CUDA-based Acceleration of the Viola-Jones Algorithm

Ibrahim Binmahfood

Kunjan Vyas

Robert Wilcox

Abstract

This report presents a CUDA-based parallel implementation of the Viola-Jones face detection algorithm. The Viola-Jones algorithm typically faces performance issues in integral image computation and cascade classifier evaluation. To improve runtime, the cascade classifier evaluation was parallelized using CUDA, significantly reducing execution times compared to the CPU implementation. Integral images are computed on the host (CPU), while candidate detection with the sliding window approach is executed on the CUDA-enabled GPU.

We referenced Francesco Comaschi's paper which claims the fixed window implementation is much more computationally expensive and time intensive as compared to sliding window implementation. Our results and analysis are following a similar trend. The CUDA based implementation takes about 60% to 70% less time as compared to sequential implementation. We are plotting parameters such as time taken and accuracy of both programs to help us visualize results better. Interestingly, the accuracy of the GPU based implementation is lower compared to the CPU program.

Introduction

Face detection is important in many applications such as security, video surveillance, and facial recognition systems. Viola-Jones is a widely-used method due to its efficiency but faces computational bottlenecks, making GPU parallelization beneficial. This project addresses these bottlenecks by leveraging CUDA to parallelize the detection step.

Background

The Viola-Jones algorithm employs Haar-like features and a cascade of weak classifiers to efficiently detect faces in images. Haar-like features are rectangular features that capture

differences in image intensity, enabling the algorithm to distinguish between facial and non-facial regions. Integral images (summed area tables) are used to enable rapid computation of these features, significantly speeding up the feature evaluation process. Despite these optimizations, the sequential evaluation of the cascade classifier, which involves applying a series of weak classifiers to candidate regions, remains computationally demanding. This sequential nature presents a bottleneck that can be alleviated through parallel processing.

Implementation

The Viola-Jones algorithm is implemented using a combination of CPU and GPU processing. The following section details the implementation of the CUDA code, with the code files (in appendices) cited in brackets where appropriate [appendix]. The main.cpp file serves as the control center, responsible for the overall execution flow [A]. It handles image loading and pre-processing on the CPU, initializes the Haar cascade classifier, and manages the transfer of data and execution control to the GPU for the parallel detection phase. After the GPU processing is complete, main.cpp receives the results, performs post-processing, and outputs the detected faces. This design clearly separates the host (CPU) and device (GPU) responsibilities.

The core Viola-Jones algorithm components are implemented in the haar_cuda.cpp/h files, which manage the cascade classifier logic [D, E], These files handle loading the classifier parameters and setting up the data structures required for feature evaluation. The computationally intensive task of cascade classifier evaluation is parallelized and executed on the GPU using CUDA, implemented in cuda_detect.cu/cuh [B, C]. Image I/O and integral image calculations are performed on the CPU using the image_cuda.c/h files [F, G]. Finally, the rectangles_cuda.cpp/h files handle post-processing of the detection results, specifically grouping candidate rectangles and drawing bounding boxes [H, I].

Detailed Description of Key Modules

- **`main.cpp`**: The `main.cpp` file orchestrates the interaction between the CPU and GPU [A]. It begins by parsing command-line arguments to obtain the input and output image paths. It then loads the input image and initializes the cascade classifier by calling functions from `haar_cuda.cpp` [A, D]. The integral image is computed on the CPU, and the necessary data, including pointers to the integral image data, is passed to the CUDA functions for GPU processing. The `main.cpp` file manages the multi-scale detection process, iterating through different scales and invoking the `runDetection` function in `cuda_detect.cu` for each scale [A, B]. After the GPU kernel execution, `main.cpp` retrieves the detection results from the GPU, performs post-processing steps such as grouping rectangles, and finally, saves the output image with bounding boxes drawn around the detected faces [A]. This file also includes timing mechanisms using `clock_gettime` to measure the execution time of the detection process [A].

- **`haar_cuda.cpp/h`**: The `haar_cuda.cpp/h` files are responsible for managing the Haar cascade classifier [D, E]. The `readTextClassifier` function loads the classifier parameters, including the number of stages, number of weak classifiers per stage, rectangle features, weights, and thresholds, from external files (`class.txt`, `info.txt`) [D]. These parameters are stored in arrays, such as `stages_array`, `rectangles_array`, `weights_array`, and `stages_thresh_array` [D]. The `setImageForCascadeClassifier` function sets up the integral image data and prepares the classifier for evaluation [D]. The `myCascade` struct in `haar_cuda.h` defines the data structures used to represent the cascade classifier, including members such as `orig_window_size`, `inv_window_area`, and pointers to integral image data (`sum`, `sqsum`) [E]. These structures are modified to facilitate data transfer and processing on the GPU.

- `Cuda_detect.cu`/`cuh`: The `cuda_detect.cu/cuh` files contain the CUDA implementation of the Viola-Jones algorithm [B, C]. The `cuda_detect.cu` file includes the CUDA kernels and supporting device functions. The `detectKernel` is the primary CUDA kernel, responsible for parallelizing the sliding window detection [B]. Each thread in this kernel processes a candidate window, and the grid and block dimensions are calculated to map threads to candidate windows [B]. The `evalWeakClassifier_device` function implements the evaluation of weak classifiers on the GPU, mirroring the functionality of the CPU-based `evalWeakClassifier` [B, D]. Unified Memory is used to manage data transfer between the host and device, allowing both the CPU and GPU to access the same memory locations for integral images and cascade classifier data [B]. Classifier parameters are stored in constant memory on the device to provide fast read-only access during kernel execution [B]. Atomic operations, specifically `atomicAdd`, are used to ensure thread-safe updates to the candidate count when detections are found [B].

- `image_cuda.c/h`: Handles reading and writing of PGM images and computes integral images. These files manage image input and the creation of integral images, a preprocessing step required for the Viola-Jones algorithm.

- `rectangles_cuda.cpp/h`: Implements functions for grouping detection results and drawing bounding rectangles around detected objects. These files handle post-processing of the detection results.

CUDA Implementation Details

The Viola-Jones algorithm was modified to leverage GPU parallelization effectively. The integral images are computed once on the CPU, and then transferred to the GPU. Detection at each

pyramid scale is performed using CUDA kernels. Key CUDA functions and implementation strategies include:

- **`runDetection()`**:

  This is the primary CUDA kernel function, responsible for the parallel execution of the Viola-Jones detection algorithm on the GPU [B]. It implements the sliding window technique, with each thread in the kernel processing a single candidate window in parallel [B].

  The kernel's grid and block dimensions are configured to map threads to candidate windows. The number of blocks and threads are calculated based on the search space within the image [B]. For example:

```
C/C++
dim3 blockDim(16, 16); // Example block size

dim3 gridDim((x_max + blockDim.x - 1) / blockDim.x,

               (y_max + blockDim.y - 1) / blockDim.y);

detectKernel<<<gridDim, blockDim>>>(...);
```

  Within the kernel, each thread calculates the feature values and passes them to the `evalWeakClassifier_device` function [B].

- **`setImageForCascadeClassifier()`**:

  This function prepares the integral images and cascade classifier data for use within the CUDA kernels [D, E]. While the integral image calculation remains on the CPU, this function is crucial for setting up the necessary pointers and data structures.

  Specifically, it loads the indices of the four corners of the filter rectangle. It also loads the index of the four corners of the filter rectangle and sets up the scaled rectangle pointers [D]. This involves calculations to access the correct memory locations within the integral image data [D].

- **`ScaleImage_Invoker()`**:

  This function, executed on the CPU, controls the multi-scale detection process [A, D]. It iterates over the image scales in the pyramid, adjusting the window size and search area

for each scale [A, D]. For each scale, it launches the `runDetection` CUDA kernel to perform the detection on the GPU.

This function also calculates the size of the image scaled up and the size of the image scaled down [D]. It also determines the difference between sizes of the scaled image and the original detection window [cite: haar_cuda.cpp].

- **Unified Memory**:

  CUDA Unified Memory is used to simplify data management between the host (CPU) and device (GPU) [B]. This allows both the CPU and GPU to access the same memory locations, reducing the need for explicit memory transfers in some cases.

  For example, integral image data and cascade classifier parameters are allocated using `cudaMallocManaged`, making them accessible from both the CPU and GPU [B].

```C/C++
CUDA_CHECK(cudaMallocManaged((void**)&d_sum, sizeof(MyIntImage)));

CUDA_CHECK(cudaMallocManaged((void**)&(d_sum->data), dataSize));
```

- **Constant Memory**:

  Classifier parameters, such as Haar feature definitions and classifier weights, are stored in CUDA constant memory [B]. This provides fast read-only access for the GPU during kernel execution. The classifier parameters are transferred from the host to the device's constant memory using `cudaMemcpyToSymbol` [B].

```C/C++
CUDA_CHECK(cudaMemcpyToSymbol(d_stages_array, &d_stages_array_dev,
sizeof(int*)));
```

- **Parallel Granularity**:

  The parallel implementation focuses on parallelizing the sliding window operation [B]. Each thread in the `detectKernel` is responsible for processing a single candidate window. This fine-grained parallelization allows for efficient utilization of the GPU's many cores.

- **Atomic Operations**:

  Atomic operations, specifically `atomicAdd`, are used to ensure thread-safe updates to shared memory locations [B]. This is necessary when multiple threads might try to modify the same memory location simultaneously, such as when incrementing a counter for the number of detected objects.

```
C/C++
atomicAdd(&d_candidateCount, 1);
```

- **Haar Feature Evaluation**:

  The `evalWeakClassifier_device` function in `cuda_detect.cu` is the CUDA equivalent of the `evalWeakClassifier` function in the CPU implementation [B, D]. It calculates the Haar feature response for a given candidate window.

  This involves accessing the integral image data at specific locations, performing calculations based on the Haar feature definition, and comparing the result to a threshold [B]. The CUDA version is adapted to work with the GPU's memory model and execution paradigm.

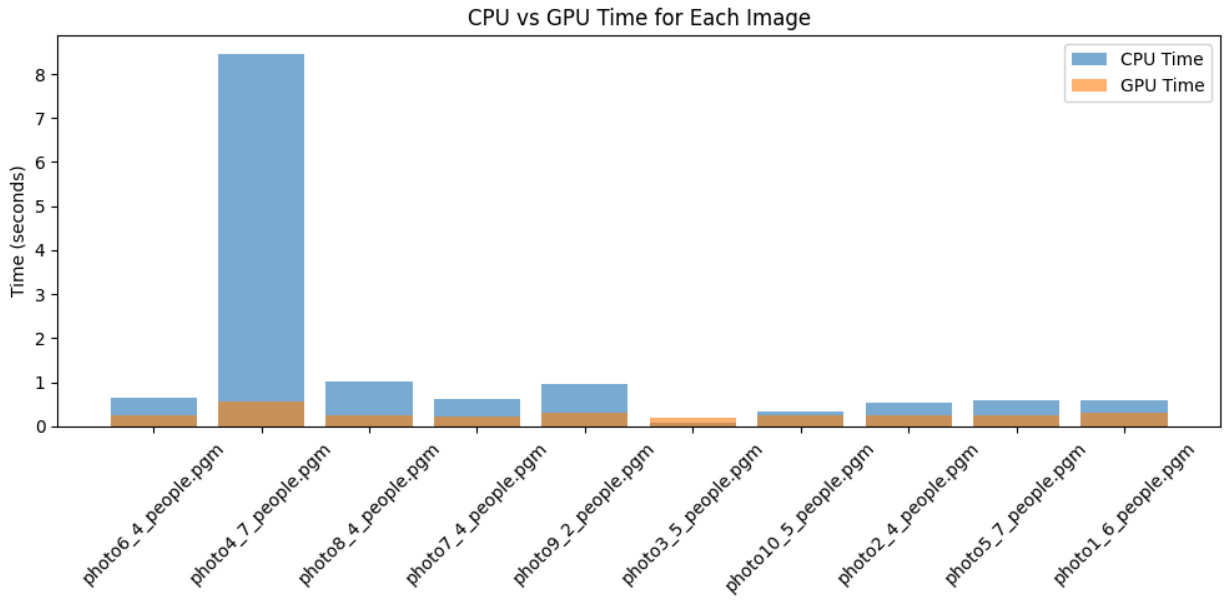
Experimental Setup and Results

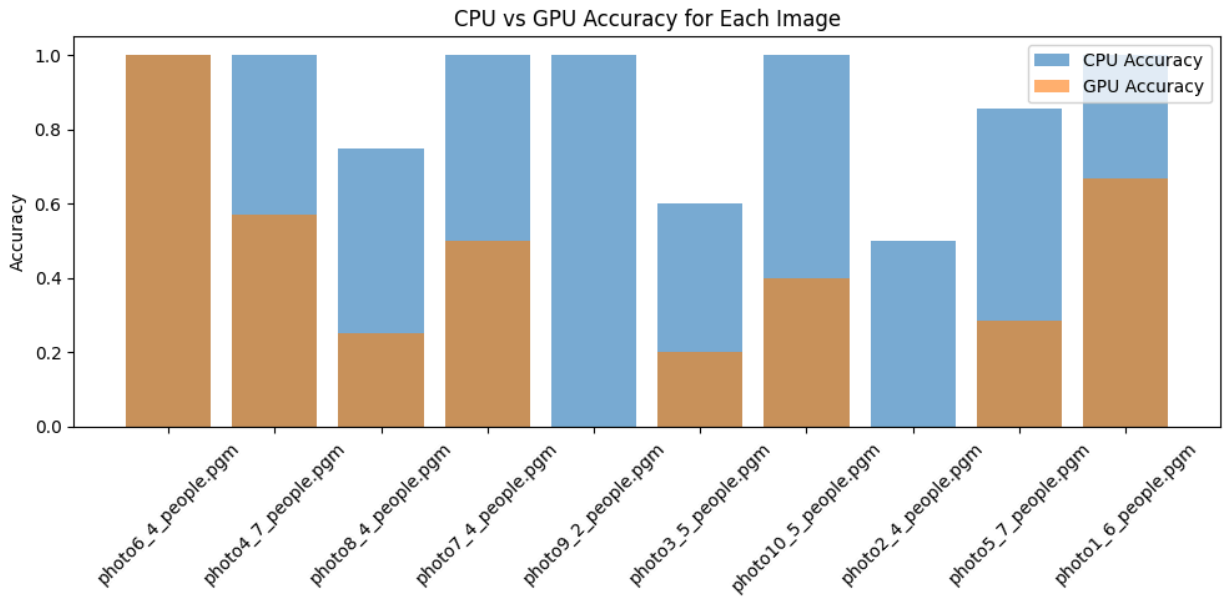
The implementation was tested on a CUDA-capable GPU system. Since we had a Nvidia 3060 TI GPU available to us, we prioritized our CUDA compilation of the source code for that. However, we thought about testing our implementation on Google Colab as well. Google Colab offers T4 Nvidia GPUs. Google Colab uses a Unix-based environment. A Makefile build system was used since it was versatile on Unix-based systems. The general stages of our setup are compilation of GPU and CPU code-base, automated-tests, and data analysis from GPU and CPU runs.

