Accelerating Boosting-based Face Detection on GPUs

David Oro*, Carles Fernández*, Carlos Segura*, Xavier Martorell† and Javier Hernando†
*Herta Security, Barcelona, Spain
†Universitat Politècnica de Catalunya, Barcelona, Spain
Email: david.oro@hertasecurity.com xavim@ac.upc.edu javier@tsc.upc.edu

***Abstract*—The goal of face detection is to determine the presence of faces in arbitrary images, along with their locations and dimensions. As it happens with any graphics workloads, these algorithms benefit from data-level parallelism. Existing parallelization efforts strictly focus on mapping different divide and conquer strategies into multicore CPUs and GPUs. However, even the most advanced single-chip many-core processors to date are still struggling to effectively handle real-time face detection under high-definition video workloads. To address this challenge, face detection algorithms typically avoid computations by dynamically evaluating a boosted cascade of classifiers. Unfortunately, this technique yields a low ALU occupancy in architectures such as GPUs, which heavily rely on large SIMD widths for maximizing data-level parallelism. In this paper we present several techniques to increase the performance of the cascade evaluation kernel, which is the most resource-intensive part of the face detection pipeline. Particularly, the usage of concurrent kernel execution in combination with cascades generated with the GentleBoost algorithm solves the problem of GPU underutilization, and achieves a 5X speedup in 1080p videos on average over the fastest known implementations, while slightly improving the accuracy. Finally, we also studied the parallelization of the cascade training process and its scalability under SMP platforms. The proposed parallelization strategy exploits both task and data-level parallelism and achieves a 3.5X speedup over single-threaded implementations.**

***Keywords*-Face detection, GPU, parallel programming, video processing**

I. Introduction

Currently available face detection algorithms usually rely on feature descriptors that perform a large amount of operations to identify which image regions contain human faces. As image and video resolutions for complex computer vision applications increase, hardware designers should also meet the demand in continued scaling of performance.

Addressing this challenge is only possible through the usage of massively parallel microarchitectures that are highly efficient in terms of performance per watt. Even though the latest multicore CPUs feature instructions with an increased SIMD width such as the AVX extensions, GPUs yield more performance with less energy consumed per floating-point operation [1]. For this reason, data centers are increasingly relying on GPUs and heterogeneous architectures to run computer-intensive services such as object or face search in photos and videos.

Recent advances in face detection propose using features such as LBP[2], SIFT[3] or SURF[4] that were originally designed for object recognition [5]. These efforts were made not only to improve the speed of detection but also to yield a better ROC curve performance. Although the feature extraction step is inherently parallel, modern face detection methods still rely on an attentional cascade derived from a learning algorithm for discarding computations. Therefore, the main parallelization challenge when evaluating these cascades in GPUs is how to solve the low utilization of GPU resources derived from early rejections and thread divergence.

In this paper we present a highly parallel face detection implementation that relies on GentleBoost [6] and concurrent kernel execution for increasing the occupancy in GPUs. Moreover, we show that a multi-level parallelization and the proper customization of the boosted cascade of classifers not only reduces the execution time, but also slightly improves the detection quality. The obtained results show that our GPU face detector is capable of detecting multiple faces in real time at 70 fps under 1080p resolutions while performing H.264 video decoding. Additionally, we also study the scalability of the parallelization of the cascade training process under SMP architectures.

This paper is structured as follows: Section 2 provides an overview of the related work. Section 3 describes the parallelization strategy and pipeline used for performing face detection. Section 4 discusses the parallelization of the cascade training process. Section 5 details the experimental setup. Section 6 conducts an evaluation of the obtained cascade and shows the experimental results of our GPU face detector implementation. Finally, in Section 7 conclusions are drawn and future work is proposed.

II. Related Work

Modern face detection frameworks are usually built from the combination of an image feature extraction process and classifiers resulting from a given learning algorithm. Viola and Jones [7] introduced the first face detector that worked at a reasonable speed with low error rates. With the selection of lightweight Haar features and the usage of integral images they showed that it was possible to dramatically reduce the required memory accesses and computations. Another major contribution was the introduction of a cascade organized in

stages constituted by a gradually larger number of classifiers to quickly reject image regions. In this approach, the classifiers were created with AdaBoost [8] and structured in stages using heuristics based on false positive and hit rates.

Most of the existing works in GPU face detection are parallelized versions of the original Viola and Jones framework. Sharma *et al.* [9] proposed a parallel detector based on an image pyramid with the purpose of maximizing the thread count and thus GPU occupancy. Instead of scaling sliding windows for detecting faces of arbitrary sizes, the window is kept constant and the input image downscaled.

Hefenbrock *et al.* [10] examined the tradeoffs between feature, window and scale parallelization and concluded that all three alternatives suffer from unbalanced distribution of work. They proposed a multi-GPU solution where each detection window is evaluated in a different thread, and each window scale computed in parallel in a different GPU.

Obukhov [11] presented an implementation that combined pixel-parallel and stage-parallel processing kernels. Since the amount of classifiers per stage grows with the depth of the cascade, occupancy is maximized when a pixel-parallel kernel is used for the strong classifiers and a stage-parallel kernel for the weaker ones.

