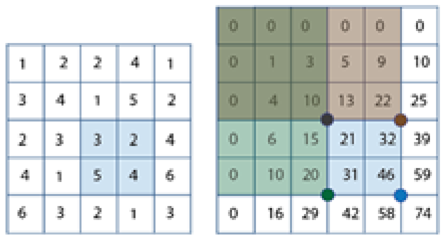
Figure 11: With v/s Without OpenMP (19 images)

Section C – Other Optimizations

Within the Viola-Jones algorithm, we had to calculate the sum of pixel values in multiple boxed regions in the image when we use Haar filters for feature detection. Using a purely iterative approach to sum each pixel value in a box of size W*H, we would need to access W*H pixel values in the image.

We know that the image's pixel values do not change during the course of the face detection algorithm. Hence it made sense to calculate an "integral" image where the value of each pixel in the integral image is the sum of all pixel values to the top and to the left of the pixel in the original image.

For instance:

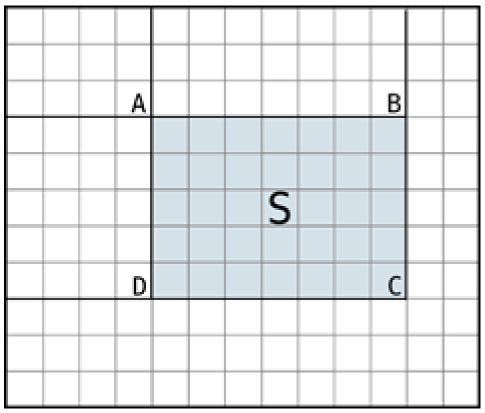


1	2	2	4	1
3	4	1	5	2
2	3	3	2	4
4	1	5	4	6
6	3	2	1	3

input image

0	0	0	0	0	0
0	1	3	5	9	10
0	4	10	13	22	25
0	6	15	21	32	39
0	10	20	31	46	59
0	16	29	42	58	74

integral image



Now the sum of the region S can be calculated by **S = D – B – C +A**, where A,B,C, D is are the integral image values at location A, B, C, D.

Thus with the integral image we need exactly 4 accesses to the integral image v/s W*H accesses to calculate the sum of a region. Computing the integral image at the beginning of the algorithm prevented memory accesses from becoming a bottleneck in our algorithm.

Section D – Results

In this section, we present a few pictures highlighting the intermediate stages of the algorithm namely face and eye detection and show timings of running the entire application

i. Face Detection Results

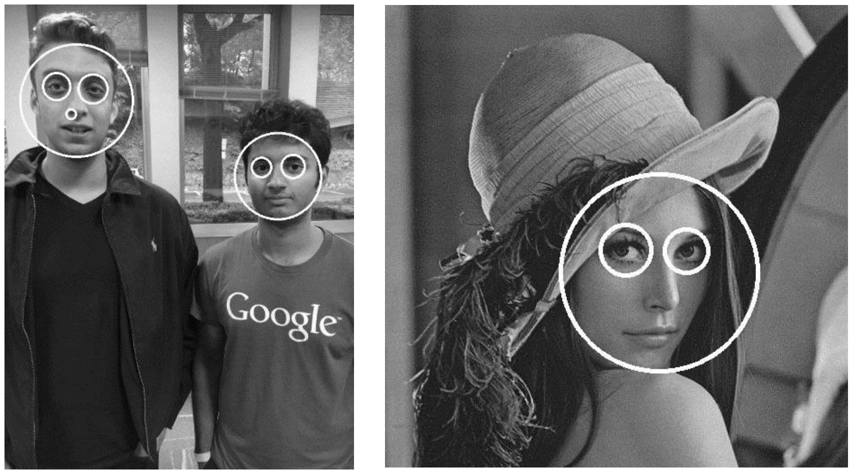


Figure 12: A few Face detection Results

ii. Full Algorithm Results with timing:

