1 Summary

We implemented an optimized facial detection system using CUDA. Given the speed of our implementation, we demonstrate that a parallel approach to classifying and detecting faces is far more efficient than sequential CPU approaches.

2 Background

2.1 Key Operations and Data Structures

The Viola-Jones algorithm is a pivotal algorithm for face detection in images. Its efficiency lies in the utilization of integral images, which are key data structures in the algorithm. The integral image, also known as the summed area table, is an array of integers that represent the sum of pixel intensities in a rectangular region of the original image. This structure facilitates rapid computation of features within candidate windows.

Another key data structure is the cascade classifier. The cascade classifier is a series of stages, each consisting of multiple strong and weak classifiers. It's a data structure that efficiently filters out non-face regions in the image. Each stage of the cascade contains a set of thresholds and decision rules based on the Haar-like features.

Key operations on these data structures include the creation of the integral image and the detection of features within image windows. The integral image is efficiently computed using a single-pass algorithm, while feature detection involves evaluating rectangular regions within the integral image to identify features indicative of facial characteristics.

The cascade classifier applies a series of weak classifiers in sequence. At each stage of the cascade classifier, the algorithm evaluates multiple Haar-like features at various positions and scales within the integral image. This operation involves calculating the difference between the sums of pixel values in the light and dark regions defined by the Haar-like features. The classifier uses decision rules to compare the computed feature values to predefined thresholds and classify image regions as either faces or non-faces.

2.2 Algorithm Outline

The algorithm takes an input image containing one or more faces that need to be detected. The output of the Viola-Jones algorithm is the location and size of bounding boxes around detected faces within the input image. Each bounding box indicates the region where a face is detected.

An outline of the algorithm is below:

- Use Haar-like features to identify different parts of the face, such as edges, lines, and rectangles of varying sizes and positions.
- Calculates the integral image from the input image and uses a cascade classifier composed of multiple stages (each stage consists of a set of weak classifiers)
- Invokes the cascade classifier to quickly reject non-face regions in the image. It does this by applying a series of these weak classifiers in sequence. If a region fails any of the weak classifiers in a stage, it is quickly discarded as not being a face.
- After passing through all stages of the cascade, the remaining regions are considered potential faces. A final threshold is applied to these potential face regions to determine if they are indeed faces.

2.3 Analyzing costs

One of the most computationally demanding aspects of the Viola-Jones algorithm occurs during the feature detection step, where the algorithm evaluates numerous Haar-like features across various positions and scales within the input image. Each feature evaluation involves calculating the difference in the sums of intensities within specified rectangular regions of the image, which would typically require a considerable number of memory accesses and arithmetic operations.

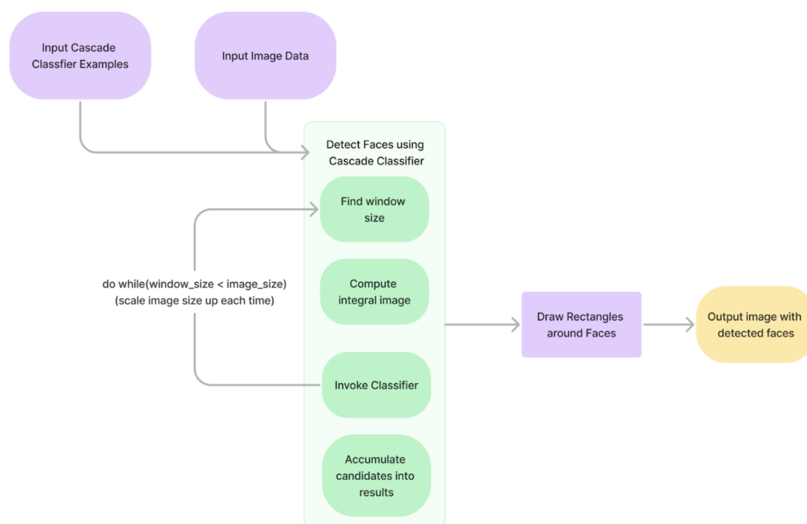
The integral image provides a precomputed representation of the image, enabling faster summation. However, even with the integral image, the large volume of feature evaluations across different positions and scales can lead to significant computational overhead.