As opposed to the previous works, Herout *et al.* [12] introduced a new GPU object detector based on WaldBoost [13] and LRP features [14]. In this approach, the evaluation of the cascade generated by WaldBoost is parallelized with a fixed-sized detection window and the low occupancy problem addressed with thread rearrangement. All image locations that have not been early rejected are reassigned into threads that share the same blocks. Then the cascade evaluation kernel is relaunched to process these blocks, and thread rearrangement repeated until all image locations are computed.

III. Parallel Face Detection

The growing trend in the high performance computing semiconductor industry is to devote a larger amount of the die area in several parallel processing units, structured among blocks of multithreaded SIMD lanes and private local caches. GPUs push to the limit this architecture by executing hundreds of thousands of threads, but may yield poor performance if the workloads are unbalanced. Even though modern face detection algorithms benefit from the data-level parallelism inherently derived from pixels, the feature evaluation process suffers from an irregular control flow when implemented in a kernel function. In this section, we introduce a pipeline that exploits different degrees of parallelism for increasing the performance of the face detection process in modern GPUs using the CUDA programming model.

A. Pipeline Overview

Typically, the face detection pipeline (see Figure 1) starts from an image or video frame. Since it is now common to have an on-die hardware video decoder, time consuming tasks such as the decoding of 1080p high definition H.264 video streams can be offloaded at the slice level to this fixed function logic. When the video decoding stage is performed in a GPU, the latency of memory transfers between the CPU and GPU address space is significantly reduced due to the fact that these transfers deal with compressed video frames. Through the usage of GPU hardware video decoding APIs (e.g. NVCUVID [15]), the decompressed video frames are then directly mapped into the texture memory.

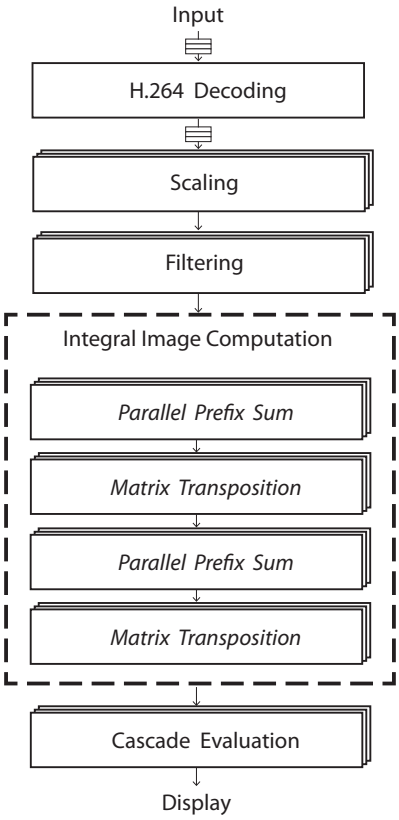

Figure 1. Proposed pipeline for parallel face detection.

At this point, the scaling stage generates n resized images by subsampling the decompressed frame stored in the texture memory. Since this memory is indexed using floating point coordinates, it is possible to configure it for performing texture fetches with linear interpolation using `tex2D` instructions [16]. The main reason for building this image pyramid is to avoid rescaling the features for detecting faces of an arbitrary size when evaluating the classifier cascade. As it was discussed in previous works [9], [11], [12], GPU occupancy is extremely low if the features are scaled and the image dimensions kept constant. In the latter case, if a single thread evaluates the classifier cascade for a given sliding window, the potential number of threads dramatically reduces as the size of the window increases (see Figure 2).

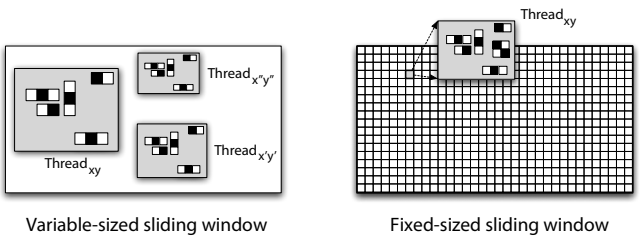

Figure 2. Different strategies for evaluating a boosted cascade of classifiers.

The filtering stage of the pipeline is necessary to avoid aliasing effects produced during the scaling stage, and thus preserve the original properties of the underlying image signal. After this stage has been completed, integral images are computed by exploiting multiple levels of parallelism with a combination of parallel prefix sum [17] [18] and matrix transposition operations [19]. By using this approach, coarse-grain parallelism is exploited through concurrently executing the kernels that implement the abovementioned operations for each one of the considered scales. Fine-grain parallelism is subsequently exploited at thread-level within each kernel.

Finally, the face detection pipeline concludes with the evaluation of the boosted cascade of classifiers from the integral images. This stage is the most resource-intensive and also exploits two degrees of parallelism. From a low-level perspective, a divide and conquer strategy is used for evaluating the cascade within a kernel, and the analysis of the input integral image performed by splitting it into equally-sized blocks corresponding to a different fixed-sized sliding window. With this parallelization pattern, faces of multiple sizes are thus detected by concurrently executing the same kernel for each integral image corresponding to a different scale.

B. Integral Image Computation

The parallelization of integral image computation was initially discussed by Hensley *et al.* [20] and implemented using row-wise and column-wise prefix sum operations in shaders. This approach was later refined by Messom *et al.* [21] and Bilgic *et al.* [22] by introducing matrix transpositions after performing row-wise prefix sum operations. For small resolutions a naive sequential $O(n \cdot m)$ CPU implementation beats the GPU due to the fact that the whole image fits in the L2 cache. However, the GPU implementation is 2.5 times faster on average for high resolution images [23].

