Parallel Face Detection

Ria Manathkar and Gaurika Sawhney
https://parallel-s24.github.io/final-project/

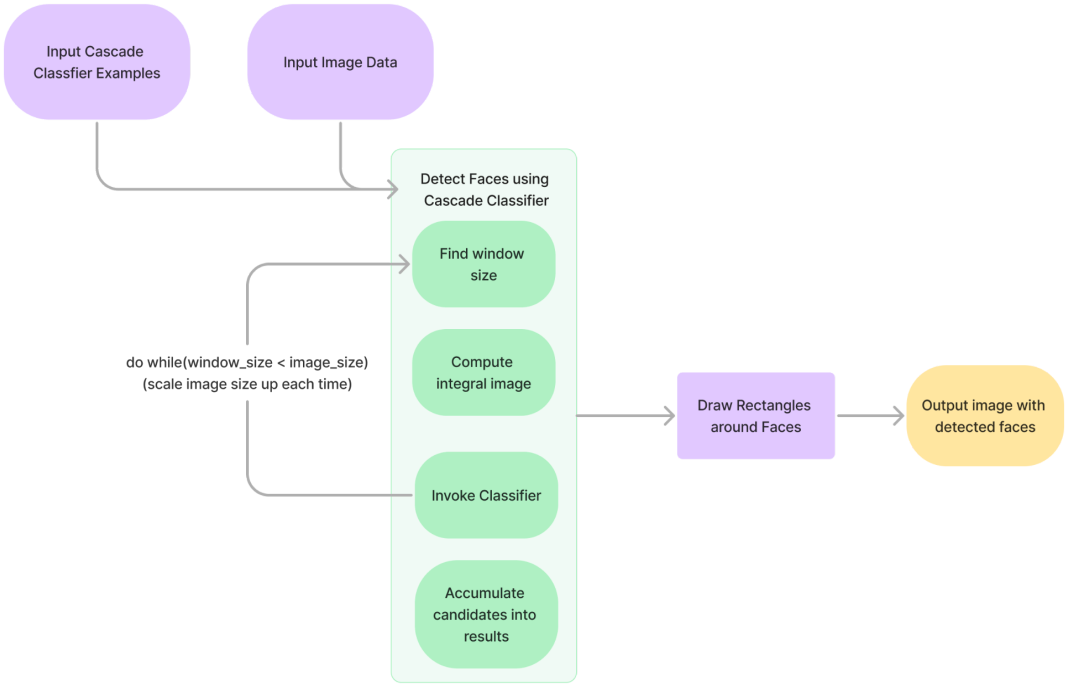
1. Summary

We are going to implement an optimized facial detection system using CUDA.

2. Background

Our project will involve the parallelization of the Viola-Jones face detection algorithm using CUDA in the context of facial detection. This algorithm, well known for its effectiveness in detecting faces within images, operates by leveraging a cascade of classifiers applied to sub-windows of an image. These sub-windows are evaluated across different scales to detect faces of various sizes. The original implementation is single-threaded and CPU-based.

The Viola-Jones algorithm begins by converting the input image into an integral image to reduce the computational complexity of summing pixel values over image subregions. Then, the algorithm employs a sliding window approach, where each window is evaluated for the presence of facial features using a pre-trained set of classifiers (e.g. to identify simple features such as edges and lines that are indicative of facial components).



When reviewing the Viola-Jones algorithm, we noticed aspects of the algorithm that involved redundant work that could be exploited for parallel processing. The evaluation of different images (integral, sub-windows, etc.) could be partitioned into block-like processing using CUDA. The initial algorithm was written sequentially due to the inherent dependencies of the classifier outputs for each window size, so we will have to navigate how best to accumulate this data. Some areas we plan to explore how to parallelize include:

- Evaluating each window for a single integral image. Each window is disjoint from its neighbor so we could leverage CUDA blocks to process and isolate candidate points in parallel and then accumulate before evaluating the next window size.
- Evaluating sub-windows of different sizes at the same time. Since the candidate points found from one iteration of the cascade classifier for different window sizes are independent, we could utilize CUDA blocks to evaluate the image with varying window sizes.