By exploiting parallel computing architectures such as multi-core GPUs, the algorithm can distribute the workload across multiple processing units. Each processing unit can independently evaluate Haar-like features within a designated region or subset of the image, simultaneously executing computations on different parts of the input data. This would likely result in efficient utilization of computational resources and reductions in processing time.

The input image can be partitioned into smaller regions or tiles (blocks), with each region assigned to a separate processing unit. By parallelizing the cascade classifier using CUDA, we can take advantage of the parallel processing capabilities of GPUs, leading to significant improvements in detection speed and efficiency.

2.4 Breaking down the workload

The integral image needs to be computed before feature detection can occur, establishing a dependency between these steps. However, once the integral image is generated, feature detection can be parallelized across different regions of the image. This parallelism arises from the fact that each region can be processed independently, making the algorithm data-parallel in nature. Additionally, the integral image enhances locality by reducing the number of memory accesses needed for feature detection, thereby improving efficiency.



Upon closer examination of the Viola-Jones algorithm, it became evident that certain aspects of the algorithm lend themselves well to parallel processing techniques. Specifically, the evaluation of different images, such as integral images and sub-windows, presents opportunities for parallelization through block-like processing using CUDA. While the initial implementation of the algorithm was sequential due to dependencies among classifier outputs for each window size, we are exploring strategies to efficiently accumulate this data.

3 Approach

Our parallelization efforts will focus on two main areas: firstly, evaluating each window for a single integral image can be parallelized by partitioning the image into disjoint blocks, which can be processed concurrently using CUDA blocks. Secondly, parallelizing the weak classification steps of the cascade classifier, which will speed up the evaluation of facial features throughout the image by encouraging early rejection of non-candidate areas.

3.1 Technologies used

For our project, we initially implemented a sequential facial detection system using C++. We then used CUDA to facilitate parallel processing directly on the GPU. Our code defines number of blocks and threads per block based on the image frame.

We used The CUDA runtime API is used to manage device (GPU) memory, control execution, and facilitate interactions between host (CPU) and device (GPU) code. We organized our project into multiple files, each handling specific aspects of the algorithm including the cascade classifier, generation of the integral images, and haar-feature calculations. This modular approach helped us manage complexity and improve the maintainability of our

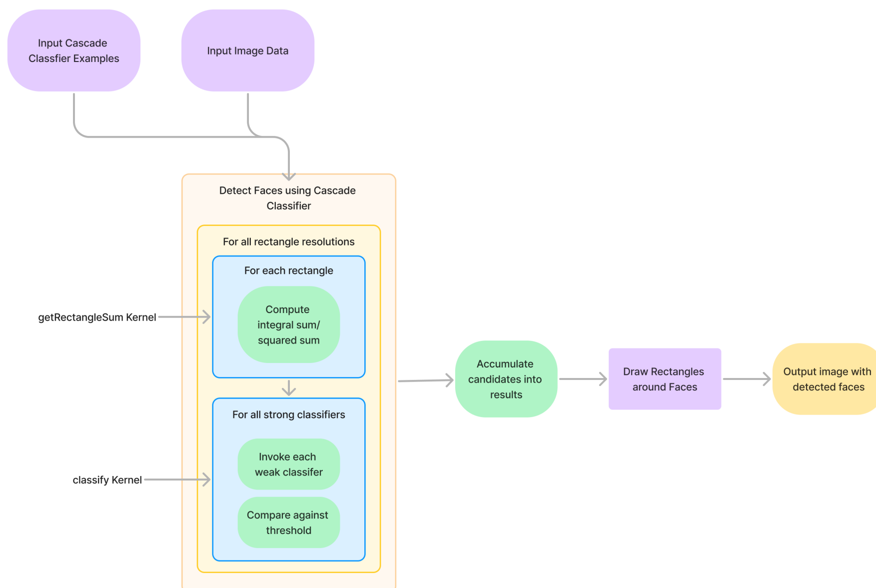
code. It also enabled us to parallelize and debug different portions of the algorithm individually.