C. Cascade Evaluation Kernel

A typical data-parallel face detection kernel must perform an exhaustive evaluation of a boosted cascade of classifiers organized in stages containing an increasingly larger number of filters. In order to detect faces of any size at multiple locations, the kernel not only has to test all integral image pixels, but also consider all possible sliding window dimensions and locations. This algorithm could also be significantly improved by performing rotations of the integral image, thus exponentially increasing the required amount of computations. Due to the particular properties of boosting algorithms, the traversal of the cascade data structure is constrained by the dependencies between stages, and must be sequentially evaluated for each candidate window. For these reasons, we propose a parallelization pattern, where each integral image I_{int} corresponding to a different downscaled video frame is divided into small and equally-sized $n \times m$ chunks.

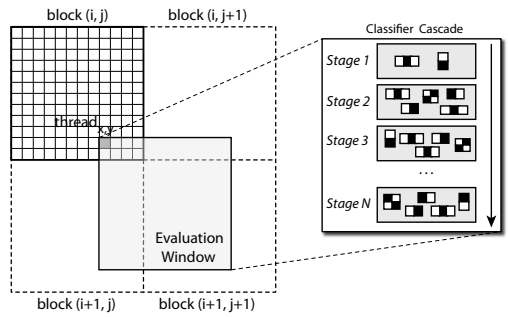

Figure 3. Evaluation of the classifier cascade for a given block thread.

Each chunk is directly mapped into a different thread block $B(i, j)$, which is then scheduled to a streaming multiprocessor (SM). Let W be the memory chunk to be transferred to a SM, the pixels of a given integral image I_{int} to be transferred by a given (x, y) thread of an $n \times m$ thread block $B(i, j)$ are determined by the following equations:

$$W(x, y) = I_{int}(\alpha, \beta) \tag{1}$$

$$W(x + n, y) = I_{int}(\alpha + n, \beta) \tag{2}$$

$$W(x, y + m) = I_{int}(\alpha, \beta + m) \tag{3}$$

$$W(x + n, y + m) = I_{int}(\alpha + n, \beta + m) \tag{4}$$

Where $\alpha = i \cdot n + x$ and $\beta = j \cdot m + y$. According to these definitions, each thread of the block will bring 4 pixels of the integral image to the shared memory, where 3 of them will be of memory regions meant to be explored by contiguous blocks (see Figure 3). Additionally, these access patterns are sequentially performed to achieve memory coalescing.

After these memory transfers have been completed (and ensured by a `_syncthreads` barrier instruction), the next step of the algorithm is to evaluate the cascade classifiers. This evaluation is both an arithmetic and a memory intensive operation. For a boosted cascade of Haar classifiers, it involves 18 memory accesses for the 2-rectangle feature and 27 memory accesses for the 3-rectangle feature.

Table I
POSSIBLE HAAR-LIKE FEATURE COMBINATIONS (24X24 PIXELS)

Haar-like Feature	Shape	Combinations
Edge		55660
Line		31878
Center-surround		3969
Diagonal		12100

For instance, a given rectangle of a Haar feature is defined by its (i, j) location within the $n \times m$ fixed-sized sliding window, its $w \times h$ dimensions, and the 4 values of the integral image used for computing the area, thus requiring 9 memory accesses. Additionally, it is also required to perform between 4 and 5 arithmetic operations in order to estimate the filter response for each feature. Therefore, the main objective is to reduce latency when fetching from memory all required feature attributes (i.e. dimensions, coordinates and weights) with the purpose of keeping the ALUs busy and thus maximize throughput.

Since these particular information is continuously reused when evaluating the cascade features for each integral image element, the right approach would involve moving all features to the constant memory before launching the evaluation kernels. Moreover, it is also expected that threads processing adjacent pixels start evaluating the cascade features at the same initial stage and diverge in later iterations. In this situation, the usage of the constant memory is appropriate due to the fact that is specifically designed for broadcasting values for all warp threads that simultaneously access the same memory address. In order to maximize the information stored in the constant memory, the Haar feature dataset can be compressed. Since all bits of the thresholds, coordinates, dimensions and weight values are not significant, we propose reencoding and combining them into two 16-bit words using simple bitwise operations and masks.

D. Display

The output of the concurrent execution of the cascade evaluation kernel is a collection of arrays with the same dimensions as the integral images, and stored in the global memory. Each element (x, y) of these arrays is an integer that represents the deepest stage of the cascade reached during the evaluating process. Therefore, the image region enclosed in a sliding window starting at (x, y) would be considered as a face if the integer value stored there equals the maximum depth of the cascade. The precise dimensions of the sliding window are determined by multiplying the downscaling factor by the normalized dimensions of the faces that have been used during the cascade training process. The abovementioned operations are implemented in the display kernel. This kernel is concurrently executed for each scale and encloses faces in a rectangle by updating the RGB values of the original input image or video frame. Finally, the RGB image is mapped into a standard texture for displaying it using the CUDA-OpenGL interoperability API.

IV. CASCADE TRAINING PROCESS

The amount of features and the way in which they are arranged in a cascade of classifiers have severe implications in both the computational performance and the accuracy of any face detection process. Therefore, the main challenge is to reduce the quantity of features while preserving the detection accuracy. This reduced feature set would imply less memory accesses and a lower computational footprint, thus effectively speeding up the face detection process.