The data set we used consisted of images of candidates ranging from 2 to 7 maximum. The images before being processed by our code base, require to be converted to PGM format. The images are named in this format: *photo#_<number of candidates>_people.pgm* According to our CPU and GPU runs, we observe a high speedup (in terms of time in seconds) for GPU runs as compared to CPU runs across our data-set. However, the accuracy from CPU runs (how many candidates detected in an image) are higher than that of GPU runs. The below table and plots represent the trends described.

An interesting trend observed, a higher speedup of the GPU time results in less accurate detections of candidates according to our data set. However, this is not always the case as noted for the outliers (photo6_4_people.pgm and photo4_7_people.pgm).

Image	Speedup (v.s. CPU time in seconds)
photo1_6_people.pgm	1.88
photo2_4_people.pgm	2.15
photo3_5_people.pgm	0.43
photo4_7_people.pgm	14.80
photo5_7_people.pgm	2.43
photo6_4_people.pgm	2.66
photo7_4_people.pgm	2.64
photo8_4_people.pgm	4.00
photo9_2_people.pgm	2.99
photo10_5_people.pgm	1.33





Analysis and Discussion

The CUDA implementation effectively parallelizes the sliding window step, leading to substantial speed improvements. However, the CUDA program performed worse at identifying faces.

- Moving integral image computation to the GPU.
- Optimizing data transfers between CPU and GPU memory.

Conclusion

This report demonstrates that implementing the Viola-Jones algorithm on CUDA significantly improves detection performance through the parallel evaluation of candidate windows. The CUDA implementation achieves substantial speedup compared to the CPU implementation. Future improvements include transferring more processing steps to the GPU, such as integral image computation, and optimizing data transfer strategies to further reduce overhead and enhance performance. The source code is located on GitHub at [Facial Feature Extraction with CUDA]

References

[1] F. Comaschi, S. Stuijk, T. Basten and H. Corporaal, "RASW: A run-time adaptive sliding window to improve Viola-Jones object detection," 2013 Seventh International Conference on Distributed Smart Cameras (ICDSC), Palm Springs, CA, USA, 2013, pp. 1-6, doi: 10.1109/ICDSC.2013.6778224.

[2] Yi-Qing Wang, An Analysis of the Viola-Jones Face Detection Algorithm, Image Processing On Line, 4 (2014), pp. 128–148.

[3] dev7saxena, “GitHub - dev7saxena/Viola-Jones-cpp: vj_cpp_original_code,” *GitHub*, 2016. https://github.com/dev7saxena/Viola-Jones-cpp/tree/master (accessed Mar., 2025).

[4] https://docs.nvidia.com/cuda/cuda-c-programming-guide/. (Accessed: Mar., 2025).

Appendix A: main.cpp

```C/C++
/**
 * @file main.cpp
 * @brief Main function for CUDA-based image detection.
 *
 * Date: 03.12.25
 * Authors: Ibrahim Binmahfood, Kunjan Vyas, Robert Wilcox
 *
 * This program demonstrates a CUDA-accelerated multi-scale face (or object) detection
 * algorithm using a Haar-like cascade classifier. The steps include:
 *   - Parsing command-line arguments for input/output image paths.
 *   - Checking for CUDA device availability.
 *   - Loading an input image and computing its integral images.
 *   - Initializing and linking a cascade classifier.
 *   - Computing additional offsets for the detection window.
 *   - Performing multi-scale detection using a CUDA detection routine.
 *   - Grouping and drawing detection results (candidate rectangles).
 *   - Saving the output image and cleaning up allocated resources.
 *
 * Usage: ./program -i [path/to/input/image] -o [path/to/output/image]
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <string.h>
#include <unistd.h>
#include <vector>
#include <cuda_runtime.h>
#include <assert.h>
#include <time.h>
#include <sys/time.h>

#include "image_cuda.h"
#include "haar_cuda.h"
#include "cuda_detect.cuh"  // runDetection is declared here.



#define MINNEIGHBORS 1

//#define FINAL_DEBUG



extern int iter_counter;
```

```
inline int myRound(float value);  // Prototype for the inline function.


void ScaleImage_Invoker(myCascade* _cascade, float _factor, int sum_row, int
sum_col, std::vector<MyRect>& _vec);


extern void nearestNeighbor(MyImage* src, MyImage* dst);


// Helper function to scan classifier data and compute the extra offsets (in x
and y)

// that account for all rectangle extents beyond the base detection window.

void computeExtraOffsets(const myCascade* cascade, int* extra_x, int* extra_y);


// Helper function to print a subset of values from an integral image.

void debugPrintIntegralImageGPU(MyIntImage* img, int numSamples);




int main(int argc, char** argv) {

    // Variable declarations for option parsing and runtime measurement.

    int opt;

    int rc;


    // Pointers to hold input and output file paths.

    char* input_file_path = NULL;

    char* output_file_path = NULL;
```

```
    // Variables to store start and end times for performance measurement.
    struct timespec t_start;
    struct timespec t_end;


    // Parse command-line options using getopt.
    while ((opt = getopt(argc, argv, "i:o:")) != -1) {
        switch (opt) {
            // Option 'i' for input image file path.
        case 'i':
            // Verify that the specified file exists.
            if (access(optarg, F_OK) != 0) {
                fprintf(stderr, "ERROR: path to file %s does not exist\n",
optarg);
                fprintf(stderr, "Usage: %s -i [path/to/image] -o
[path/to/output/image]\nExitting...\n", argv[0]);
                exit(1);
            }
            // Store valid input file path.
            input_file_path = optarg;
            break;


            // Option 'o' for output image file path.
        case 'o':
```

```
            output_file_path = optarg;
            break;

            // Default case if an unknown option is provided.
        default:
            fprintf(stderr, "Usage: %s -i [path/to/image] -o
[path/to/output/image]\nExitting...\n", argv[0]);
            exit(1);
        }
    }

    // Indicate entry into the main function.
    printf("-- entering main function --\n");

    // Check for available CUDA devices.
    int deviceCount = 0;
    cudaError_t cudaStatus = cudaGetDeviceCount(&deviceCount);
    if (cudaStatus != cudaSuccess || deviceCount == 0) {
        printf("No CUDA devices found or CUDA error: %s\n",
cudaGetErrorString(cudaStatus));
        return 1;
    }
    printf("Found %d CUDA device(s).\n", deviceCount);
```

```
// 1. Load the input image.
MyImage imageObj;
MyImage* image = &imageObj;
if (readPgm(input_file_path, image) == -1) {
    printf("Unable to open input image\n");
    return 1;
}

// 2. Compute integral images for fast feature computation.
MyIntImage sumObj, sqsumObj;
MyIntImage* sum = &sumObj;
MyIntImage* sqsum = &sqsumObj;
createSumImage(image->width, image->height, sum);
createSumImage(image->width, image->height, sqsum);
integralImages(image, sum, sqsum);

// 3. Initialize the cascade classifier parameters.
myCascade cascadeObj;
myCascade* cascade = &cascadeObj;
cascade->n_stages = 25;
cascade->total_nodes = 2913;
cascade->orig_window_size.width = 24;
cascade->orig_window_size.height = 24;
```

```
    MySize minSize = { 20, 20 };

    MySize maxSize = { 0, 0 };


    // Load the cascade classifier data.

    printf("-- loading cascade classifier --\n");

    readTextClassifier(cascade);

    printf("-- cascade classifier loaded --\n");


    // Validate that the classifier rectangles have been loaded.

    if (cascade->scaled_rectangles_array == NULL) {

        printf("ERROR: cascade->scaled_rectangles_array is NULL after
readTextClassifier!\n");

    }

    else {

        printf("cascade->scaled_rectangles_array is NOT NULL after
readTextClassifier: %p\n", cascade->scaled_rectangles_array);

    }


    // 4. Link the computed integral images to the cascade classifier.

    printf("-- linking integral images to cascade --\n");

    setImageForCascadeClassifier(cascade, sum, sqsum);

    printf("-- After setImageForCascadeClassifier call --\n");

    printf("-- integral images linked to cascade --\n");
```

```
    // Compute extra offsets that adjust the detection window dimensions.
    int extra_x = 0, extra_y = 0;
    computeExtraOffsets(cascade, &extra_x, &extra_y);
    printf("Computed extra offsets: extra_x = %d, extra_y = %d\n", extra_x,
extra_y);


    // Adjust the detection window size to include extra offsets.
    int adjusted_width = cascade->orig_window_size.width + extra_x;
    int adjusted_height = cascade->orig_window_size.height + extra_y;
    printf("Adjusted detection window size: (%d, %d)\n", adjusted_width,
adjusted_height);


    // Allocate buffers for scaled image and its integral images.
    MyImage scaledImg;
    createImage(image->width, image->height, &scaledImg);
    MyIntImage scaledSum, scaledSqSum;
    createSumImage(image->width, image->height, &scaledSum);
    createSumImage(image->width, image->height, &scaledSqSum);
    float factor = 1.0f;


    // ***** Run CUDA detection at each scale in the image pyramid *****
    // Prepare a vector to store all candidate detections from GPU.
    std::vector<MyRect> allGpuCandidates;
    int iter_counter = 1;
```

```
    factor = 1.0f;


    // Start the timer for performance measurement.
    rc = clock_gettime(CLOCK_REALTIME, &t_start);
    assert(rc == 0);


    // Loop over different scales of the image.
    while (true) {
        // Calculate new dimensions based on the scaling factor.
        int newWidth = (int)(image->width / factor);
        int newHeight = (int)(image->height / factor);
        int winWidth = myRound(cascade->orig_window_size.width * factor);
        int winHeight = myRound(cascade->orig_window_size.height * factor);
        MySize sz = { newWidth, newHeight };
        MySize winSize = { winWidth, winHeight };
        // Compute the available difference in dimensions for placing the
detection window.
        MySize diff = { sz.width - cascade->orig_window_size.width, sz.height -
cascade->orig_window_size.height };


        // If the difference is negative, the window no longer fits; exit the
loop.
        if (diff.width < 0 || diff.height < 0)
            break;
```

```
        // Skip scales that produce a detection window smaller than the minimum
allowed size.
        if (winSize.width < minSize.width || winSize.height < minSize.height) {
            factor *= 1.2f;
            continue;
        }

        // Reallocate buffers for the current scale.
        freeImage(&scaledImg);
        freeSumImage(&scaledSum);
        freeSumImage(&scaledSqSum);
        createImage(newWidth, newHeight, &scaledImg);
        createSumImage(newWidth, newHeight, &scaledSum);
        createSumImage(newWidth, newHeight, &scaledSqSum);

        // Scale the input image using nearest neighbor interpolation.
        nearestNeighbor(image, &scaledImg);
        // Compute the integral images for the scaled image.
        integralImages(&scaledImg, &scaledSum, &scaledSqSum);
        // Link the scaled integral images to the cascade classifier.
        setImageForCascadeClassifier(cascade, &scaledSum, &scaledSqSum);

        // Check if the detection window fits within the scaled integral image
dimensions.
```

```
        if (factor * (cascade->orig_window_size.width + extra_x) <
scaledSum.width &&

            factor * (cascade->orig_window_size.height + extra_y) <
scaledSum.height) {

            // Run the CUDA detection for the current scale.

            std::vector<MyRect> gpuCandidates = runDetection(&scaledSum,
&scaledSqSum, cascade, 10000000, factor, adjusted_width, adjusted_height,
iter_counter);

            // Merge the current scale's candidates into the overall candidate
list.

            allGpuCandidates.insert(allGpuCandidates.end(),
gpuCandidates.begin(), gpuCandidates.end());

        }

        else {

            // Scale factor too high; detection window does not fit. (Debug
print commented out.)

        }

        // Increment the scale factor for the next iteration.

        factor *= 1.2f;

    }



    // Group nearby candidate rectangles to remove duplicates.

    groupRectangles(allGpuCandidates, MINNEIGHBORS, 0.4f);



    // Stop the timer and calculate the runtime.

    rc = clock_gettime(CLOCK_REALTIME, &t_end);
```

```
    assert(rc == 0);


    unsigned long long int runtime = 1000000000 * (t_end.tv_sec -
t_start.tv_sec) + t_end.tv_nsec - t_start.tv_nsec;


    // Output the number of candidates and runtime details.

    printf("\nCUDA detection detected %zu candidates.\n",
allGpuCandidates.size());

    printf("Time = %lld nanoseconds\t(%lld.%09lld sec)\n\n", runtime, runtime /
1000000000, runtime % 1000000000);


    // Debug output: print the coordinates and dimensions for each detected
candidate.

    for (size_t i = 0; i < allGpuCandidates.size(); i++) {

        printf("[DEBUG] CUDA Candidate %zu: x=%d, y=%d, width=%d, height=%d\n",

            i, allGpuCandidates[i].x, allGpuCandidates[i].y,
allGpuCandidates[i].width, allGpuCandidates[i].height);

    }


    // 8. Draw detection rectangles (candidate face boxes) on the original
image.

    for (size_t i = 0; i < allGpuCandidates.size(); i++) {

        drawRectangle(image, allGpuCandidates[i]);

    }


    // 9. Save the output image with drawn candidate rectangles.
```

```
    printf("-- saving output --\n");
    int flag = writePgm(output_file_path, image);
    if (flag == -1)
        printf("Unable to save output image\n");
    else
        printf("-- image saved as %s --\n", output_file_path);

    // 10. Clean up and release resources.
    releaseTextClassifier(cascade);
    freeImage(image);
    freeSumImage(sum);
    freeSumImage(sqsum);

    // Free the temporary buffers allocated for scaled images.
    freeImage(&scaledImg);
    freeSumImage(&scaledSum);
    freeSumImage(&scaledSqSum);

    return 0;
}

// Helper function to scan classifier data and compute the extra offsets (in x
and y)
// that account for all rectangle extents beyond the base detection window.
```

```
void computeExtraOffsets(const myCascade* cascade, int* extra_x, int* extra_y)
{
    *extra_x = 0;
    *extra_y = 0;

    int totalRectElems = cascade->total_nodes * 12;

    for (int i = 0; i < totalRectElems; i += 4) {
        int rx = cascade->rectangles_array[i];
        int ry = cascade->rectangles_array[i + 1];
        int rw = cascade->rectangles_array[i + 2];
        int rh = cascade->rectangles_array[i + 3];

        if (rx == 0 && ry == 0 && rw == 0 && rh == 0)
            continue;

        int current_right = rx + rw;
        int current_bottom = ry + rh;

        if (current_right > *extra_x)
            *extra_x = current_right;

        if (current_bottom > *extra_y)
            *extra_y = current_bottom;
```

```
    }
}


// Helper function to print a subset of values from an integral image.
void debugPrintIntegralImageGPU(MyIntImage* img, int numSamples) {
    int width = img->width;
    int height = img->height;
    int total = width * height;

#ifdef FINAL_DEBUG
    printf("GPU Integral image summary: width = %d, height = %d, total values =
%d\n", width, height, total);
#endif

#ifdef FINAL_DEBUG
    // Print the four corner values
    printf("Top-left (index 0): %d\n", img->data[0]);
    printf("Top-right (index %d): %d\n", width - 1, img->data[width - 1]);
    printf("Bottom-left (index %d): %d\n", (height - 1) * width,
img->data[(height - 1) * width]);
    printf("Bottom-right (index %d): %d\n", total - 1, img->data[total - 1]);
#endif

    int step = total / numSamples;
```

```
    if (step < 1)
        step = 1;


#ifdef FINAL_DEBUG
    printf("Printing %d sample values (every %d-th value):\n", numSamples,
step);
    for (int i = 0; i < total; i += step) {
        printf("Index %d: %d\n", i, img->data[i]);
    }
#endif
}
```