We conducted all of our performance tests on the PSC machines since our project required both an NVIDIA GPU as well as access to the Open-CV library.

3.2 Mapping the problem

In terms of mapping our sequential code to utilize CUDA parallelization on GPUs, we aimed to isolate specific parts of the algorithm to do in parallel.

Our final parallelized CUDA algorithm is shown in the flowchart below. Note the GPU-optimized processes are shown in blue:



3.2.1 Cascade Classifier

One of the main challenges of our parallelization was the cascade classifier. Inherently, the cascade classifier is a sequential process. The cascade is structured as a series of stages where each stage contains one or multiple weak classifiers. The weak classifiers in each stage are usually designed to identify simple features of the target object (e.g. edges, lines, or specific texture patterns). Each stage of the classifier makes a binary decision about whether the region of the image under consideration likely contains the object of interest. If a stage classifies the region as negative (i.e., the object is not present), the processing of that region stops, and it is rejected as a potential match. If the stage classifies the region as positive, the region is passed on to the next stage in the cascade. The key challenge here was to parallelize this process without doing too much extra computation due to the dependencies in decisions throughout classification.

While the cascade itself executes sequentially since each stage must be passed before proceeding to the next, we parallelized the evaluation within each stage. We found this to be the best tradeoff between parallelization speedup and minimizing computational redundancy. We wrote a CUDA kernel function for the evaluation of weak classifiers shown below.

The kernel evaluates a set of weak classifiers against a sub-region of an image to determine if the region meets certain criteria defined by the classifiers. The function computes a unique index for each thread across all blocks.

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

This index is used to assign a specific weak classifier to each thread, ensuring that each classifier is evaluated independently of others. The kernel checks if the thread's index is less than the total number of classifiers. If true, the thread proceeds to evaluate its assigned classifier.

```
integral.computeFeatureDevice(classifiers[idx].haarlike, d_data
    , sx, sy);
```

Depending on the type of the Haar-like feature, an adjustment is made to the feature value by adding a normalized product of dimensions and the mean pixel value of the region.

```
if (classifiers[idx].haarlike.type == 2) {
        f += (classifiers[idx].haarlike.w * 3 * classifiers[idx
    ].haarlike.h * mean) / 3;
    } else if (classifiers[idx].haarlike.type == 4) {
        f += (classifiers[idx].haarlike.w * classifiers[idx].
    haarlike.h * 3 * mean) / 3;
    }
```

This normalization helps in adjusting the feature value relative to the average intensity of the region. Finally, the kernel stores the boolean result of the classifier evaluation into the results array, based on whether the computed score meets the specified threshold.

```
results[idx] = classifiers[idx].classify(f) * weights[idx];
```

By evaluating classifiers in parallel, the kernel minimizes latency and maximizes throughput. This approach scales well with the increase in GPU cores, as more classifiers can be evaluated simultaneously.

3.2.2 Detect Function

For the detect function, we implemented a combination of both host and device functions that we found to work the most effectively. The `getRectangleSumKernel` is designed to compute the sum of pixel values within a specified rectangle (sub-window) of the integral image.

The detect function uses the cascade classifier to detect objects within the image. In the sequential version of the code, the algorithm implemented a sliding window approach. The approach involves moving a window of a fixed size across the image and analyzing the content within that window at each position to determine if it contains the object of interest (a facial feature in our case). This aspect seemed like an interesting area to parallelize since we would have to navigate the dependencies in the image as well as keep computation costs low.