For these reasons, we decided to build from scratch an optimized cascade using the GentleBoost [6] learning algorithm. Furthermore, due to great number of Haar-like feature combinations that must be tested during the offline boosting-based training, we parallelized GentleBoost using a combination of task and data-level parallelism.

Task parallelism is exploited by creating as many threads as Haar filters are required to be evaluated for each combination hypothesis. Data-level parallelism is exploited by mapping Haar filters to vectors and then evaluating them using the SSE4 extensions for each image in the training set database. With this simple parallelization pattern, we implemented GentleBoost using a single large loop, which iteratively builds a cascade by adding at each iteration a new classifier until both the target hit and false acceptance rate are met. An additional bootstrapping routine is added at the end of the loop to avoid redundancy in the set of background images, while improving the discriminative power of the boosting algorithm.

In order to select the proper classifier, at each iteration all the combinations presented in Table I must be tested for each image of the training set. This latter nested loop is split into 4 different loops (one corresponding to each Haar filter type) and then parallelized with OpenMP using the `#pragma omp parallel for` construct.

The training set used for building the cascade consisted of 11742 frontal faces of 24x24 pixels and 3500 backgrounds which are also required for providing negative examples. Each 24x24 image is then packed in a big matrix using the Eigen v3 C++ template library [24]. As Figure 4 shows, each iteration of the parallelization loop gathers the coordinates and dimensions of the two rectangles constituting the Haar edge feature. This is implemented by accessing the `haar2` matrix, which contains as many rows as feature combinations. By contrast, each column of this matrix contains the x, y coordinates and $w \times h$ dimensions of the two rectangles.

Therefore, each thread corresponding to a given i iteration of the loop will evaluate a different i Haar feature combination, for all faces and backgrounds of the training set. In order to speedup the evaluation process, the `dataset` matrix

```
#pragma omp parallel for
for(int i=0; i<haar2.rows(); i++)
{
  VectorXi eval(dataset.cols());
  eval = (-1) * (dataset.row(haar2(i,0)) +
                dataset.row(haar2(i,1)) -
                dataset.row(haar2(i,2)) -
                dataset.row(haar2(i,3))) +
            2 * (dataset.row(haar2(i,4)) +
                dataset.row(haar2(i,5)) -
                dataset.row(haar2(i,6)) -
                dataset.row(haar2(i,7)));
  regression(eval, i);
}
```

Figure 4. Parallel loop for testing edge feature combinations.

stores the precomputed values of the integral images of the training set rather than the original pixel values.

The data-level parallelization pattern exploits the fact that, since all integral images have the same dimension (24x24 pixels), it is then possible to encode each of them in a single column of the `dataset` matrix using 576 rows. Hence, this matrix will have as many columns as integral images the training set has. With this simple data transformation, the filter response of each Haar feature to be tested is easily vectorized with SSE4 through the overloaded arithmetic operators provided by the Eigen library.

At the end of the loop, the thread private `eval` vector is used for storing the result of the evaluation of the i Haar feature for the whole training set. Then, the `regression` function is called for estimating the parameters of the weak classifiers from the filter responses stored in `eval`.

Finally, when the four parallelized loops that test the different Haar feature types conclude, a ranking function selects the weak classifier that best discriminates between faces and backgrounds. This weak classifier is then added to the cascade, and the main external loop continues until the target hit and false acceptance rate are achieved.

V. Experimental Setup

In order to evaluate the performance and the accuracy of our proposed GPU face detection implementation, we built a benchmark dataset by retrieving a collection of 10 videos from the iTunes Movie Trailers website [25]. All the selected movie trailers featured a 1920x1080p resolution with an average bitrate of 9 Mbps and were originally compressed using the H.264/AVC codec in a QuickTime container format. The testing platform was equipped with an Intel Core i7-2600K 3.4 GHz, quad-core CPU and an NVIDIA GTX470 graphics card. For the implementation of the GPU kernels, we targeted the CUDA Toolkit 4.1 for Linux and the `sm_20` architecture. Since the on-die GPU video decoder works with bitstreams, we relied on the `libavformat` library [26] for demuxing the input videos. Then the demuxed H.264 video frames were enqueued and