Appendix B: cuda_detect.cu

C/C++

```
//
============================================================================
===============

// Filename: cuda_detect.cu

//

// Description:

//     Implements a CUDA-accelerated sliding window detector for the
Viola-Jones algorithm.

//     This implementation evaluates the real weak classifiers and leverages
Unified Memory

//     for integral images, cascade structures, and detection results.

//

// Date: 03.12.25

// Authors: Ibrahim Binmahfood, Kunjan Vyas, Robert Wilcox

//
----------------------------------------------------------------------------



#include "cuda_detect.cuh"  // Include header file for CUDA detection

#include <stdio.h>          // Standard I/O

#include <math.h>           // Math functions

#include <cuda_runtime.h>   // CUDA runtime

#include <device_launch_parameters.h>

#include <vector>           // For std::vector
```

```
#include <string.h>         // For memcpy
#include <assert.h>         // For device-side assertions



//#define FINAL_DEBUG


#define DEBUG_CANDIDATE_X 2015
#define DEBUG_CANDIDATE_Y 863


#define CUDA_CHECK(call) do { \
    cudaError_t err = call; \
    if (err != cudaSuccess) { \
        printf("[CUDA ERROR] %s:%d: %s\n", __FILE__, __LINE__,
cudaGetErrorString(err)); \
        return std::vector<MyRect>(); \
    } \
} while(0)


// ------------------------------------------------------------------
// Constant memory for classifier parameters.
__constant__ int* d_stages_array;
__constant__ float* d_stages_thresh_array;
__constant__ int* d_rectangles_array;
__constant__ int* d_weights_array;
```

```
__constant__ int* d_alpha1_array;

__constant__ int* d_alpha2_array;

__constant__ int* d_tree_thresh_array;


// ------------------------------------------------------------------------
// Declaration of atomicCAS.
extern __device__ int atomicCAS(int* address, int compare, int val);



// ------------------------------------------------------------------------
// Global device variable for debug print count
__device__ int d_debug_print_count = 0;


// ------------------------------------------------------------------------
// Device function: Integer square root for the GPU.
// This function replicates the behavior of the CPU's int_sqrt.
__device__ int int_sqrt_device(int value) {
    int i;
    unsigned int a = 0, b = 0, c = 0;
    for (i = 0; i < (32 >> 1); i++) {
        c <<= 2;
        c += (value >> 30); // get the upper 2 bits of value
        value <<= 2;
```

```
        a <<= 1;
        b = (a << 1) | 1;
        if (c >= b) {
            c -= b;
            a++;
        }
    }
    return a;
}


// ------------------------------------------------------------------
// Device function: Rounding function mirroring CPU implementation
__device__ inline int myRound_device(float value) {
    return (int)(value + (value >= 0 ? 0.5f : -0.5f));
}




// ------------------------------------------------------------------
// Device function: Evaluate a weak classifier for candidate window p.
// Assumes that for each feature, d_rectangles_array stores 12 ints in the order:
// [x_offset1, y_offset1, width1, height1, x_offset2, y_offset2, width2, height2,
//  x_offset3, y_offset3, width3, height3]
```

```
__device__ float evalWeakClassifier_device(const myCascade* d_cascade, int
variance_norm_factor, MyPoint p,
    int haar_counter, int w_index, int r_index, float scaleFactor)
{


    //printf("[Device] entered evalWeakClassifier_device\n");


    // Print candidate coordinates for every 100th candidate


    int* rect = d_rectangles_array + r_index;


    // --- First Rectangle ---
    int tl1_x = p.x + (int)myRound_device(rect[0]);
    int tl1_y = p.y + (int)myRound_device(rect[1]);
    int br1_x = tl1_x + (int)myRound_device(rect[2]);
    int br1_y = tl1_y + (int)myRound_device(rect[3]);


    // Check bounds
    assert(tl1_x >= 0 && tl1_x < d_cascade->sum.width);
    assert(tl1_y >= 0 && tl1_y < d_cascade->sum.height);
    assert(br1_x >= 0 && br1_x < d_cascade->sum.width);
    assert(br1_y >= 0 && br1_y < d_cascade->sum.height);


    int idx_tl1 = tl1_y * d_cascade->sum.width + tl1_x;
```

```
    int idx_tr1 = tl1_y * d_cascade->sum.width + br1_x;

    int idx_bl1 = br1_y * d_cascade->sum.width + tl1_x;

    int idx_br1 = br1_y * d_cascade->sum.width + br1_x;




    int sum1 = d_cascade->p0[idx_br1] - d_cascade->p0[idx_tr1] -
d_cascade->p0[idx_bl1] + d_cascade->p0[idx_tl1];

    sum1 = sum1 * d_weights_array[w_index + 0];



#ifdef FINAL_DEBUG

    // Debug Statement for First Rectangle

    if (p.x == DEBUG_CANDIDATE_X && p.y == DEBUG_CANDIDATE_Y && haar_counter ==
0) {

        printf("[Device DEBUG] Rect 1: tl=(%d,%d), br=(%d,%d), sum1=%d,
weight=%d\n",

            tl1_x, tl1_y, br1_x, br1_y, sum1, d_weights_array[w_index]);

    }

#endif



    // --- Second Rectangle ---

    int tl2_x = p.x + (int)myRound_device(rect[4]);

    int tl2_y = p.y + (int)myRound_device(rect[5]);

    int br2_x = tl2_x + (int)myRound_device(rect[6]);

    int br2_y = tl2_y + (int)myRound_device(rect[7]);
```

```
    assert(tl2_x >= 0 && tl2_x < d_cascade->sum.width);

    assert(tl2_y >= 0 && tl2_y < d_cascade->sum.height);

    assert(br2_x >= 0 && br2_x < d_cascade->sum.width);

    assert(br2_y >= 0 && br2_y < d_cascade->sum.height);


    int idx_tl2 = tl2_y * d_cascade->sum.width + tl2_x;

    int idx_tr2 = tl2_y * d_cascade->sum.width + br2_x;

    int idx_bl2 = br2_y * d_cascade->sum.width + tl2_x;

    int idx_br2 = br2_y * d_cascade->sum.width + br2_x;




    int sum2 = d_cascade->p0[idx_br2] - d_cascade->p0[idx_tr2] -
d_cascade->p0[idx_bl2] + d_cascade->p0[idx_tl2];

    sum2 = sum2 * d_weights_array[w_index + 1];


#ifdef FINAL_DEBUG

    // Debug Statement for Second Rectangle

    if (p.x == DEBUG_CANDIDATE_X && p.y == DEBUG_CANDIDATE_Y && haar_counter ==
0) {

        printf("[Device DEBUG] Rect 2: tl=(%d,%d), br=(%d,%d), sum2=%d,
weight=%d\n",

            tl2_x, tl2_y, br2_x, br2_y, sum2, d_weights_array[w_index + 1]);

    }

#endif
```

```
    int total_sum = sum1 + sum2;


    int sum3 = 0;


    // --- Third Rectangle (if present) ---
    if (d_weights_array[w_index + 2] != 0)
    {
        int tl3_x = p.x + (int)myRound_device(rect[8]);
        int tl3_y = p.y + (int)myRound_device(rect[9]);
        int br3_x = tl3_x + (int)myRound_device(rect[10]);
        int br3_y = tl3_y + (int)myRound_device(rect[11]);


#ifdef FINAL_DEBUG
        if (p.x == DEBUG_CANDIDATE_X && p.y == DEBUG_CANDIDATE_Y &&
            haar_counter == 0 && w_index == 0 && r_index == 0) {
            printf("[Device DEBUG] Third rectangle: tl=(%d,%d), br=(%d,%d)\n",
tl3_x, tl3_y, br3_x, br3_y);
        }
#endif


        assert(tl3_x >= 0 && tl3_x < d_cascade->sum.width);
        assert(tl3_y >= 0 && tl3_y < d_cascade->sum.height);
        assert(br3_x >= 0 && br3_x < d_cascade->sum.width);
        assert(br3_y >= 0 && br3_y < d_cascade->sum.height);
```

```
        int idx_tl3 = tl3_y * d_cascade->sum.width + tl3_x;

        int idx_tr3 = tl3_y * d_cascade->sum.width + br3_x;

        int idx_bl3 = br3_y * d_cascade->sum.width + tl3_x;

        int idx_br3 = br3_y * d_cascade->sum.width + br3_x;

        sum3 = d_cascade->p0[idx_br3] - d_cascade->p0[idx_tr3] -
d_cascade->p0[idx_bl3] + d_cascade->p0[idx_tl3];

        sum3 *= d_weights_array[w_index + 2];

        total_sum += sum3;



#ifdef FINAL_DEBUG

        // Debug Statement for Third Rectangle (only if it exists)

        if (p.x == DEBUG_CANDIDATE_X && p.y == DEBUG_CANDIDATE_Y &&
haar_counter == 0) {

            printf("[Device DEBUG] Rect 3: tl=(%d,%d), br=(%d,%d), sum3=%d,
weight=%d\n",

                tl3_x, tl3_y, br3_x, br3_y, sum3, d_weights_array[w_index +
2]);

        }

#endif



    }

    int threshold = d_tree_thresh_array[haar_counter] * (variance_norm_factor);



#ifdef FINAL_DEBUG
```

```
    // Debug only for the specific candidate and first few features
    if (p.x == DEBUG_CANDIDATE_X && p.y == DEBUG_CANDIDATE_Y && haar_counter <
5) {
        printf("[Device DEBUG] Candidate (%d,%d), Feature %d: sum1=%d, sum2=%d,
sum3=%d, total_sum=%d, threshold=%d\n",
            p.x, p.y, haar_counter, sum1, sum2, sum3, total_sum, threshold);
    }
#endif

    return (total_sum >= threshold) ? d_alpha2_array[haar_counter] :
d_alpha1_array[haar_counter];
}



__device__ int runCascadeClassifier_device(MyIntImage* d_sum, MyIntImage*
d_sqsum,
    const myCascade* d_cascade, MyPoint p, int start_stage, float scaleFactor)
{
    int width = d_cascade->sum.width;

    // Compute candidate offsets
    int p_offset = p.y * width + p.x;
    int pq_offset = p.y * d_cascade->sqsum.width + p.x;

    // Compute the integral image values at the four corners
```

```
    int top_left = d_cascade->p0[p_offset];
    int top_right = d_cascade->p1[p_offset];
    int bottom_left = d_cascade->p2[p_offset];
    int bottom_right = d_cascade->p3[p_offset];

    int mean = bottom_right - top_right - bottom_left + top_left;

    int sq_top_left = d_cascade->pq0[pq_offset];
    int sq_top_right = d_cascade->pq1[pq_offset];
    int sq_bottom_left = d_cascade->pq2[pq_offset];
    int sq_bottom_right = d_cascade->pq3[pq_offset];

    int var_norm = (sq_bottom_right - sq_top_right - sq_bottom_left +
sq_top_left);
    var_norm = (int)((var_norm * d_cascade->inv_window_area) - mean * mean);

    if (var_norm > 0)
        var_norm = int_sqrt_device(var_norm);
    else
        var_norm = 1;

#ifdef FINAL_DEBUG
    // Integral Debugging
    if (p.x == DEBUG_CANDIDATE_X && p.y == DEBUG_CANDIDATE_Y) {
```

```
             printf("\n-----------------------------------------------------\n");
        printf("[Device DEBUG] Candidate (%d,%d):\n", p.x, p.y);
        printf("[Device DEBUG] Integral corners: p0=%d, p1=%d, p2=%d, p3=%d\n",
            d_cascade->p0[p_offset], d_cascade->p1[p_offset],
d_cascade->p2[p_offset], d_cascade->p3[p_offset]);
        printf("[Device DEBUG] Squared integral corners: pq0=%d, pq1=%d,
pq2=%d, pq3=%d\n",
            d_cascade->pq0[pq_offset], d_cascade->pq1[pq_offset],
            d_cascade->pq2[pq_offset], d_cascade->pq3[pq_offset]);
        printf("[Device DEBUG] mean = %u, var_norm = %u\n", mean, var_norm);
        printf("-----------------------------------------------------\n\n");
    }
#endif


    int haar_counter = 0;
    int w_index = 0;
    int r_index = 0;
    float stage_sum = 0.0f;


    for (int i = start_stage; i < d_cascade->n_stages; i++) {
        stage_sum = 0.0f;
        int num_features = d_stages_array[i];


        for (int j = 0; j < num_features; j++) {
```

```
            int feature_result = evalWeakClassifier_device(d_cascade,
(int)var_norm, p,
                haar_counter, w_index, r_index, scaleFactor);
            stage_sum += feature_result;


            haar_counter++;
            w_index += 3;
            r_index += 12;
        }


#ifdef FINAL_DEBUG
        // Debugging after each stage:
        if (p.x == DEBUG_CANDIDATE_X && p.y == DEBUG_CANDIDATE_Y) {
            printf("[Device DEBUG] Stage %d complete: stage_sum = %f, threshold
= %f\n",
                i, stage_sum, 0.4f * d_stages_thresh_array[i]);
        }
#endif


        if (stage_sum < 0.4 * d_stages_thresh_array[i])
            return -i;
    }
    return 1;
}
```

```
// ------------------------------------------------------------------
// CUDA kernel: Each thread processes one candidate window.
__global__ void detectKernel(MyIntImage* d_sum, MyIntImage* d_sqsum,
    myCascade* d_cascade, float scaleFactor,
    int x_max, int y_max,
    MyRect* d_candidates, int* d_candidateCount,
    int maxCandidates)
{



    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int current_count = atomicAdd(&d_debug_print_count, 1);


#ifdef FINAL_DEBUG
    if (current_count < 10) {
        printf("[GPU Debug: detectKernel] x=%d, y=%d, x_max=%d, y_max=%d)\n",
            x, y, x_max, y_max);
    }
#endif
```

```
    // Debug: For the first thread in each block, print x and y
    if (threadIdx.x == 0 && threadIdx.y == 0) {
        //printf("[Device DEBUG] Block (%d,%d): first thread: x=%d, y=%d,
x_max=%d, y_max=%d\n",
            //blockIdx.x, blockIdx.y, x, y, x_max, y_max);
    }

    // Check the bounds unconditionally for a few threads
    if (x >= x_max || y >= y_max) {

#ifdef FINAL_DEBUG
        // debug print: iterations 4,5,6, limited to first 10 prints
        if (1) {
            // Atomically increment global print count and check limi
            if (current_count < 10) {
                printf("[GPU Debug: (x >= x_max || y >= y_max)] x=%d, y=%d,
x_max=%d, y_max=%d)\n",
                    x, y, x_max, y_max);
            }
        }
#endif
        return;
    }
```

```
#ifdef FINAL_DEBUG
    if (current_count == 2) {
        printf("\nMADE IT PAST if (x >= x_max || y >= y_max)\n");
    }
#endif

    MyPoint p;
    p.x = x;
    p.y = y;

#ifdef FINAL_DEBUG
    if (current_count == 2) {
        printf("\nMADE IT PAST p.x and p.y assignments\n");
    }
#endif

    int result = runCascadeClassifier_device(d_sum, d_sqsum, d_cascade, p, 0,
scaleFactor);

#ifdef FINAL_DEBUG
    if (current_count == 2) {
        printf("\nMADE IT PAST p.x and p.y runCascadeClassifier_device. \n
RESULT: %d\n\n", result);
```

```
    }
#endif


#ifdef FINAL_DEBUG
    if (result > 0) {
        printf("\nResult positiive!\n RESULT: %d\n\n", result);
    }
#endif


    if (result > 0) {
        MyRect r;
        r.x = (int)myRound_device(x * scaleFactor);
        r.y = (int)myRound_device(y * scaleFactor);

        r.width = (int)myRound_device(d_cascade->orig_window_size.width *
scaleFactor);
        r.height = (int)myRound_device(d_cascade->orig_window_size.height *
scaleFactor);
        int idx = atomicAdd(d_candidateCount, 1);
        if (idx < maxCandidates) {
            d_candidates[idx] = r;
        }

    }
```

```
}


// runDetection in cuda_detect.cu
std::vector<MyRect> runDetection(MyIntImage* h_sum, MyIntImage* h_sqsum,
    myCascade* cascade, int maxCandidates,
    float scaleFactor, int extra_x, int extra_y, int iter_counter)
{
    std::vector<MyRect> candidates;

    // --- Step 1: Allocate Unified Memory for sum integral image ---
    int dataSize = h_sum->width * h_sum->height * sizeof(int);
    MyIntImage* d_sum = nullptr;
    CUDA_CHECK(cudaMallocManaged((void**)&d_sum, sizeof(MyIntImage)));
    CUDA_CHECK(cudaMallocManaged((void**)&(d_sum->data), dataSize));
    CUDA_CHECK(cudaMemcpy(d_sum->data, h_sum->data, dataSize,
cudaMemcpyHostToDevice));
    d_sum->width = h_sum->width;
    d_sum->height = h_sum->height;

    // --- Step 2: Allocate Unified Memory for squared sum integral image ---
    dataSize = h_sqsum->width * h_sqsum->height * sizeof(int);
    MyIntImage* d_sqsum = nullptr;
    CUDA_CHECK(cudaMallocManaged((void**)&d_sqsum, sizeof(MyIntImage)));
    CUDA_CHECK(cudaMallocManaged((void**)&(d_sqsum->data), dataSize));
```

```
    CUDA_CHECK(cudaMemcpy(d_sqsum->data, h_sqsum->data, dataSize,
cudaMemcpyHostToDevice));

    d_sqsum->width = h_sqsum->width;

    d_sqsum->height = h_sqsum->height;


    // --- Step 3: Allocate Unified Memory for the cascade structure ---

    myCascade* d_cascade = nullptr;

    CUDA_CHECK(cudaMallocManaged((void**)&d_cascade, sizeof(myCascade)));

    *d_cascade = *cascade;  // copy host cascade to unified memory


    // --- Step 4: Update the cascade with unified memory pointers for integral
images ---

    d_cascade->sum = *d_sum;

    d_cascade->sqsum = *d_sqsum;

    d_cascade->sum.data = d_sum->data;   // Use unified memory data buffer of
d_sum

    d_cascade->sqsum.data = d_sqsum->data; // Use unified memory data buffer of
d_sqsum

    d_cascade->sum.width = d_sum->width;

    d_cascade->sum.height = d_sum->height;

    d_cascade->sqsum.width = d_sqsum->width;

    d_cascade->sqsum.height = d_sqsum->height;


    // Use the original window size for classification

    int winW = d_cascade->orig_window_size.width;
```

```
    int winH = d_cascade->orig_window_size.height;

    d_cascade->p0 = d_cascade->sum.data;

    d_cascade->p1 = d_cascade->sum.data + winW - 1;

    d_cascade->p2 = d_cascade->sum.data + d_cascade->sum.width * (winH - 1);

    d_cascade->p3 = d_cascade->sum.data + d_cascade->sum.width * (winH - 1) +
(winW - 1);



    d_cascade->pq0 = d_cascade->sqsum.data;

    d_cascade->pq1 = d_cascade->sqsum.data + winW - 1;

    d_cascade->pq2 = d_cascade->sqsum.data + d_cascade->sqsum.width * (winH -
1);

    d_cascade->pq3 = d_cascade->sqsum.data + d_cascade->sqsum.width * (winH -
1) + (winW - 1);



#ifdef FINAL_DEBUG

    printf("Cascade corner pointers:\n");

    printf(" p0 = %p\n", (void*)d_cascade->p0);

    printf(" p1 = %p (offset: %td)\n", (void*)d_cascade->p1, d_cascade->p1 -
d_cascade->sum.data);

    printf(" p2 = %p (offset: %td)\n", (void*)d_cascade->p2, d_cascade->p2 -
d_cascade->sum.data);

    printf(" p3 = %p (offset: %td)\n", (void*)d_cascade->p3, d_cascade->p3 -
d_cascade->sum.data);

    printf(" pq0 = %p\n", (void*)d_cascade->pq0);

    printf(" pq1 = %p (offset: %td)\n", (void*)d_cascade->pq1, d_cascade->pq1 -
d_cascade->sqsum.data);
```

```
    printf(" pq2 = %p (offset: %td)\n", (void*)d_cascade->pq2, d_cascade->pq2 -
d_cascade->sqsum.data);

    printf(" pq3 = %p (offset: %td)\n", (void*)d_cascade->pq3, d_cascade->pq3 -
d_cascade->sqsum.data);

#endif



    // --- Step 5: Transfer classifier parameters to device constant memory ---

    // (Allocate device memory for the classifier arrays and copy them from
host.)

    int* d_stages_array_dev = nullptr;

    CUDA_CHECK(cudaMalloc((void**)&d_stages_array_dev, cascade->n_stages *
sizeof(int)));

    CUDA_CHECK(cudaMemcpy(d_stages_array_dev, cascade->stages_array,
cascade->n_stages * sizeof(int), cudaMemcpyHostToDevice));



    float* d_stages_thresh_array_dev = nullptr;

    CUDA_CHECK(cudaMalloc((void**)&d_stages_thresh_array_dev, cascade->n_stages
* sizeof(float)));

    CUDA_CHECK(cudaMemcpy(d_stages_thresh_array_dev,
cascade->stages_thresh_array, cascade->n_stages * sizeof(float),
cudaMemcpyHostToDevice));



    int* d_rectangles_array_dev = nullptr;

    CUDA_CHECK(cudaMalloc((void**)&d_rectangles_array_dev, cascade->total_nodes
* 12 * sizeof(int)));

    CUDA_CHECK(cudaMemcpy(d_rectangles_array_dev, cascade->rectangles_array,
cascade->total_nodes * 12 * sizeof(int), cudaMemcpyHostToDevice));
```

```
    int* d_weights_array_dev = nullptr;

    CUDA_CHECK(cudaMalloc((void**)&d_weights_array_dev, cascade->total_nodes *
3 * sizeof(int)));

    CUDA_CHECK(cudaMemcpy(d_weights_array_dev, cascade->weights_array,
cascade->total_nodes * 3 * sizeof(int), cudaMemcpyHostToDevice));



    int* d_alpha1_array_dev = nullptr;

    CUDA_CHECK(cudaMalloc((void**)&d_alpha1_array_dev, cascade->total_nodes *
sizeof(int)));

    CUDA_CHECK(cudaMemcpy(d_alpha1_array_dev, cascade->alpha1_array,
cascade->total_nodes * sizeof(int), cudaMemcpyHostToDevice));



    int* d_alpha2_array_dev = nullptr;

    CUDA_CHECK(cudaMalloc((void**)&d_alpha2_array_dev, cascade->total_nodes *
sizeof(int)));

    CUDA_CHECK(cudaMemcpy(d_alpha2_array_dev, cascade->alpha2_array,
cascade->total_nodes * sizeof(int), cudaMemcpyHostToDevice));



    int* d_tree_thresh_array_dev = nullptr;

    CUDA_CHECK(cudaMalloc((void**)&d_tree_thresh_array_dev,
cascade->total_nodes * sizeof(int)));

    CUDA_CHECK(cudaMemcpy(d_tree_thresh_array_dev, cascade->tree_thresh_array,
cascade->total_nodes * sizeof(int), cudaMemcpyHostToDevice));



    CUDA_CHECK(cudaMemcpyToSymbol(d_stages_array, &d_stages_array_dev,
sizeof(int*)));

    CUDA_CHECK(cudaMemcpyToSymbol(d_stages_thresh_array,
&d_stages_thresh_array_dev, sizeof(float*)));
```

```
    CUDA_CHECK(cudaMemcpyToSymbol(d_rectangles_array, &d_rectangles_array_dev,
sizeof(int*)));

    CUDA_CHECK(cudaMemcpyToSymbol(d_weights_array, &d_weights_array_dev,
sizeof(int*)));

    CUDA_CHECK(cudaMemcpyToSymbol(d_alpha1_array, &d_alpha1_array_dev,
sizeof(int*)));

    CUDA_CHECK(cudaMemcpyToSymbol(d_alpha2_array, &d_alpha2_array_dev,
sizeof(int*)));

    CUDA_CHECK(cudaMemcpyToSymbol(d_tree_thresh_array,
&d_tree_thresh_array_dev, sizeof(int*)));



#ifdef FINAL_DEBUG

    printf("[Host DEBUG] Transferred classifier parameters to device constant
memory.\n");

#endif



    // --- Step 6: Allocate Unified Memory for detection results ---

    MyRect* d_candidates = nullptr;

    CUDA_CHECK(cudaMallocManaged((void**)&d_candidates, maxCandidates *
sizeof(MyRect)));

    int* d_candidateCount = nullptr;

    CUDA_CHECK(cudaMallocManaged((void**)&d_candidateCount, sizeof(int)));

    *d_candidateCount = 0;



#ifdef FINAL_DEBUG

    printf("[Host DEBUG] d_candidates allocated at %p, d_candidateCount
allocated at %p, initial candidate count = %d\n",
```

```
        (void*)d_candidates, (void*)d_candidateCount, *d_candidateCount);
#endif

    // --- Step 7: Determine search space dimensions and launch the detection
kernel ---
    // Use extra_x and extra_y only to clip the search space so that the
sliding window remains in bounds.
    // The classifier window size remains the original size
(cascade->orig_window_size) * scaleFactor.
    int baseWidth = cascade->orig_window_size.width;
    int baseHeight = cascade->orig_window_size.height;

    // Compute maximum valid starting positions for the sliding window.
    int x_max = d_sum->width - baseWidth;
    int y_max = d_sum->height - baseHeight;
    if (x_max < 0) x_max = 0;
    if (y_max < 0) y_max = 0;

#ifdef FINAL_DEBUG
    printf("[Host DEBUG] Search space dimensions (with extra margins):
x_max=%d, y_max=%d\n", x_max, y_max);
#endif

    dim3 blockDim(16, 16);
    dim3 gridDim((x_max + blockDim.x - 1) / blockDim.x,
```

```
        (y_max + blockDim.y - 1) / blockDim.y);


#ifdef FINAL_DEBUG
    printf("[Host] Launching kernel with gridDim=(%d, %d), blockDim=(%d,
%d)\n",
        gridDim.x, gridDim.y, blockDim.x, blockDim.y);
#endif


#ifdef FINAL_DEBUG
    printf("\nDetection window: %d x %d, x_max=%d, y_max=%d\n",
        baseWidth, baseHeight, x_max, y_max);
#endif




    // Launch the kernel with the original base window size for classification.
    CUDA_CHECK(cudaDeviceSynchronize());


#ifdef FINAL_DEBUG


printf("\n---------------------------------------------------------------------
----------\n");
    printf("[Kernel Launch Debug] scaleFactor = %.3f, x_max = %d, y_max = %d,
maxCandidates = %d\n",
        scaleFactor, x_max, y_max, maxCandidates);
    printf("[Kernel Launch Debug] d_sum = %p, d_sqsum = %p, d_cascade = %p,
d_candidates = %p, d_candidateCount = %p\n",
```

```
        (void*)d_sum, (void*)d_sqsum, (void*)d_cascade, (void*)d_candidates,
(void*)d_candidateCount);

    printf("[Kernel Launch Debug] gridDim = (%d, %d), blockDim = (%d, %d)\n",

        gridDim.x, gridDim.y, blockDim.x, blockDim.y);


printf("\n----------------------------------------------------------------------
----------\n");

#endif



    int zero = 0;

    CUDA_CHECK(cudaMemcpyToSymbol(d_debug_print_count, &zero, sizeof(int)));



    detectKernel << <gridDim, blockDim >> > (d_sum, d_sqsum, d_cascade,
scaleFactor, x_max, y_max, d_candidates, d_candidateCount, maxCandidates);

    CUDA_CHECK(cudaGetLastError());

   // printf("[Host DEBUG] Kernel launched.\n");

    CUDA_CHECK(cudaDeviceSynchronize());

   // printf("[Host DEBUG] Kernel execution completed.\n");



    int hostCandidateCount = 0;

    CUDA_CHECK(cudaMemcpy(&hostCandidateCount, d_candidateCount, sizeof(int),
cudaMemcpyDeviceToHost));

   // printf("[Host] Detected %d candidate windows.\n", hostCandidateCount);

    for (int i = 0; i < hostCandidateCount; i++) {

        candidates.push_back(d_candidates[i]);

    }
```

```
   // printf("[Host DEBUG] Cleaning up Unified Memory and device memory
allocated with cudaMalloc.\n");

    cudaFree(d_candidates);

    cudaFree(d_candidateCount);

    cudaFree(d_cascade);

    cudaFree(d_sum->data);

    cudaFree(d_sum);

    cudaFree(d_sqsum->data);

    cudaFree(d_sqsum);

    cudaFree(d_stages_array_dev);

    cudaFree(d_stages_thresh_array_dev);

    cudaFree(d_rectangles_array_dev);

    cudaFree(d_weights_array_dev);

    cudaFree(d_alpha1_array_dev);

    cudaFree(d_alpha2_array_dev);

    cudaFree(d_tree_thresh_array_dev);


   // printf("[Host] runDetection() completed.\n");

    return candidates;

}
```