Instead of moving the window step by step in a single sequence, the CUDA kernel is designed to process multiple windows simultaneously. Each thread in the GPU can handle the classification of a different window beginning at its corresponding pixel. This allows for the system to analyze all windows of a single size at once, eliminating the need for considering window dependencies. For each rectangle, the kernel computes the sum of pixel values within the window based on the integral image.

We do this for several rectangle sizes, adjusting the `baseResolution` of the cascade classifier by scaling it up for the next iteration to detect larger objects. This scaling factor is passed in as a parameter and is crucial for a multi-scale detection approach where objects of varying sizes are detected by progressively increasing the window size. The loop continues until the `baseResolution` is larger than the image dimensions, ensuring that all possible object sizes are checked.

After all windows of a single size have been evaluated, we synchronize the CUDA threads and invoke the cascade classifier before continuing on to the next size. This is when candidate boxes that pass the calculated threshold are marked as features.

3.3 Starter code

To help implement our sequential C++ facial detection system, we used the following github repository:

`https://github.com/noahlevenson/wasmface`

This github consists of a sequential implementation of the Viola-Jones algorithm with CPU optimizations (no GPU optimizations), and is written entirely in C++ and Javascript. We used their C++ implementation as a baseline for our approach but had to make many adjustments as the code was designed to take in HTML5 ImageData and was run using WebAssembly.

We also utilized human-face.js (which we converted to a JSON file) which was included in the github. This is a cascade classifier model trained to detect faces. The model has been trained on around 13,000 positive examples and 10,000 negative examples generated from stock photos.

4 Results

4.1 Experimental Setup

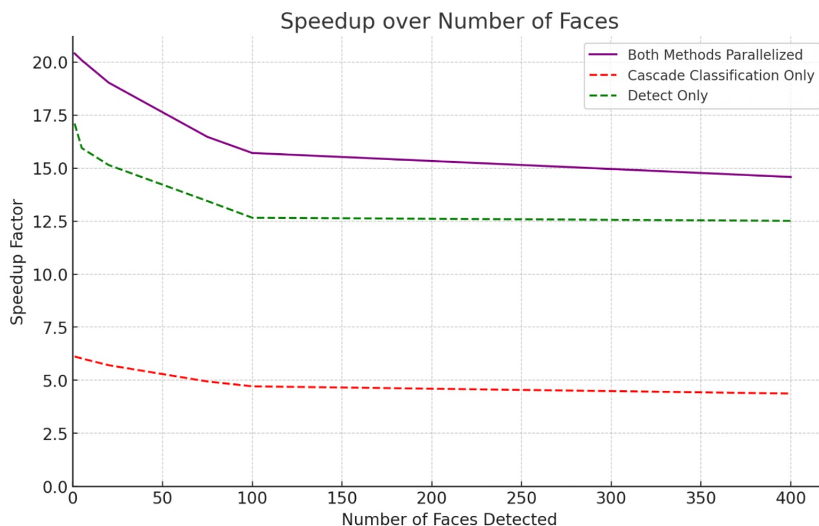
For our performance tests, we gathered images of varying amount of faces to judge the performance of our algorithm with an increasingly large input size. This is because the classifiers often return early once it is realized that a face is not detected, so the presence of more faces would require much more computation.

Our parameters for performance tests are as follows:

1. Delta (detector sweep delta to apply): 1.0
2. Overlap threshold for post processing: 0.3
3. Neighbor threshold for post processing: 5
4. Number of threads for strong classification: 128
5. Number of blocks for strong classification based on number of weak classifiers

4.2 Measuring Performance

Since our project involved optimizing the current sequential implementation of the Viola Jones algorithm, we wanted our performance metrics to encapsulate speedup and computation time. We measured speedup comparing the performance of the our parallel CUDA implementation to a the C++ sequential algorithm as a baseline. In the context of our project, computation time would be an important metric, especially considering real-life applications concerning live video feeds and large dataset processing.