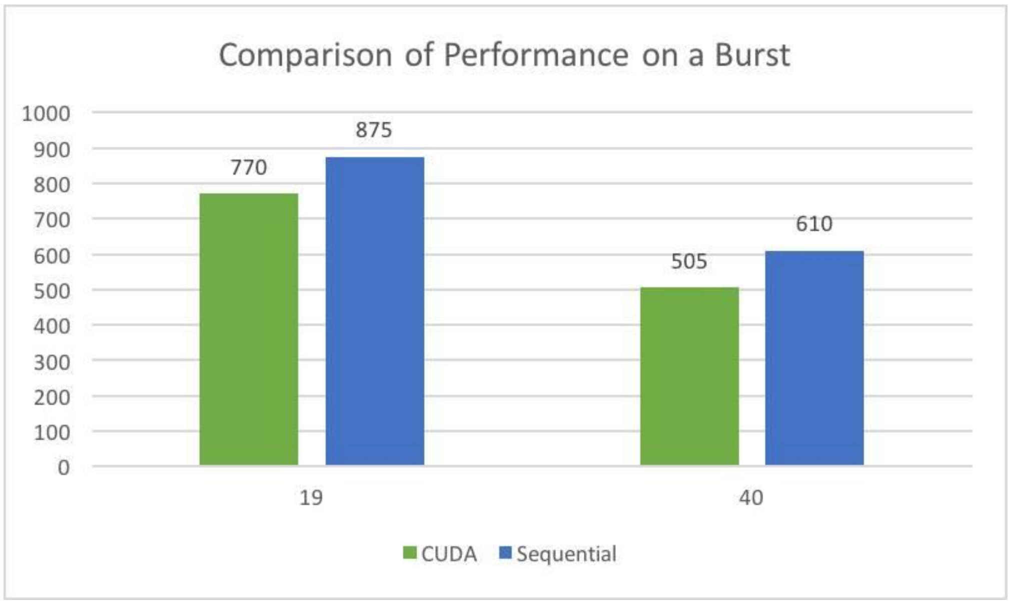


Figure 13: A final comparison (Time in ms)

Viola Jones is the main bottle neck where the CUDA version performs much better. However eye state detection is sequential. Here most of the difference in time is because of the viola jones implementations.

Section E – Limitations and Scope for Improvement

Having completed this version of Para Burst, we analyze the limitations of our approach and look at further improvements we could make.

i. Limitations:

- Our Algorithm works solely on checking eye states and not other parameters like smile or lighting conditions.
- Our approach is NOT invariant rotations. Sometimes if faces are tilted or turned the algorithm may fail to detect them. This is a result of using the Haar filters in the cascade classifier that are not rotation invariant.
- There is some load imbalance in our CUDA implementation. For instance, a thread with a sub window without any facial feature might stop much before a thread with a sub window consisting of facial features; the latter will go through more stages of the cascade classifier.

ii. Scope for improvement

- Use additional parameters like smile, blur and movement to rate images in the burst and return the best image to the user.
- We could improve speed by using a different classifier. Even during our implementation we discovered other classifiers like the linear binary classifier that is significantly faster as it has fewer classification stages but is slightly less accurate.
- Further different classifiers with filters or methods that are rotation invariant would also help us get more accurate results as it would detect faces that are turned or tilted.
- Using a more robust attempt for open eye detection which is more accurate in general compared to the approach we used just using mean and standard deviation.
- Improve load imbalance problem in CUDA implementation by devoting idle threads to other computation-intensive tasks.
- Lastly we use parameters for open eye detection based of literature values. We could further implement a machine learning algorithm that “trains” these parameters so that determining whether eyes are opened or closed becomes more accurate.
- Our final goal is to integrate ParaBurst into a mobile or web app to make it easier and more practical to use.

Bibliography

1. A Parallel Architecture for Multiple-Face Detection Technique Using AdaBoost Algorithm and Haar Cascade
2. Improvements in OpenCV’s Viola Jones Algorithm in Face Detection - Tilted Face Detection By Astha Jain, Jyoti Bharti1 and M.K.Gupta.
3. Face Detection CUDA Accelerating By Jaromír Krpec And Martin Němec.
4. A fast and accurate algorithm for eye opening or closing detection based on local maximum vertical derivative pattern.
5. A FAST AND EFFICIENT SIFT DETECTOR USING THE MOBILE GPU
   Blaine Rister, Guohui Wang, Michael Wu and Joseph R. Cavallaro