Appendix C: cuda_detect.cuh

C/C++

```
//
============================================================================
===============

// Filename: cuda_detect.h

//

// Description:

//     Header file declaring the interface for CUDA-based Viola-Jones detection
functions.

//     It includes necessary CUDA and project-specific headers, defines
conditional compilation

//     for C++, and provides the runDetection function declaration.

//

// Date: 03.12.25

// Authors: Ibrahim Binmahfood, Kunjan Vyas, Robert Wilcox

//
----------------------------------------------------------------------------



#ifndef CUDA_DETECT_H

#define CUDA_DETECT_H



#include "haar_cuda.h"

#include "image_cuda.h"

#include <cuda_runtime.h>

#include <device_launch_parameters.h>
```

```
#include <string.h>

#include <assert.h>


#ifdef __cplusplus

#include <vector>


// runDetection launches the CUDA detection kernel using the host-side

// integral images and cascade classifier. It transfers data to the GPU,

// launches the kernel, retrieves results, cleans up device memory,

// and returns a std::vector<MyRect> containing candidate detections.

// Parameters:

//   h_sum       - pointer to the host MyIntImage for the integral image

//   h_sqsum     - pointer to the host MyIntImage for the squared integral
image

//   cascade     - pointer to the host cascade classifier (after
setImageForCascadeClassifier)

//   maxCandidates - maximum number of candidate detections allocated on the
device

//   scaleFactor - current scale factor (e.g., 1.0f)

std::vector<MyRect> runDetection(MyIntImage* h_sum, MyIntImage* h_sqsum,

        myCascade* cascade,

        int maxCandidates,

        float scaleFactor,

        int extra_x,

        int extra_y,
```

```
        int iter_counter);
#endif // __cplusplus


#endif // CUDA_DETECT_H
```