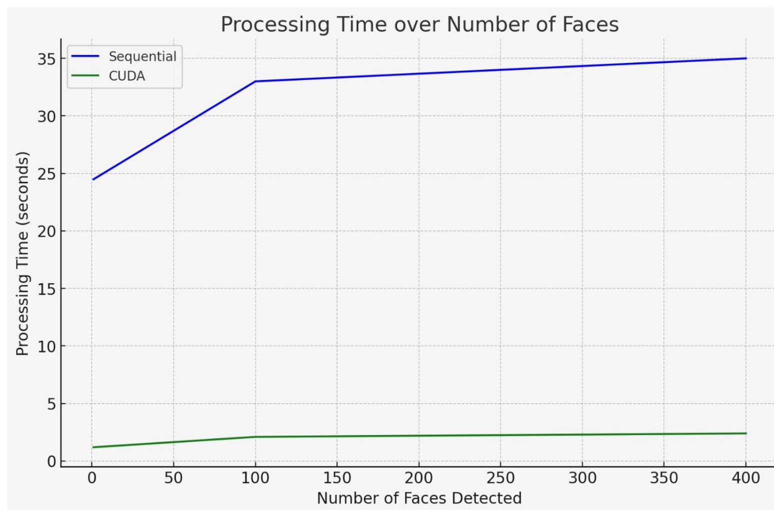


We can see from the graph that both methods combined achieve the highest speedup, starting at nearly 20x when very few faces are detected and stabilizing around 15x as the number of faces increases. The initial high speedup suggests that parallelizing both components is significantly more effective when the number of detected faces is low. This could be due to less computational overhead and fewer data dependencies early in the process when fewer faces are involved.

As the number of faces increases, the speedup factor decreases, particularly noticeable in the purple line. This decline could be due to increased computational complexity and overhead from managing more data (i.e., more detected faces requiring more classification checks and possibly more integral image calculations across different sub-windows).

For the cascade classification only, the declining trend implies that as the workload increases with more faces, the potential for parallelizing this task alone offers diminishing returns, possibly due to increased synchronization or communication overhead among threads. This makes sense in terms of the algorithm as the number of faces increase since this increases the computational load for evaluating the classifiers.

Our analysis suggests that while parallelization significantly improves performance, the maximum benefit is derived when both integral image computation and strong classification are parallelized together. This strategy effectively balances the computational load across the GPU's resources, leading to better overall performance.



The graph above shows the processing time required for facial detection using both the sequential (C++ CPU-based) and a CUDA (GPU-based) approach as the number of faces detected increases. The blue line representing the sequential version shows a gradual increase in processing time as the number of faces detected increases. Starting from around 24 seconds, it rises to approximately 35 seconds as the number of faces approaches 400. The CUDA implementation's processing time remains nearly constant and significantly lower, hovering around just above 3 seconds, regardless of the number of faces detected.

The nearly flat green line indicates that the CUDA-based facial detection maintains consistent even as the number of faces increases. The blue line's upward trend demonstrates that the processing time for the sequential method increases linearly with the number of faces. This behavior makes sense for CPU-based processing where each additional face incrementally adds to the computation load, impacting the overall performance. The stark contrast between the two lines illustrates the effectiveness of using CUDA for parallel processing in this context. The CUDA implementation not only drastically reduces the processing time but also shows stability against increasing workloads.

4.3 Limitations

Although the speedup is significant, especially when both the detect function and cascade classification are parallelized, there is a noted decline in speedup as the number of faces increases, which later stabilizes. Diminishing returns can occur as input size increases (more faces are present), which can be attributed to the overhead costs that negates the benefits of additional parallelism.

In terms of bottlenecks, data transfer between the host and the GPU often becomes a bottleneck, particularly when large images or multiple images are processed in batch operations. The time taken to transfer the image data from the host memory to the GPU memory can significantly impact overall performance, particularly if the integral images or classification data are large.

GPU performance using CUDA can degrade due to thread divergence within warps, where different threads of a single warp follow different execution paths. This situation can occur when different parts of the image require significantly different amounts of computation, causing some threads to idle while others are working, leading to inefficient use of the GPU cores.