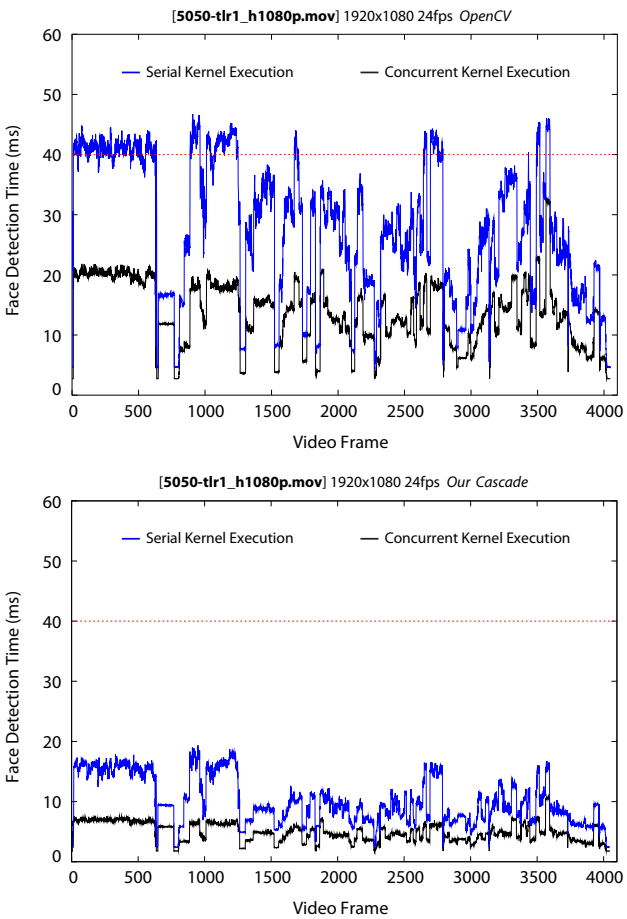

Figure 5. Face detection elapsed time per frame for the *50/50* movie trailer. The obtained results show the difference between serial and concurrent kernel execution for two different cascades: the OpenCV frontal feature set and ours.

decoded using the CUVID callback functions, and finally mapped to a GPU pointer for conducting further processing with the CUDA kernels. Since the hardware decodes frames in NV12 format, it is enough to consider only the initial array of luminance components as the input of the scaling process and subsequent pipeline stages.

For measuring the performance of the GPU kernels, we relied on the CUDA compute command line profiler [27]. Unfortunately, the profiler does not allow gathering values from the performance counters while concurrently executing kernels. To avoid kernel serialization, we enabled and disabled the `conckerneltrace` profiling directive between multiple executions. The objective was to capture the timestamps of the kernels issued in different streams, while performing face detections in a video, in order to determine whether they were effectively overlapping computations or not. Similarly, concurrent execution was deliberately disabled for measuring statistics such as branch and control flow divergence in each streaming multiprocessor (SM).

Finally, we selected the OpenCV frontal face feature set designed by Lienhart *et al.* [28] as a baseline for estimating the accuracy of our own cascade, and also for comparing our parallel detector with the cascade used in previous works [11].

VI. Results

Since we have relied on a combination of parallelization and cascade tuning for speeding up face detection on GPUs, we have not only analyzed the computational performance, but also the accuracy in terms of true and false detections. Even though the cascade training process is carried out offline, it is also a time-consuming process (i.e. usually requiring several days of computation in a quad-core workstation), and should be repeated several times with fine-tuned parameters in order to achieve good results. For this reason, we have also measured the speedup of our parallelized implementation of GentleBoost.

A. Performance

Due to the followed parallelization pattern, we emphasized metrics such as the execution time and branch divergence rather than the memory bandwidth achieved by the cascade evaluation kernels. Since initially integral images are stored in the GPU DRAM memory, and then moved in chunks to the shared memory before performing any computation, the obtained DRAM read throughput for this specific kernel was low. Particularly, it ranged between 532 MB/s and 9.57 MB/s depending on the kernel corresponding to each downscaled input image.

Even though the integral image computation kernels (i.e. scan and parallel matrix transposition) are computationally intensive, they only account on average for a 20% of the total face detection computation time. Given that they were extensively analyzed in previous works [22] [23], we strictly focused on profiling the cascade evaluation kernel and the latency of the whole face detection process.

As Table II shows, switching from serial to concurrent kernel execution doubles the performance when detecting faces in video frames of the selected 1080p movie trailers. Therefore, the low GPU occupancy achieved by serially executing the kernels corresponding to smaller scales is thus effectively avoided when these scales are processed in parallel. The impact on performance of overlapping the cascade evaluation kernel computations for small scales is visible in the execution trace depicted in Figure 6. This trace was produced by gathering the initial and final execution timestamps of each kernel during profiling. Since each kernel is mapped into different CUDA streams, the GPU scheduler is capable of issuing and executing thread instructions from different kernels.

The results also show that replacing the OpenCV feature dataset with our own cascade offers an additional 2.5X performance improvement. Therefore, the combined approach

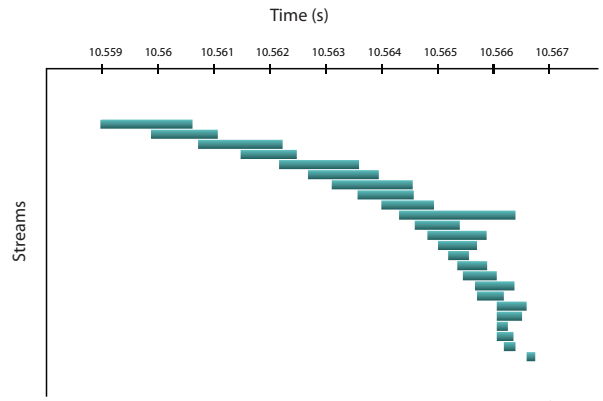

Figure 6. Execution trace of the cascade evaluation kernels for a given video frame of the *50/50* movie trailer. The kernels processing the smaller image scales are executed completely overlapped for maximizing performance.

(i.e. concurrent execution + our cascade) offers a 5X speedup over the serial kernel execution of the OpenCV cascade. Although both cascades feature 25 stages, the OpenCV cascade has 2913 weak classifiers, whereas ours has only 1446. As expected, this reduction in the Haar feature count translates in fewer computations and memory accesses thus reducing the face detection latency.

It should be noted that Table II represents only the face detection elapsed time, and does not take into account the H.264 video decoding latency. As it was discussed in Section III, this task is completely offloaded to the on-die GPU video decoder. For the selected movie trailers, the average decoding latency of a given frame ranged between 8 and 10 ms thus yielding an average throughput of 70 fps when performing both tasks (i.e. video decoding and face detection) in the GPU.