Appendix D: haar_cuda.cpp

C/C++

```
/*
 *  TU Eindhoven
 *  Eindhoven, The Netherlands
 *
 *  Name            :   haar.cpp
 *
 *  Author          :   Francesco Comaschi (f.comaschi@tue.nl)
 *
 *  Date            :   November 12, 2012
 *
 *  Function        :   Haar features evaluation for face detection
 *
 *  History         :
 *      12-11-12    :   Initial version.
 *
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program;  If not, see <http://www.gnu.org/licenses/>
 *
 * In other words, you are welcome to use, share and improve this program.
 * You are forbidden to forbid anyone else to use, share and improve
 * what you give them.   Happy coding!
 *
 * Modified 03.12.25 by: Ibrahim Binmahfood, Kunjan Vyas, Robert Wilcox
 * to include CUDA code for parallel processing
 */


#include "haar_cuda.h"
#include <cmath>


#define DEBUG_PRINT 1
///#define FINAL_DEBUG


/* TODO: use matrices */
/* classifier parameters */
```

```
/***********************************
 * Notes:
 * To paralleism the filter,
 * these monolithic arrays may
 * need to be splitted or duplicated
 ***********************************/


int clock_counter = 0;
float n_features = 0;




int iter_counter = 0;


/* compute integral images */
void integralImages( MyImage *src, MyIntImage *sum, MyIntImage *sqsum );


/* scale down the image */
void ScaleImage_Invoker( myCascade* _cascade, float _factor, int sum_row, int
sum_col, std::vector<MyRect>& _vec);


/* compute scaled image */
void nearestNeighbor (MyImage *src, MyImage *dst);


/* rounding function */
```

```
inline  int  myRound( float value )
{
       return std::lroundf(value);
}


/*****************************************************
 * Function: detectObjects
 * Description: It calls all the major steps
 *****************************************************/


std::vector<MyRect> detectObjects( MyImage* _img, MySize minSize, MySize
maxSize, myCascade* cascade,
                              float scaleFactor, int minNeighbors)
{


  /* group overlaping windows */
  const float GROUP_EPS = 0.4f;
  /* pointer to input image */
  MyImage *img = _img;
  /**********************************
   * create structs for images
   * see haar.h for details
   * img1: normal image (unsigned char)
   * sum1: integral image (int)
```

```
 * sqsum1: square integral image (int)
 *********************************/
MyImage image1Obj;
MyIntImage sum1Obj;
MyIntImage sqsum1Obj;
/* pointers for the created structs */
MyImage *img1 = &image1Obj;
MyIntImage *sum1 = &sum1Obj;
MyIntImage *sqsum1 = &sqsum1Obj;


/*******************************************************
 * allCandidates is the preliminaray face candidate,
 * which will be refined later.
 *
 * std::vector is a sequential container
 * http://en.wikipedia.org/wiki/Sequence_container_(C++)
 *
 * Each element of the std::vector is a "MyRect" struct
 * MyRect struct keeps the info of a rectangle (see haar.h)
 * The rectangle contains one face candidate
 *****************************************************/
std::vector<MyRect> allCandidates;
```

```
/* scaling factor */
float factor;

/* maxSize */
if( maxSize.height == 0 || maxSize.width == 0 )
  {
    maxSize.height = img->height;
    maxSize.width = img->width;
  }

/* window size of the training set */
MySize winSize0 = cascade->orig_window_size;

/* malloc for img1: unsigned char */
createImage(img->width, img->height, img1);
/* malloc for sum1: unsigned char */
createSumImage(img->width, img->height, sum1);
/* malloc for sqsum1: unsigned char */
createSumImage(img->width, img->height, sqsum1);

/* initial scaling factor */
factor = 1;
```

```
  /* iterate over the image pyramid */

  for( factor = 1; ; factor *= scaleFactor )

    {

      /* iteration counter */

      iter_counter++;



      /* size of the image scaled up */

         MySize winSize;

         winSize.width = myRound(winSize0.width * factor);

         winSize.height = myRound(winSize0.height * factor);



      /* size of the image scaled down (from bigger to smaller) */

      MySize sz = { myRound( img->width/factor ), myRound( img->height/factor )
};



      /* difference between sizes of the scaled image and the original
detection window */

      MySize sz1 = { sz.width - winSize0.width, sz.height - winSize0.height };



      /* if the actual scaled image is smaller than the original detection
window, break */

      if( sz1.width < 0 || sz1.height < 0 )

       break;
```

```
      /* if a minSize different from the original detection window is
specified, continue to the next scaling */
      if( winSize.width < minSize.width || winSize.height < minSize.height )
       continue;

      /************************************
       * Set the width and height of
       * img1: normal image (unsigned char)
       * sum1: integral image (int)
       * sqsum1: squared integral image (int)
       * see image.c for details
       ************************************/
      setImage(sz.width, sz.height, img1);
      setSumImage(sz.width, sz.height, sum1);
      setSumImage(sz.width, sz.height, sqsum1);

      /***************************************
       * Compute-intensive step:
       * building image pyramid by downsampling
       * downsampling using nearest neighbor
       ***************************************/
      nearestNeighbor(img, img1);

      /*************************************************
```

```
         * Compute-intensive step:
         * At each scale of the image pyramid,
         * compute a new integral and squared integral image
         ***************************************************/
        integralImages(img1, sum1, sqsum1);

        /* sets images for haar classifier cascade */
        /***************************************************
         * Note:
         * Summing pixels within a haar window is done by
         * using four corners of the integral image:
         * http://en.wikipedia.org/wiki/Summed_area_table
         *
         * This function loads the four corners,
         * but does not do compuation based on four coners.
         * The computation is done next in ScaleImage_Invoker
         ***************************************************/
        setImageForCascadeClassifier( cascade, sum1, sqsum1);

#ifdef FINAL_DEBUG
        /* print out for each scale of the image pyramid */
        printf("detecting faces, iter := %d\n", iter_counter);
#endif
```

```
      /*************************************************
       * Process the current scale with the cascaded fitler.
       * The main computations are invoked by this function.
       * Optimization oppurtunity:
       * the same cascade filter is invoked each time
       *************************************************/
      ScaleImage_Invoker(cascade, factor, sum1->height, sum1->width,
                         allCandidates);
    } /* end of the factor loop, finish all scales in pyramid*/

  if( minNeighbors != 0)
    {
      groupRectangles(allCandidates, minNeighbors, GROUP_EPS);
    }

  freeImage(img1);
  freeSumImage(sum1);
  freeSumImage(sqsum1);
  return allCandidates;

}
```

```
/*********************************************
 * Note:
 * The int_sqrt is softwar integer squre root.
 * GPU has hardware for floating squre root (sqrtf).
 * In GPU, it is wise to convert an int variable
 * into floating point, and use HW sqrtf function.
 * More info:
 *
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#standard-functi
ons
 *********************************************/
/*****************************************************
 * The int_sqrt is only used in runCascadeClassifier
 * If you want to replace int_sqrt with HW sqrtf in GPU,
 * simple look into the runCascadeClassifier function.
 *****************************************************/
unsigned int int_sqrt (unsigned int value)
{
  int i;
  unsigned int a = 0, b = 0, c = 0;
  for (i=0; i < (32 >> 1); i++)
    {
      c<<= 2;
#define UPPERBITS(value) (value>>30)
```

```
      c += UPPERBITS(value);
#undef UPPERBITS
      value <<= 2;
      a <<= 1;
      b = (a<<1) | 1;
      if (c >= b)
       {
         c -= b;
         a++;
       }
    }
  return a;
}



void setImageForCascadeClassifier(myCascade* _cascade, MyIntImage* _sum,
MyIntImage* _sqsum)
{

#ifdef FINAL_DEBUG
    printf("\n-- Entering setImageForCascadeClassifier --\n");
#endif

    MyIntImage* sum = _sum;
```

```
    MyIntImage* sqsum = _sqsum;
    myCascade* cascade = _cascade;
    int i, j, k;
    int r_index = 0;
    int w_index = 0;
    MyRect tr;

    // Assign the integral image structures to the cascade.
    cascade->sum = *sum;
    cascade->sqsum = *sqsum;

    // Use the original window size for the filter rectangle.
    MyRect equRect;
    equRect.x = equRect.y = 0;
    equRect.width = cascade->orig_window_size.width;
    equRect.height = cascade->orig_window_size.height;

#ifdef FINAL_DEBUG
    // Print the original window size.
    printf("DEBUG: cascade->orig_window_size.width = %d,
cascade->orig_window_size.height = %d\n",
        cascade->orig_window_size.width, cascade->orig_window_size.height);
#endif
```

```
    // Check for zero dimensions.
    if (equRect.width == 0 || equRect.height == 0)
    {
       // printf("ERROR: Detection window has zero dimension(s): width=%d,
height=%d\n", equRect.width, equRect.height);
    }

    int window_area = equRect.width * equRect.height;

#ifdef FINAL_DEBUG
    printf("DEBUG: Calculated window area = %d\n", window_area);
#endif

#ifdef FINAL_DEBUG
    printf("\n-- Calculating inverse window area --\n");
#endif
    // Assuming cascade->inv_window_area is a float.
    cascade->inv_window_area = 1.0f / window_area;
    // Cast to double when printing with %f (printf promotes float to double
anyway)

#ifdef FINAL_DEBUG
    printf("DEBUG: cascade->inv_window_area = %f\n",
(double)cascade->inv_window_area);
#endif
```

```
#ifdef FINAL_DEBUG

    printf("\n-- Setting integral image corner pointers in cascade --\n");

#endif



    cascade->p0 = sum->data;                                        //
Top-left

    cascade->p1 = sum->data + equRect.width - 1;                      //
Top-right

    cascade->p2 = sum->data + sum->width * (equRect.height - 1);          //
Bottom-left

    cascade->p3 = sum->data + sum->width * (equRect.height - 1) +
(equRect.width - 1); // Bottom-right



    cascade->pq0 = sqsum->data;

    cascade->pq1 = sqsum->data + equRect.width - 1;

    cascade->pq2 = sqsum->data + sqsum->width * (equRect.height - 1);

    cascade->pq3 = sqsum->data + sqsum->width * (equRect.height - 1) +
(equRect.width - 1);



    /****************************************
     * Process the classifier parameters
     * for each stage and feature.
     ****************************************/

#ifdef FINAL_DEBUG

    printf("\n-- Starting stage loop in setImageForCascadeClassifier --\n");
```

```
#endif


    for (i = 0; i < cascade->n_stages; i++)
    {
        //printf("  -- Stage: %d --\n", i);
        for (j = 0; j < cascade->stages_array[i]; j++)
        {
            //printf("    -- Feature: %d --\n", j);
            int nr = 3;  // Number of rectangles per feature
            for (k = 0; k < nr; k++)
            {
                //printf("      -- Rectangle: %d --\n", k);
                // Read the rectangle parameters from the classifier array.
                tr.x = cascade->rectangles_array[r_index + k * 4];
                //printf("      -- tr.x = %d; --\n", tr.x);
                tr.y = cascade->rectangles_array[r_index + 1 + k * 4];
                //printf("      -- tr.y = %d; --\n", tr.y);
                tr.width = cascade->rectangles_array[r_index + 2 + k * 4];
                //printf("      -- tr.width = %d; --\n", tr.width);
                tr.height = cascade->rectangles_array[r_index + 3 + k * 4];
                //printf("      -- tr.height = %d; --\n", tr.height);

                // Set up the scaled rectangle pointers.
```

```
                // For the first two rectangles, always compute the pointer.
                if (k < 2)
                {
                    cascade->scaled_rectangles_array[r_index + k * 4] =
                        (int*)(sum->data + sum->width * tr.y + tr.x);
                    cascade->scaled_rectangles_array[r_index + k * 4 + 1] =
                        (int*)(sum->data + sum->width * tr.y + (tr.x +
tr.width));
                    cascade->scaled_rectangles_array[r_index + k * 4 + 2] =
                        (int*)(sum->data + sum->width * (tr.y + tr.height) +
tr.x);
                    cascade->scaled_rectangles_array[r_index + k * 4 + 3] =
                        (int*)(sum->data + sum->width * (tr.y + tr.height) +
(tr.x + tr.width));
                }
                else
                {
                    // For the third rectangle, check if it is used.
                    if ((tr.x == 0) && (tr.y == 0) && (tr.width == 0) &&
(tr.height == 0))
                    {
                        cascade->scaled_rectangles_array[r_index + k * 4] =
NULL;
                        cascade->scaled_rectangles_array[r_index + k * 4 + 1] =
NULL;
```

```
                        cascade->scaled_rectangles_array[r_index + k * 4 + 2] =
NULL;
                        cascade->scaled_rectangles_array[r_index + k * 4 + 3] =
NULL;
                    }
                    else
                    {
                        cascade->scaled_rectangles_array[r_index + k * 4] =
                            (int*)(sum->data + sum->width * tr.y + tr.x);
                        cascade->scaled_rectangles_array[r_index + k * 4 + 1] =
                            (int*)(sum->data + sum->width * tr.y + (tr.x +
tr.width));
                        cascade->scaled_rectangles_array[r_index + k * 4 + 2] =
                            (int*)(sum->data + sum->width * (tr.y + tr.height)
+ tr.x);
                        cascade->scaled_rectangles_array[r_index + k * 4 + 3] =
                            (int*)(sum->data + sum->width * (tr.y + tr.height)
+ (tr.x + tr.width));
                    }
                }
            }
            //printf("    -- Finished processing feature, updating indices
--\n");
            r_index += 12;  // 3 rectangles × 4 parameters each
            w_index += 3;   // 3 weights per feature
        }
```

```
        //printf("  -- Finished stage %d --\n", i);
    }
#ifdef FINAL_DEBUG
    printf("\n-- Exiting setImageForCascadeClassifier --\n");
#endif
}



/***************************************************
 * evalWeakClassifier:
 * the actual computation of a haar filter.
 * More info:
 * http://en.wikipedia.org/wiki/Haar-like_features
 ***************************************************/
inline int evalWeakClassifier(myCascade* cascade, int variance_norm_factor, int
p_offset, int tree_index, int w_index, int r_index )
{

  /* the node threshold is multiplied by the standard deviation of the image */
  int t = cascade->tree_thresh_array[tree_index] * variance_norm_factor;

  int sum = (*(cascade->scaled_rectangles_array[r_index] + p_offset)
      - *(cascade->scaled_rectangles_array[r_index + 1] + p_offset)
```

```
      - *(cascade->scaled_rectangles_array[r_index + 2] + p_offset)
      + *(cascade->scaled_rectangles_array[r_index + 3] + p_offset))
      * cascade->weights_array[w_index];

//printf("sum1: %d\n",sum);

  sum += (*(cascade->scaled_rectangles_array[r_index + 4] + p_offset)
      - *(cascade->scaled_rectangles_array[r_index + 5] + p_offset)
      - *(cascade->scaled_rectangles_array[r_index + 6] + p_offset)
      + *(cascade->scaled_rectangles_array[r_index + 7] + p_offset))
      * cascade->weights_array[w_index + 1];

//printf("sum2: %d\n",sum);

  if (cascade->scaled_rectangles_array[r_index + 8] != NULL)
      sum += (*(cascade->scaled_rectangles_array[r_index + 8] + p_offset)
          - *(cascade->scaled_rectangles_array[r_index + 9] + p_offset)
          - *(cascade->scaled_rectangles_array[r_index + 10] + p_offset)
          + *(cascade->scaled_rectangles_array[r_index + 11] + p_offset))
      * cascade->weights_array[w_index + 2];

//printf("sum3: %d\n",sum);
```

```
  if(sum >= t)
    return cascade->alpha2_array[tree_index];
  else
    return cascade->alpha1_array[tree_index];

}




int runCascadeClassifier( myCascade* _cascade, MyPoint pt, int start_stage )
{

  int p_offset, pq_offset;
  int i, j;
  unsigned int mean;
  unsigned int variance_norm_factor;
  int haar_counter = 0;
  int w_index = 0;
  int r_index = 0;
  int stage_sum;
  myCascade* cascade;
  cascade = _cascade;
```

```
  p_offset = pt.y * (cascade->sum.width) + pt.x;

  pq_offset = pt.y * (cascade->sqsum.width) + pt.x;


  /***************************************************************************

   * Image normalization

   * mean is the mean of the pixels in the detection window

   * cascade->pqi[pq_offset] are the squared pixel values (using the squared
integral image)

   * inv_window_area is 1 over the total number of pixels in the detection
window

   ***************************************************************************/


  variance_norm_factor =  (cascade->pq0[pq_offset] - cascade->pq1[pq_offset] -
cascade->pq2[pq_offset] + cascade->pq3[pq_offset]);

  mean = (cascade->p0[p_offset] - cascade->p1[p_offset] - cascade->p2[p_offset]