4.4 Deeper Analysis

We can likely split execution time for our CUDA implentation in the following way:

(i) Data Transfer Time (10%) : Time spent transferring data between host (CPU) memory and device (GPU) memory. Note this is an estimation based on how our implementation scaled over input sizes.

(ii) Kernel Execution Time (85%) : Time consumed by the actual computation kernels running on the GPU including:

- Integral Image Calculation (15%): Time taken to compute integral images necessary for feature calculation.
- Feature Calculation (35%): Time used to compute features from the integral image. This is highly dependent on the number of candidate features in the image and may

vary from this estimation.

– Classification (35%): Time spent on executing the cascade classifier which uses the computed features to detect faces. This is highly dependent on the number of candidate features in the image and may vary from this estimation.

(iii) Synchronization and Overhead (5%): Time spent on synchronizing between different threads and other overheads including kernel launch overhead. Note this is an estimation based on how our implementation scaled over input sizes.

We know that the kernel execution time is highly dependent on feature calculation and classification because our speedup decreases when the number of faces increases, which only impacts the level of feature and classification computation required.

Because of the combination of both host and device functions, one area of improvement could have been reducing the data transfer time, especially for high-resolution images or real-time applications. Techniques like using asynchronous memory transfers can help.

Additionally, our hope to achieve goal was to improve cache efficiency and reducing cache misses despite the distributed work amongst processors. Unfortunately, we were not able to investigate and implement cache optimizations due to the time it took to implement our parallelization techniques.

4.5 Machine Choice

The choice of a GPU for our project proved sound since tasks like integral image computation and feature calculation can be parallelized in terms of CUDA blocks, making them well-suited for the GPU's architecture. The Viola-Jones algorithm, especially when implementing a cascade classifier, is computationally intensive and benefits from the parallel processing power of GPUs.

While CPUs are capable of running the algorithm, the sequential nature of CPU processing would result in much higher processing times for large images or real-time requirements (as shown in the graphs above). The parallelism offered by modern multi-core CPUs is still limited compared to GPUs, especially for large-scale image data.

5 References

Github: https://github.com/noahlevenson/wasmface

(see Section 3.3 for more information on how we used this resource)

Research Paper: https://www.cs.cmu.edu/ efros/courses/LBMV07/Papers/viola-cvpr-01.pdf

6 Work by Student

Ria Manathkar (50%), Gaurika Sawhney (50%)

We worked collaboratively on every aspect of the project, including write-ups, reports, coding/debugging, etc.

7 Appendix

7.1 Initial Goal Setting

Plan to Achieve

Goal	Description	Justification
1	Successfully implementing the Viola-Jones algorithm in CUDA.	As a foundational building block for our project, we need to ensure we have a working sequential algorithm that we can use to measure the speedup of following parallel versions we create.
2.1	Assigning window processing (of all sizes) to CUDA blocks so these can all be processed in parallel	By determining all windows with various window sizes at the beginning, we will be able to distribute the work for each window better in terms of CUDA blocks, making the algorithm more efficient
2.2	Parallelize the steps the cascade classifier takes for each integral image	This may be challenging since the cascading classifier is inherently sequential due to layered dependencies. If we are able to parallelize this step somehow, by breaking apart some functions or finding a way around the dependencies, this could help speed up immensely.
3	Improving the amount of memory accesses across the algorithm	As stated above, this face detection algorithm involves many accesses to the input and integral image so overall latency could be greatly impacted by the frequency of memory accesses.
4	Run experiments on both implementations with different problem sizes and input images	We want to determine our performance limitations and the tradeoffs between our two potential methods of parallelization.

Hope to Achieve

Goal	Description	Justification
5	Improving cache efficiency and reducing cache misses despite the distributed work amongst processors	Because each processor will be processing windows across different parts of the image, cache coherency will be a big challenge. We hope to make some optimizations for this, by better distributing work across processors.