As Figure 5 depicts, the face detection latency per frame for a given movie trailer may experience a huge variability. This latency basically depends on the number of faces that appear on each video frame. Since both cascades were trained using backgrounds and other objects as examples of non-faces, the image regions that do not have any faces are quickly rejected during the cascade evaluation process. Similarly, it also shows that the OpenCV cascade face detection latency pattern is quite similar to our cascade, but violates several times the 40 ms deadline (i.e. the limit for displaying the video at 24 fps) when kernels are serially executed. This slowdown may be also noticeable when the H.264 video decoding latency is also considered for the concurrent OpenCV cascade evaluation.

Usually, thread divergence is one of the most undesired side effects that negatively affects performance when porting serial applications to the multithreaded SIMD architectures of modern GPUs. Due to the fact that each warp thread

Table II
AVERAGE FACE DETECTION TIME PER FRAME (MILLISECONDS)

Movie Trailer	Our cascade		OpenCV cascade	
	Concurrent	*Serial*	*Concurrent*	*Serial*
21 Jump Street	4.17	8.53	10.91	22.12
50/50	4.91	10.17	13.58	27.86
American Reunion	4.01	8.12	9.98	20.12
Bad Teacher	4.8	9.13	12.43	23.37
Friends With Kids	4.68	9.11	12.52	24.05
One For The Money	4.17	8.43	10.72	21.40
The Dictator	4.7	8.99	12.55	22.65
Tim & Eric's Billion Dollar Movie	4.83	9.03	12.56	22.66
Unicorn City	4.23	8.41	11.03	20.99
What To Expect When You're Expecting	4.13	8.52	10.43	20.51

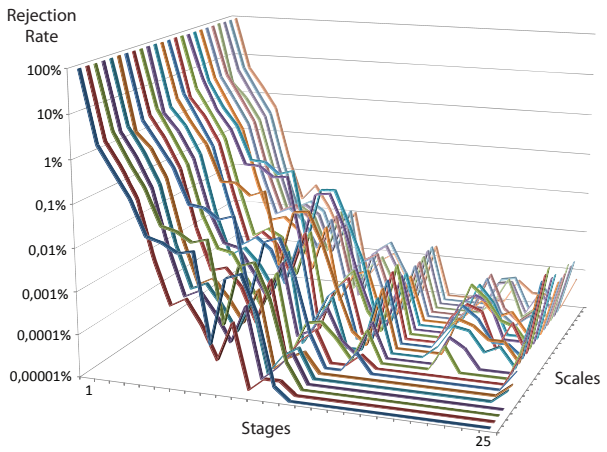

Figure 7. Rejection rates for each cascade stage and image scale for the movie trailer *What To Expect When You're Expecting*.

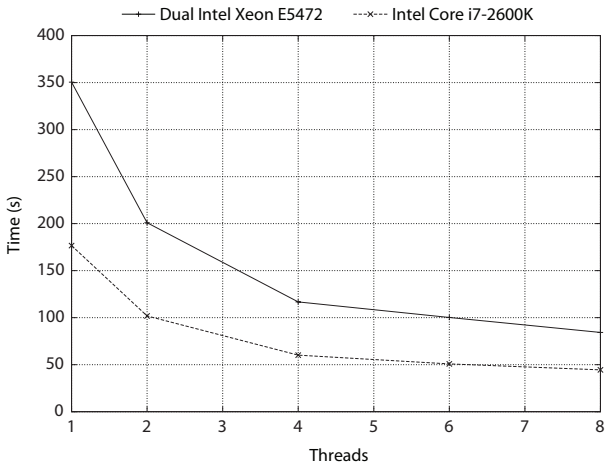

Figure 8. Execution time for a single iteration of the parallelized version of GentleBoost under different SMP scenarios.

evaluates the cascade of classifiers for a different sliding window, our proposed parallelization pattern may potentially yield a high percentage of divergent branches. Under this scenario, multiple threads of a warp could trigger an early exit at the first stages of the cascade during the evaluation, while the others finish in deeper stages. This situation would force some warp threads to stall thus underutilizing the GPU resources.

In order to determine the branch efficiency of the cascade evaluation kernel, we gathered the ratio of non-divergent branches to total branches during execution using the GPU performance counters. The obtained results showed that on average 98.9% of branches were non-divergent.

The reason for this behaviour is that when a given warp evaluates the boosted cascade of classifiers, adjacent integral image elements mostly end at the same stage. Since our parallel face detector stores in the GPU global memory the deepest stage reached during the cascade evaluation process, we decided to keep this information saved in the CPU address space for all pixels and scales across all video frames. The purpose of this task was to study the distribution of the deepest stage reached by all GPU threads when evaluating the pixels of a given video frame for each considered scale.

Figure 7 shows the aggregated results obtained for all the video frames of a selected movie trailer. On average, a thread evaluating a window starting at a given (x, y) pixel yields a rejection rate of 94.52% at the first stage of the cascade, and thus quickly discards non-face regions. Similarly, the average rejection rate for the second stage was 4%, and then it is dramatically reduced for the remaining stages. The results for the other movie trailers were virtually the same.