+ cascade->p3[p_offset]);


  variance_norm_factor = (variance_norm_factor*cascade->inv_window_area);

  variance_norm_factor =  variance_norm_factor - mean*mean;


  /*********************************************

   * Note:

   * The int_sqrt is softwar integer squre root.

   * GPU has hardware for floating squre root (sqrtf).

   * In GPU, it is wise to convert the variance norm
```

```
   * into floating point, and use HW sqrtf function.
   * More info:
   *
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#standard-functi
ons
   ********************************************/
  if( variance_norm_factor > 0 )
    variance_norm_factor = int_sqrt(variance_norm_factor);
  else
    variance_norm_factor = 1;

  /***********************************************
   * The major computation happens here.
   * For each scale in the image pyramid,
   * and for each shifted step of the filter,
   * send the shifted window through cascade filter.
   *
   * Note:
   *
   * Stages in the cascade filter are independent.
   * However, a face can be rejected by any stage.
   * Running stages in parallel delays the rejection,
   * which induces unnecessary computation.
   *
```

```
   * Filters in the same stage are also independent,
   * except that filter results need to be merged,
   * and compared with a per-stage threshold.
   ************************************************/
  for( i = start_stage; i < cascade->n_stages; i++ )
    {

      /****************************************************
       * A shared variable that induces false dependency
       *
       * To avoid it from limiting parallelism,
       * we can duplicate it multiple times,
       * e.g., using stage_sum_array[number_of_threads].
       * Then threads only need to sync at the end
       ****************************************************/
      stage_sum = 0;

      for( j = 0; j < cascade->stages_array[i]; j++ )
       {
         /***************************************************
          * Send the shifted window to a haar filter.
          ***************************************************/
         stage_sum += evalWeakClassifier(cascade, variance_norm_factor,
p_offset, haar_counter, w_index, r_index);
```

```
          n_features++;

          haar_counter++;

          w_index+=3;

          r_index+=12;

        } /* end of j loop */


       /**************************************************************

        * threshold of the stage.

        * If the sum is below the threshold,

        * no faces are detected,

        * and the search is abandoned at the i-th stage (-i).

        * Otherwise, a face is detected (1)

        **************************************************************/


       /* the number "0.4" is empirically chosen for 5kk73 */

       if( stage_sum < 0.4 * cascade->stages_thresh_array[i] ){

        return -i;

       } /* end of the per-stage thresholding */

     } /* end of i loop */

   return 1;

}
```

```
void ScaleImage_Invoker( myCascade* _cascade, float _factor, int sum_row, int
sum_col, std::vector<MyRect>& _vec)

{


  myCascade* cascade = _cascade;


  float factor = _factor;

  MyPoint p;

  int result;

  int y1, y2, x2, x, y, step;

  std::vector<MyRect> *vec = &_vec;


  MySize winSize0 = cascade->orig_window_size;

  MySize winSize;


  winSize.width =  myRound(winSize0.width*factor);

  winSize.height =  myRound(winSize0.height*factor);

  y1 = 0;


  /*****************************************

  * When filter window shifts to image boarder,

  * some margin need to be kept

  *****************************************/

  y2 = sum_row - winSize0.height;
```

```
x2 = sum_col - winSize0.width;


/*****************************************
 * Step size of filter window shifting
 * Reducing step makes program faster,
 * but decreases quality of detection.
 * example:
 * step = factor > 2 ? 1 : 2;
 *
 * For 5kk73,
 * the factor and step can be kept constant,
 * unless you want to change input image.
 *
 * The step size is set to 1 for 5kk73,
 * i.e., shift the filter window by 1 pixel.
 *****************************************/
step = 1;


/*****************************************
 * Shift the filter window over the image.
 * Each shift step is independent.
 * Shared data structure may limit parallelism.
 *
```

```
   * Some random hints (may or may not work):
   * Split or duplicate data structure.
   * Merge functions/loops to increase locality
   * Tiling to increase computation-to-memory ratio
   ********************************************/
  for( x = 0; x <= x2-1; x += step )        //changed x <= x2 ...to... x <=
x2-1
    for( y = y1; y <= y2-1; y += step )     //changed y <= y2 ...to... y <=
y2-1
      {
       p.x = x;
       p.y = y;

       /********************************************
        * Optimization Oppotunity:
        * The same cascade filter is used each time
        ********************************************/
       result = runCascadeClassifier( cascade, p, 0 );

       /*****************************************************
        * If a face is detected,
        * record the coordinates of the filter window
        * the "push_back" function is from std:vec, more info:
        * http://en.wikipedia.org/wiki/Sequence_container_(C++)
```

```
        *
        * Note that, if the filter runs on GPUs,
        * the push_back operation is not possible on GPUs.
        * The GPU may need to use a simpler data structure,
        * e.g., an array, to store the coordinates of face,
        * which can be later memcpy from GPU to CPU to do push_back
        *****************************************************/
       if( result > 0 )
         {
           MyRect r = {myRound(x*factor), myRound(y*factor), winSize.width,
winSize.height};
           vec->push_back(r);
         }
      }
}


/***************************************************
 * Compute the integral image (and squared integral)
 * Integral image helps quickly sum up an area.
 * More info:
 * http://en.wikipedia.org/wiki/Summed_area_table
 ****************************************************/
void integralImages( MyImage *src, MyIntImage *sum, MyIntImage *sqsum )
{
```

```
int x, y, s, sq, t, tq;
unsigned char it;
int height = src->height;
int width = src->width;
unsigned char *data = src->data;
int * sumData = sum->data;
int * sqsumData = sqsum->data;
for( y = 0; y < height; y++)
  {
    s = 0;
    sq = 0;
    /* loop over the number of columns */
    for( x = 0; x < width; x ++)
     {
       it = data[y*width+x];
       /* sum of the current row (integer)*/
       s += it;
       sq += it*it;

       t = s;
       tq = sq;
       if (y != 0)
         {
```

```
            t += sumData[(y-1)*width+x];
            tq += sqsumData[(y-1)*width+x];
          }
        sumData[y*width+x]=t;
        sqsumData[y*width+x]=tq;
      }
    }
}

/*************************************************************
 * This function downsample an image using nearest neighbor
 * It is used to build the image pyramid
 *************************************************************/
void nearestNeighbor(MyImage* src, MyImage* dst) {
    int i, j, x, y, rat;
    unsigned char* t;
    unsigned char* p;
    int w1 = src->width;
    int h1 = src->height;
    int w2 = dst->width;
    int h2 = dst->height;

#ifdef FINAL_DEBUG
```

```
    printf("In nearestNeighbor: src->data = %p, dst->data = %p, w1=%d, h1=%d,
w2=%d, h2=%d\n",
        src->data, dst->data, w1, h1, w2, h2);
#endif



    if (w2 <= 0 || h2 <= 0) {
#ifdef FINAL_DEBUG
        printf("Destination dimensions invalid: w2=%d, h2=%d\n", w2, h2);
#endif
        return;
    }


    unsigned char* src_data = src->data;
    unsigned char* dst_data = dst->data;


    int x_ratio = (int)((w1 << 16) / w2) + 1;
    int y_ratio = (int)((h1 << 16) / h2) + 1;


    for (i = 0; i < h2; i++) {
        t = dst_data + i * w2;
        y = ((i * y_ratio) >> 16);
        if (y < 0 || y >= h1) {
#ifdef FINAL_DEBUG
```

```
            printf("Invalid y = %d at iteration i=%d\n", y, i);
#endif
            y = (y < 0) ? 0 : h1 - 1;
        }
        p = src_data + y * w1;
        rat = 0;
        for (j = 0; j < w2; j++) {
            x = (rat >> 16);
            if (x >= w1) {  // Ensure x is within bounds
                x = w1 - 1;
            }
            *t++ = p[x];
            rat += x_ratio;
        }
    }
}



void readTextClassifier(myCascade* cascade) // Modified function to accept
myCascade*
{
    /*number of stages of the cascade classifier*/
    int stages;
    /*total number of weak classifiers (one node each)*/
```

```
    int total_nodes = 0;
    int i, j, k, l;
    char mystring[12];
    int r_index = 0;
    int w_index = 0;
    int tree_index = 0;
    FILE* finfo = fopen("info.txt", "rb");


#ifdef FINAL_DEBUG
    printf("\n-- Entering readTextClassifier --\n");
#endif




    if (finfo == NULL) { // Check if file opened successfully
#ifdef FINAL_DEBUG
        printf("Error opening info.txt!\n");
#endif
        return; // Exit if file not opened
    }
#ifdef FINAL_DEBUG
    printf("Successfully opened info.txt\n"); // Added print
#endif
```

```
    /************************************************
    * how many stages are in the cascaded filter?
    * the first line of info.txt is the number of stages
    * (in the 5kk73 example, there are 25 stages)
    ************************************************/
    if (fgets(mystring, 12, finfo) != NULL)
    {
        stages = atoi(mystring);
#ifdef FINAL_DEBUG
        printf("Number of stages read: %d\n", stages); // Added print
#endif
    }
    else {
#ifdef FINAL_DEBUG
        printf("Error reading number of stages from info.txt!\n");
#endif
        fclose(finfo);
        return;
    }
    i = 0;

    int* stages_array = (int*)malloc(sizeof(int) * stages); // Local variable,
not static global
```

```
    cascade->stages_array = stages_array;                // Assign to cascade
struct member




    /***************************************************
     * how many filters in each stage?
     * They are specified in info.txt,
     * starting from second line.
     * (in the 5kk73 example, from line 2 to line 26)
     ***************************************************/
#ifdef FINAL_DEBUG
    printf("\n-- Reading stages array from info.txt --\n"); // Added print
#endif
    while (fgets(mystring, 12, finfo) != NULL)
    {
        stages_array[i] = atoi(mystring);
#ifdef FINAL_DEBUG
        printf("Stage %d filters: %d\n", i, stages_array[i]); // Added print
#endif
        total_nodes += stages_array[i];
        i++;
    }
    fclose(finfo);
#ifdef FINAL_DEBUG
```

```
    printf("\n-- Finished reading stages array from info.txt --\n");
#endif



    /* TODO: use matrices where appropriate */
    /********************************************
     * Allocate a lot of array structures
     * Note that, to increase parallelism,
     * some arrays need to be splitted or duplicated
     ********************************************/
    int* rectangles_array = (int*)malloc(sizeof(int) * total_nodes * 12); // Local variable
    cascade->rectangles_array = rectangles_array;                     // Assign to cascade struct member

    int** scaled_rectangles_array = (int**)malloc(sizeof(int*) * total_nodes * 12); // Local variable - Note: Is this used in CUDA? If not, can be removed.
    cascade->scaled_rectangles_array = scaled_rectangles_array; // Assign to cascade struct member

    int* weights_array = (int*)malloc(sizeof(int) * total_nodes * 3); // Local variable
    cascade->weights_array = weights_array;                       // Assign to cascade struct member
```

```
    int* alpha1_array = (int*)malloc(sizeof(int) * total_nodes);   // Local
variable

    cascade->alpha1_array = alpha1_array;                          // Assign to
cascade struct member



    int* alpha2_array = (int*)malloc(sizeof(int) * total_nodes);   // Local
variable

    cascade->alpha2_array = alpha2_array;                          // Assign to
cascade struct member



    int* tree_thresh_array = (int*)malloc(sizeof(int) * total_nodes); // Local
variable

    cascade->tree_thresh_array = tree_thresh_array;                   // Assign
to cascade struct member



    float* stages_thresh_array = (float*)malloc(sizeof(float) * stages); //
Local variable - Note: Changed to float* to match myCascade struct

    cascade->stages_thresh_array = stages_thresh_array;                //
Assign to cascade struct member




    FILE* fp = fopen("class.txt", "rb");

    if (fp == NULL) { // Check if file opened successfully


#ifdef FINAL_DEBUG

        printf("Error opening class.txt!\n");

#endif
```

```
        return; // Exit if file not opened
    }
#ifdef FINAL_DEBUG
    printf("Successfully opened class.txt\n"); // Added print
#endif


#ifdef FINAL_DEBUG
    printf("\n-- Reading classifier parameters from class.txt --\n"); // Added
print
#endif


    /****************************************
     * Read the filter parameters in class.txt
     * ... (rest of the parameter reading code is the same) ...
     ****************************************/


    /* loop over n of stages */
    for (i = 0; i < stages; i++)
    {    /* loop over n of trees */
        for (j = 0; j < stages_array[i]; j++)
        {    /* loop over n of rectangular features */
            for (k = 0; k < 3; k++)
            {    /* loop over the n of vertices */
                for (l = 0; l < 4; l++)
```

```
                {
                    if (fgets(mystring, 12, fp) != NULL) {
                        rectangles_array[r_index] = atoi(mystring);
                        //printf("rectangles_array[%d] = %d\n", r_index,
rectangles_array[r_index]); // Added print
                    }
                    else {
#ifdef FINAL_DEBUG
                        printf("Error reading rectangles_array at index %d from
class.txt!\n", r_index);
#endif
                        fclose(fp);
                        return;
                    }
                    r_index++;
                } /* end of l loop */
                if (fgets(mystring, 12, fp) != NULL)
                {
                    weights_array[w_index] = atoi(mystring);
                    //printf("weights_array[%d] = %d\n", w_index,
weights_array[w_index]); // Added print
                    /* Shift value to avoid overflow in the haar evaluation */
                    /*TODO: make more general */
                    /*weights_array[w_index]>>=8; */
                }
```

```
                else {
#ifdef FINAL_DEBUG
                    printf("Error reading weights_array at index %d from
class.txt!\n", w_index);
#endif
                    fclose(fp);
                    return;
                }
                w_index++;
            } /* end of k loop */
            if (fgets(mystring, 12, fp) != NULL) {
                tree_thresh_array[tree_index] = atoi(mystring);
                //printf("tree_thresh_array[%d] = %d\n", tree_index,
tree_thresh_array[tree_index]); // Added print
            }
            else {
                printf("Error reading tree_thresh_array at index %d from
class.txt!\n", tree_index);
                fclose(fp);
                return;
            }
            if (fgets(mystring, 12, fp) != NULL) {
                alpha1_array[tree_index] = atoi(mystring);
                //printf("alpha1_array[%d] = %d\n", tree_index,
alpha1_array[tree_index]); // Added print
```

```
            }
            else {
                printf("Error reading alpha1_array at index %d from
class.txt!\n", tree_index);
                fclose(fp);
                return;
            }
            if (fgets(mystring, 12, fp) != NULL) {
                alpha2_array[tree_index] = atoi(mystring);
                //printf("alpha2_array[%d] = %d\n", tree_index,
alpha2_array[tree_index]); // Added print
            }
            else {
                printf("Error reading alpha2_array at index %d from
class.txt!\n", tree_index);
                fclose(fp);
                return;
            }
            tree_index++;
            if (j == stages_array[i] - 1)
            {
                if (fgets(mystring, 12, fp) != NULL) {
                    stages_thresh_array[i] = atoi(mystring);
                    //printf("stages_thresh_array[%d] = %f\n", i,
stages_thresh_array[i]); // Added print - Changed to %f for float
```

```
                }
                else {
                    printf("Error reading stages_thresh_array at index %d from
class.txt!\n", i);
                    fclose(fp);
                    return;
                }
            }
        } /* end of j loop */
    } /* end of i loop */
    fclose(fp);

#ifdef FINAL_DEBUG
    printf("\n-- Finished reading classifier parameters from class.txt --\n");
// Added print
    printf("\n-- Exiting readTextClassifier --\n"); // Added print
#endif
}



void releaseTextClassifier(myCascade* cascade)
{
  free(cascade->stages_array);
  free(cascade->rectangles_array);
```

```
  free(cascade->scaled_rectangles_array);
  free(cascade->weights_array);
  free(cascade->tree_thresh_array);
  free(cascade->alpha1_array);
  free(cascade->alpha2_array);
  free(cascade->stages_thresh_array);
}
/* End of file. */
```