3. The Challenge

There are a few aspects of the Viola-Jones algorithm that will make it challenging to parallelize. In terms of dependencies, the cascade of classifiers in the Viola-Jones algorithm processes sub-windows sequentially, where each stage of the classifier may discard a sub-window from further processing. This sequential dependency is critical for reducing computational load but complicates parallelization. Moreover, each pixel within the integral image depends on the values above and to the left of it. This restricts straightforward partitioning of the integral image computation, though we plan to explore applications of the 2D grid-solver we discussed in class and other static assignment methods.

The workload for each sub-window varies significantly since some sub-windows are discarded after the first few classifiers, while others may pass through multiple stages of the cascade. This variability can lead to load imbalances, where some threads or cores finish their tasks much earlier than others, and thus may be left idle waiting for the other threads to finish.

Considering the implications for memory access, we must prioritize cache efficiency and minimize memory access when possible. The algorithm requires access to various parts of the image and the integral image frequently. This can lead to high memory bandwidth usage and latency if our augmented algorithm does not consider these implications.

Finally, results from various windows will need to be accumulated somehow at the end, involving synchronization and communication between parallel processors. Currently, we are not entirely sure how this final accumulation step will be implemented, however are considering implementing a gather or reduction technique (similar to assignment 4) but in CUDA.

4. Resources

We will be using an existing open-source implementation of the Viola-Jones algorithm available in the OpenCV library as a reference point. Specifically, utilizing the cv::CascadeClassifier class for face detection. Our focus would be on rewriting specific components to utilize CUDA for parallel processing,

such as the calculation of integral images and the sliding window mechanism over the image data. We plan to use some online OpenCV tutorials to do this since we won't be using any starter code.

We plan to read the following paper for better insight to the original algorithm:
P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features", in Conference on Computer Vision and Pattern Recognition 2001
(https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf)

5. Goals & Deliverables

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.

For the final demo, we plan to show the face detection output on a few pre-selected images as well as live images we will take of our classmates to show the correctness and efficiency of our algorithms. To communicate our achievement, we will show speedup graphs of our two different methods and discuss the tradeoffs we discovered. We hope to illustrate what aspects of this algorithm benefited from parallel processing the most and why. Moreover, we want to learn how best to leverage CUDA in constrained problems such as the Viola-Jones face detection algorithm and how we can apply these lessons to future real-world problems.

6. Platform Choice

We are choosing to use C++, OpenCV, and CUDA. CUDA enables the exploitation of large amounts of parallelism, significantly reducing the time required for face detection across images. It also provides a flexible and diverse memory hierarchy, including global, shared, constant, and texture memory, which can be used for different optimization needs. For example, using shared memory for frequently accessed data within a thread block (reducing latency and bandwidth demand on slower global memory) or using constant and texture memory for the cascade classifier parameters (which are read multiple times across various threads but not modified, benefiting from caching). We will use OpenCV to load a pre-trained cascade classifier using C++, which we will employ in our parallelized algorithm.

7. Schedule

Week	Item
March 31- April 6	● Set up CUDA environment ● Implement and test the Viola-Jones algorithm sequentially.
April 7 - April 13 [CARNIVAL]	● Research optimization strategies for parallel execution ● Prepare detailed plans for Goals 2.1, 2.2, and 3.
April 14 - April 20	● Implement parallel window processing ● Explore parallelizing cascade classifier ● Project Milestone Report (Due April 16th)
April 21 - April 27	● Implement parallelizing cascade classifier ● Debug and test parallel implementations
April 28 - May 4	● Optimize cache efficiency ● Implement memory optimization strategies ● Conduct comprehensive testing and performance evaluations
May 5 - May 11	● Final Project Report (Due May 5th) ● Project Poster Session (Due May 6th)