To conclude, we also evaluated the scalability of the parallelized cascade training algorithm under two different SMP scenarios. Since this process usually takes several days, we decided just to measure the speedup obtained in the first iteration of the main loop of the GentleBoost algorithm. Furthermore, since each iteration of the main loop must test all possible Haar-like feature combinations for all input images, the performance improvement should be noticeable. The algorithm was executed using as an input the dataset described in Section IV, and the number of threads modified

between multiple runs by setting the `OMP_NUM_THREADS` environment variable accordingly. As Figure 8 depicts, the obtained speedup was close to 3.5X in both scenarios when the inner loops were parallelized with 8 threads. Even though we executed GentleBoost on a workstation equipped with two quad-core CPUs (Intel Xeon E5472), a newer single quad-core Intel Core i7-2660K outperformed the latter with a 2X performance improvement on average.

B. Accuracy

There exists several proposals of metrics in the literature to represent the degree of match between an object detection d_i and an annotated region l_j. One of the most used metrics is the ratio of intersected areas to joined areas [29], defined as:

$$S_{square}(d_i, l_j) = \frac{area(d_i) \cap area(l_j)}{area(d_i) \cup area(l_j)} \tag{5}$$

Nevertheless, this score loses significance when the applied preprocessing face alignment techniques are different among cascades. The reason is that the localization of the facial content within the candidate windows may not be aligned in this case. The score based on the ratio of intersected areas also does not account for rotations of the face. Hence, the following distance metric is proposed instead, which is based on the annotated and predicted localization of the eyes:

$$S_{eyes}(d_i, l_j) = \frac{d_{le} + d_{re}}{\min(d_1, d_2)} \tag{6}$$

The d_{le} and d_{re} values are the pixel distances between predicted and annotated locations of the left and right eye in the image, respectively; and d_1, d_2 are the pixel distances between the eyes according to each cascade.

In practice, for each face in an image, the proposed face detection pipeline results in a large number of detection windows at slightly different positions and scales. In order to associate them with the available ground truth annotations, overlapping detections first require to be grouped together. To do so, we consider that two detection windows d_i and d_j overlap whenever $S_{eyes}(d_i, d_j) < 0.5$. Then, an iterative process reduces the initial set of windows by progressively averaging those with the highest overlapping. Finally, each detection window in the reduced set is assigned to the ground truth annotation l_j employing the Hungarian algorithm [30], by establishing the $S_{eyes}(d_i, l_j)$ metric as a cost function.

Based on the resulting associations, the accuracy is evaluated by means of a curve similar to the conventional ROC, following the recommendations discussed in [29]. Each non-matching association is accounted as a false positive (FP), whereas matchings increase the number of true positives (TP). True positive rates (TPR) are obtained by dividing TP by the total number of faces in the ground truth dataset. In addition, a large set of backgrounds (i.e. images not

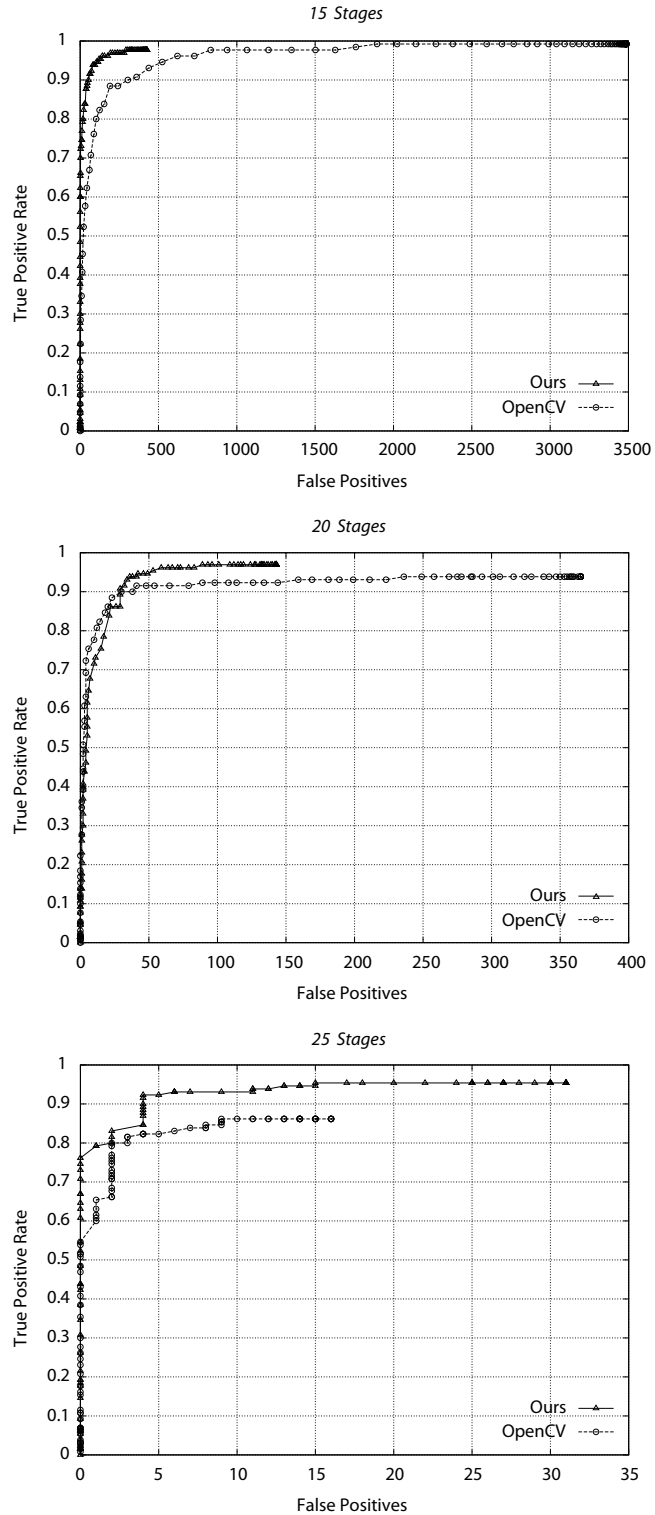

Figure 9. TPR/FP curves for OpenCV feature set and our cascade (15, 20 and 25 stages)