Appendix E: haar_cuda.h

C/C++

```
/*
 *  TU Eindhoven
 *  Eindhoven, The Netherlands
 *
 *  Name            :   haar.h
 *
 *  Author          :   Francesco Comaschi (f.comaschi@tue.nl)
 *
 *  Date            :   November 12, 2012
 *
 *  Function        :   Haar features evaluation for face detection
 *
 *  History         :
 *      12-11-12    :   Initial version.
 *
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; If not, see <http://www.gnu.org/licenses/>
 *
 * In other words, you are welcome to use, share and improve this program.
 * You are forbidden to forbid anyone else to use, share and improve
 * what you give them.   Happy coding!
 *
 * Modified 03.12.25 by: Ibrahim Binmahfood, Kunjan Vyas, Robert Wilcox
 * to include CUDA code for parallel processing
 */


#ifndef __HAAR_CUDA_H__
#define __HAAR_CUDA_H__


#include <stdio.h>
#include <stdlib.h>
#include "image_cuda.h"
#include "rectangles_cuda.h"
```

```
#ifdef __cplusplus
#include <vector>
#endif

#define MAXLABELS 50

#ifdef __cplusplus
extern "C" {
#endif

    /* C-Compatible Type Definitions */

    typedef int sumtype;
    typedef int sqsumtype;

    typedef struct {
        int x;
        int y;
    } MyPoint;

    typedef struct {
        int width;
        int height;
```

```
    } MySize;


    typedef struct {
        int n_stages;
        int total_nodes;
        float scale;
        MySize orig_window_size;
        float inv_window_area;
        MyIntImage sum;
        MyIntImage sqsum;
        sqsumtype* pq0, * pq1, * pq2, * pq3;
        sumtype* p0, * p1, * p2, * p3;


        // Added members for CUDA implementation:
        int* stages_array;         // Array to store number of weak classifiers
per stage
        float* stages_thresh_array;    // Array to store stage thresholds
        int* rectangles_array;       // Array to store rectangle features
        int* weights_array;          // Array to store weights
        int* alpha1_array;         // Array to store alpha1 values
        int* alpha2_array;         // Array to store alpha2 values
        int* tree_thresh_array;      // Array to store tree thresholds
             int** scaled_rectangles_array;  // Array to store scaled
rectangles
```

```
    } myCascade;


    /* C-Compatible Function Declarations */


    /* Sets images for Haar classifier cascade */

    void setImageForCascadeClassifier(myCascade* cascade, MyIntImage* sum,
MyIntImage* sqsum);


    /* Runs the cascade on the specified window */

    int runCascadeClassifier(myCascade* cascade, MyPoint pt, int start_stage);


    /* Reads the classifier file into memory */

    void readTextClassifier(myCascade* cascade);


    /* Releases classifier resources */

    void releaseTextClassifier(myCascade* cascade);


    /* Computes integral images (and squared integral images) from a source
image */

    void integralImages(MyImage* src, MyIntImage* sum, MyIntImage* sqsum);


#ifdef __cplusplus

} // End of extern "C"
```

```
#endif


#ifdef __cplusplus
/* C++-Only Function Declarations (using std::vector) */

std::vector<MyRect> detectObjects(MyImage* image, MySize minSize, MySize
maxSize,
    myCascade* cascade, float scale_factor, int min_neighbors);
#endif


#endif // __HAAR_CUDA_H__
```

Appendix F: image_cuda.c

```
C/C++

/*
 *  TU Eindhoven
 *  Eindhoven, The Netherlands
 *
 *  Name            :   image.c
 *
 *  Author          :   Francesco Comaschi (f.comaschi@tue.nl)
 *
 *  Date            :   November 12, 2012
 *
 *  Function        :   Functions to manage .pgm images and integral images
 *
 *  History         :
 *      12-11-12    :   Initial version.
 *
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program;  If not, see <http://www.gnu.org/licenses/>
 *
 * In other words, you are welcome to use, share and improve this program.
 * You are forbidden to forbid anyone else to use, share and improve
 * what you give them.   Happy coding!
 *
 * Modified 03.12.25 by: Ibrahim Binmahfood, Kunjan Vyas, Robert Wilcox
 * to include CUDA code for parallel processing
 */


#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "image_cuda.h"


char* strrev(char* str)
{
      char *p1, *p2;
```

```
	if (!str || !*str)
		return str;
	for (p1 = str, p2 = str + strlen(str) - 1; p2 > p1; ++p1, --p2)
	{
		*p1 ^= *p2;
		*p2 ^= *p1;
		*p1 ^= *p2;
	}
	return str;
}


//int chartoi(const char *string)
//{
//	int i;
//	i=0;
//	while(*string)
//	{
//		// i<<3 is equivalent of multiplying by 2*2*2 or 8
//		// so i<<3 + i<<1 means multiply by 10
//		i=(i<<3) + (i<<1) + (*string - '0');
//		string++;
//
//		// Dont increment i!
```

```
//
//     }
//     return(i);
//}


int myatoi (char* string)
{
       int sign = 1;
       // how many characters in the string
       int length = strlen(string);
       int i = 0;
       int number = 0;

       // handle sign
       if (string[0] == '-')
       {
              sign = -1;
              i++;
       }

//     for (i; i < length; i++)
       while(i < length)
       {
```

```
		// handle the decimal place if there is one
		if (string[i] == '.')
			break;
		number = number * 10 + (string[i]- 48);
		i++;
	}

	number *= sign;

	return number;
}

void itochar(int x, char* szBuffer, int radix)
{
	int i = 0, n, xx;
	n = x;
	while (n > 0)
	{
		xx = n%radix;
		n = n/radix;
		szBuffer[i++] = '0' + xx;
	}
	szBuffer[i] = '\0';
```

```
	strrev(szBuffer);
}



int readPgm(char *fileName, MyImage *image)
{
	FILE *in_file;
	char ch;
	int type;
	char line[100];
	char mystring [20];
	char *pch;
	int i;
	long int position;


    //changed from 'r' to 'rb' to fix sub and cntrl-z errors since windows
reads 'r' as text and 'rb' as binary
    //link to suggestion:
http://stackoverflow.com/questions/15874619/reading-in-a-text-file-with-a-sub-1
a-control-z-character-in-r-on-windows
    //another link:
http://article.gmane.org/gmane.comp.lang.r.devel/33213/match=duncan+murdoch+con
trol+z
	in_file = fopen(fileName, "rb");
	if (in_file == NULL)
	{
```

```
        printf("ERROR: Unable to open file %s\n\n", fileName);
        return -1;
}
printf("\nReading image file: %s\n", fileName);
// Determine image type (only pgm format is allowed)*/
ch = fgetc(in_file);
if(ch != 'P')
{
        printf("ERROR: Not valid pgm file type\n");
        return -1;
}

ch = fgetc(in_file);



/*convert the one digit integer currently represented as a character to

  an integer(48 == '0')*/

type = ch - 48;

if(type != 5)
{
```

```
                printf("ERROR: only pgm raw format is allowed\n");
                return -1;
        }
        // Skip comments
//      char line[100];
        while ((ch = fgetc(in_file)) != EOF && isspace(ch));
        position = ftell(in_file);



        // skip comments
        if (ch == '#')
                {
                        fgets(line, sizeof(line), in_file);
                        while ((ch = fgetc(in_file)) != EOF &&
isspace(ch));//increment steam position until
                        position = ftell(in_file);//ftell: get current position is
stream
                }


        fseek(in_file, position-1, SEEK_SET);//originally position-1, set to -3,
then changed back to -1 after 'rb' change in reading the file


        fgets (mystring , 20, in_file);
        pch = (char *)strtok(mystring," ");
        image->width = atoi(pch);
```

```
        pch = (char *)strtok(NULL," ");
        image->height = atoi(pch);
        fgets (mystring , 5, in_file);
        image->maxgrey = atoi(mystring);
        image->data = (unsigned char*)malloc(sizeof(unsigned
char)*(image->height*image->width));//new unsigned char[row*col];
        image->flag = 1;
        for(i=0;i<(image->height*image->width);i++)
    //for(i=0;i<(738);i++)
        {
              ch = fgetc(in_file);
              image->data[i] = (unsigned char)ch;
        }
        fclose(in_file);
        return 0;
}


int writePgm(char *fileName, MyImage *image)
{
        char parameters_str[5];
        int i;
        const char *format = "P5";
        if (image->flag == 0)
        {
```

```
		return -1;
	}

	//changed from 'w' to 'wb' to fix sub and cntrl-z errors since windows writes 'w' as text and 'wb' as binary
    //this is similar to the issue resolved in the readPgm() function
	FILE *fp = fopen(fileName, "wb");
	if (!fp)
	{
		printf("Unable to open file %s\n", fileName);
		return -1;
	}
	fputs(format, fp);
	fputc('\n', fp);

	itochar(image->width, parameters_str, 10);
	fputs(parameters_str, fp);
	parameters_str[0] = 0;
	fputc(' ', fp);

	itochar(image->height, parameters_str, 10);
	fputs(parameters_str, fp);
	parameters_str[0] = 0;
	fputc('\n', fp);
```

```
	itochar(image->maxgrey, parameters_str, 10);
	fputs(parameters_str, fp);
	fputc('\n', fp);

	for (i = 0; i < (image->width * image->height); i++)
	{
		fputc(image->data[i], fp);
	}
	fclose(fp);
	return 0;
}

int cpyPgm(MyImage* src, MyImage* dst)
{
	int i = 0;
	if (src->flag == 0)
	{
		printf("No data available in the specified source image\n");
		return -1;
	}
	dst->width = src->width;
	dst->height = src->height;
```

```
	dst->maxgrey = src->maxgrey;

	dst->data = (unsigned char*)malloc(sizeof(unsigned
char)*(dst->height*dst->width));

	dst->flag = 1;

	for (i = 0; i < (dst->width * dst->height); i++)

	{

		dst->data[i] = src->data[i];

	}

	return 0;

}




void createImage(int width, int height, MyImage *image)

{

	image->width = width;

	image->height = height;

	image->flag = 1;

	image->data = (unsigned char *)malloc(sizeof(unsigned
char)*(height*width));

}



void createSumImage(int width, int height, MyIntImage *image)

{

	image->width = width;
```

```
	image->height = height;
	image->flag = 1;
	image->data = (int *)malloc(sizeof(int)*(height*width));
}


int freeImage(MyImage* image)
{
	if (image->flag == 0)
	{
		printf("no image to delete\n");
		return -1;
	}
	else
	{
//		printf("image deleted\n");
		free(image->data);
		return 0;
	}
}


int freeSumImage(MyIntImage* image)
{
	if (image->flag == 0)
```

```
	{
		printf("no image to delete\n");
		return -1;
	}
	else
	{
//		printf("image deleted\n");
		free(image->data);
		return 0;
	}
}

void setImage(int width, int height, MyImage *image)
{
	image->width = width;
	image->height = height;
}

void setSumImage(int width, int height, MyIntImage *image)
{
	image->width = width;
	image->height = height;
}
```

Appendix G: image_cuda.h

C/C++

```
/*
 *  TU Eindhoven
 *  Eindhoven, The Netherlands
 *
 *  Name            :   image.h
 *
 *  Author          :   Francesco Comaschi (f.comaschi@tue.nl)
 *
 *  Date            :   November 12, 2012
 *
 *  Function        :   Functions to manage .pgm images and integral images
 *
 *  History         :
 *      12-11-12    :   Initial version.
 *
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program;  If not, see <http://www.gnu.org/licenses/>
 *
 * In other words, you are welcome to use, share and improve this program.
 * You are forbidden to forbid anyone else to use, share and improve
 * what you give them.   Happy coding!
 *
 * Modified 03.12.25 by: Ibrahim Binmahfood, Kunjan Vyas, Robert Wilcox
 * to include CUDA code for parallel processing
 */


#ifndef __IMAGE_CUDA_H__
#define __IMAGE_CUDA_H__


#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#ifdef __cplusplus
```

```
extern "C" {
#endif

typedef struct
{
	int width;
	int height;
	int maxgrey;
	unsigned char* data;
	int flag;
	int allocatedSize;  // track allocated buffer size
}
MyImage;

typedef struct
{
	int width;
	int height;
	int* data;
	int flag;
}
MyIntImage;
```

```
int readPgm(char *fileName, MyImage* image);

int writePgm(char *fileName, MyImage* image);

int cpyPgm(MyImage *src, MyImage *dst);

void createImage(int width, int height, MyImage *image);

void createSumImage(int width, int height, MyIntImage *image);

int freeImage(MyImage* image);

int freeSumImage(MyIntImage* image);

void setImage(int width, int height, MyImage *image);

void setSumImage(int width, int height, MyIntImage *image);


#ifdef __cplusplus

}

#endif


#endif
```

Appendix H: rectangles_cuda.cpp

```C/C++
#include "haar_cuda.h"


int myMax(int a, int b)
{
  if (a >= b)
    return a;
  else
    return b;
}


int myMin(int a, int b)
{
  if (a <= b)
    return a;
  else
    return b;
}


inline  int  myRound( float value )
{
  return (int)(value + (value >= 0 ? 0.5 : -0.5));
```

```
}


int myAbs(int n)

{

  if (n >= 0)

    return n;

  else

    return -n;

}


int predicate(float eps, MyRect& r1, MyRect& r2)

{

  float delta = eps*(myMin(r1.width, r2.width) + myMin(r1.height,
r2.height))*0.5;

  return myAbs(r1.x - r2.x) <= delta &&

    myAbs(r1.y - r2.y) <= delta &&

    myAbs(r1.x + r1.width - r2.x - r2.width) <= delta &&

    myAbs(r1.y + r1.height - r2.y - r2.height) <= delta;

}


void groupRectangles(std::vector<MyRect>& rectList, int groupThreshold, float
eps)

{

  if( groupThreshold <= 0 || rectList.empty() )
```

```
  return;



std::vector<int> labels;


int nclasses = partition(rectList, labels, eps);


std::vector<MyRect> rrects(nclasses);
std::vector<int> rweights(nclasses);


int i, j, nlabels = (int)labels.size();



for( i = 0; i < nlabels; i++ )
  {
    int cls = labels[i];
    rrects[cls].x += rectList[i].x;
    rrects[cls].y += rectList[i].y;
    rrects[cls].width += rectList[i].width;
    rrects[cls].height += rectList[i].height;
    rweights[cls]++;
  }
for( i = 0; i < nclasses; i++ )
```

```
  {
    MyRect r = rrects[i];
    float s = 1.f/rweights[i];
    rrects[i].x = myRound(r.x*s);
    rrects[i].y = myRound(r.y*s);
    rrects[i].width = myRound(r.width*s);
    rrects[i].height = myRound(r.height*s);

  }

rectList.clear();

for( i = 0; i < nclasses; i++ )
  {
    MyRect r1 = rrects[i];
    int n1 = rweights[i];
    if( n1 <= groupThreshold )
     continue;
    /* filter out small face rectangles inside large rectangles */
    for( j = 0; j < nclasses; j++ )
      {
       int n2 = rweights[j];
       /*********************************
```

```
     * if it is the same rectangle,
     * or the number of rectangles in class j is < group threshold,
     * do nothing
     ********************************/
    if( j == i || n2 <= groupThreshold )
      continue;
    MyRect r2 = rrects[j];

    int dx = myRound( r2.width * eps );
    int dy = myRound( r2.height * eps );

    if( i != j &&
        r1.x >= r2.x - dx &&
        r1.y >= r2.y - dy &&
        r1.x + r1.width <= r2.x + r2.width + dx &&
        r1.y + r1.height <= r2.y + r2.height + dy &&
        (n2 > myMax(3, n1) || n1 < 3) )
      break;
   }

 if( j == nclasses )
   {
    rectList.push_back(r1); // insert back r1
```

```
        }
    }

}



int partition(std::vector<MyRect>& _vec, std::vector<int>& labels, float eps)
{
  int i, j, N = (int)_vec.size();

  MyRect* vec = &_vec[0];

  const int PARENT=0;
  const int RANK=1;

  std::vector<int> _nodes(N*2);

  int (*nodes)[2] = (int(*)[2])&_nodes[0];

  /* The first O(N) pass: create N single-vertex trees */
  for(i = 0; i < N; i++)
    {
      nodes[i][PARENT]=-1;
```

```
    nodes[i][RANK] = 0;
  }

/* The main O(N^2) pass: merge connected components */
for( i = 0; i < N; i++ )
  {
    int root = i;

    /* find root */
    while( nodes[root][PARENT] >= 0 )
     root = nodes[root][PARENT];

    for( j = 0; j < N; j++ )
     {
       if( i == j || !predicate(eps, vec[i], vec[j]))
         continue;
       int root2 = j;

       while( nodes[root2][PARENT] >= 0 )
         root2 = nodes[root2][PARENT];

       if( root2 != root )
         {
```

```
/* unite both trees */
int rank = nodes[root][RANK], rank2 = nodes[root2][RANK];
if( rank > rank2 )
 nodes[root2][PARENT] = root;
else
 {
   nodes[root][PARENT] = root2;
   nodes[root2][RANK] += rank == rank2;
   root = root2;
 }

int k = j, parent;

/* compress the path from node2 to root */
while( (parent = nodes[k][PARENT]) >= 0 )
 {
   nodes[k][PARENT] = root;
   k = parent;
 }

/* compress the path from node to root */
k = i;
while( (parent = nodes[k][PARENT]) >= 0 )
```

```
            {
              nodes[k][PARENT] = root;
              k = parent;
            }
          }
      }
  }

/* Final O(N) pass: enumerate classes */
labels.resize(N);
int nclasses = 0;

for( i = 0; i < N; i++ )
  {
    int root = i;
    while( nodes[root][PARENT] >= 0 )
     root = nodes[root][PARENT];
    /* re-use the rank as the class label */
    if( nodes[root][RANK] >= 0 )
     nodes[root][RANK] = ~nclasses++;
    labels[i] = ~nodes[root][RANK];
  }
```

```
  return nclasses;
}



/* draw white bounding boxes around detected faces */
void drawRectangle(MyImage* image, MyRect r)
{
       int i;
       int col = image->width;

       for (i = 0; i < r.width; i++)
       {
      image->data[col*r.y + r.x + i] = 255;
       }
       for (i = 0; i < r.height; i++)
       {
              image->data[col*(r.y+i) + r.x + r.width] = 255;
       }
       for (i = 0; i < r.width; i++)
       {
              image->data[col*(r.y + r.height) + r.x + r.width - i] = 255;
       }
       for (i = 0; i < r.height; i++)
```

```
        {
                image->data[col*(r.y + r.height - i) + r.x] = 255;
        }


}
```

Appendix I: rectangles_cuda.h

```C/C++
#ifndef __RECTANGLES_CUDA_H

#define __RECTANGLES_CUDA_H


#include <stdio.h>

#include <stdlib.h>

#include "haar_cuda.h"

#include "image_cuda.h"


#ifdef __cplusplus

#include <vector>

#endif


#ifdef __cplusplus

extern "C" {

#endif


typedef struct {

    int x;

    int y;

    int width;

    int height;
```

```
} MyRect;


int myMax(int a, int b);


int myMin(int a, int b);


inline  int  myRound( float value );


int myAbs(int n);


int predicate(float eps, MyRect& r1, MyRect& r2);


/* Draws white bounding boxes around detected faces */
void drawRectangle(MyImage* image, MyRect r);
#ifdef __cplusplus
} // End of extern "C"
#endif


#ifdef __cplusplus
/* C++-Only Function Declarations (using std::vector) */


int partition(std::vector<MyRect>& _vec, std::vector<int>& labels, float eps);
```

```
void groupRectangles(std::vector<MyRect>& _vec, int groupThreshold, float eps);


#endif


#endif // __RECTANGLES_CUDA_H
```