containing faces) has been used to obtain statistics about false positives (FP). Lastly, the resulting curve is plotted by varying a threshold over the detection score, and thus obtaining different combinations of the ratio TPR/FP.

Several tests have been conducted over the subset of visible light mug shot frontal images of the SCFace database [31], which has been increased with 3000 high-resolution background images. The accuracy of the face detection algorithm at stages 15, 20, and 25 is shown in Figure 9 for each of the two cascades. In addition, the level of discrimination increases as more stages are considered. Although the proposed cascade contains less filters, the obtained results show that generally outperforms the OpenCV cascade in terms of TPR/FP.

VII. Conclusions and Future Work

In this paper we have presented a highly parallel GPU face detector that is optimized for HD videos. Our proposal heavily exploits concurrent kernel execution and a customized cascade for dramatically reducing the detection latency, while slightly improving the detection accuracy. The obtained results show that the combination of both techniques yields a 5X speedup over the fastest implementations known to date.

Even though we have relied on our own cascade for achieving these results, the usage of concurrent kernel execution with a fixed-sized sliding window has proven to be useful for increasing the GPU occupancy of any feature dataset (e.g. OpenCV). Furthermore, we also presented a SMP parallel GentleBoost implementation that exploits both task and data-level parallelism for reducing the duration of the cascade training process. With the combination of OpenMP and the SSE4 vector extensions we showed that it was possible to obtain a 3.5X speedup over the conventional serial implementation with little effort.

Finally, as a future work we plan to port our parallel face detector to other platforms such as the Intel Many Integrated Core (MIC) architecture, and further improve the accuracy of our feature set with soft cascades [32].

Acknowledgements

This work has been partially supported by the European Commission in the context of the HiPEAC3 Network of Excellence (FP7/ICT 287759), the Spanish Ministry of Education (TIN2007-60625, TEC2010-21040-C02-01, CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980), and Herta Security.

References

[1] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011.

[2] T. Ahonen, A. Hadid, and M. Pietikäinen. Face Recognition with Local Binary Patterns. *8th European Conference on Computer Vision*, pages 469–481, 2004.

[3] D.G. Lowe. Object Recognition from Local Scale-invariant Features. In *7th IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157. IEEE, 1999.

[4] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *9th European Conference on Computer Vision*, pages 404–417, 2006.

[5] C. Zhang and Z. Zhang. A Survey of Recent Advances in Face Detection. *Microsoft Research Techical Report MSR-TR-2010-6*, 2010.

[6] J. Friedman, T. Hastie, and R. Tibshirani. Additive Logistic Regression: A Statistical View of Boosting. *The Annals of Statistics*, 28(2):337–407, 2000.

[7] P. Viola and M. Jones. Robust Real-time Object Detection. *International Journal of Computer Vision*, 57(2):137–154, 2001.

[8] Y. Freund and R. Schapire. A Desicion-Theoretic Generalization of On-Line Learning and an Application to Boosting. In *Computational Learning Theory*, pages 23–37, 1995.

[9] B. Sharma, R. Thota, N. Vydyanathan, and A. Kale. Towards a Robust, Real-Time Face Processing System Using CUDA-enabled GPUs. In *International Conference on High Performance Computing*, pages 368–377, 2009.

[10] D. Hefenbrock, J. Oberg, N. Thanh, R. Kastner, and S.B. Baden. Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs. In *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 11–18, 2010.

[11] A. Obukhov. Haar Classifiers for Object Detection with CUDA. In *GPU Computing Gems Emerald Edition*, pages 517–544. Morgan Kaufmann, 2011.

[12] A. Herout, R. Josth, R. Juranek, J. Havel, M. Hradis, and P. Zemcik. Real-time Object Detection on CUDA. *Journal of Real-Time Image Processing*, pages 1–12, 2011.

[13] J. Sochman and J. Matas. WaldBoost-Learning for Time Constrained Sequential Detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 150–156, 2005.

[14] M. Hradis, A. Herout, and P. Zemcik. Local Rank Patterns–Novel Features for Rapid Object Detection. *Computer Vision and Graphics*, pages 239–248, 2009.

[15] F. Jargstorff and E. Young. CUDA Video Decoder API. *NVIDIA Corporation*, 2010.

[16] Parallel Thread Execution ISA Version 3.0. *NVIDIA Corporation*, 2012.

[17] M. Harris, S. Sengupta, and J.D. Owens. Parallel Prefix Sum (scan) with CUDA. In *GPU Gems 3*, pages 851–876. Addison Wesley, 2007.

[18] S. Sengupta, M. Harris, and M. Garland. Efficient Parallel Scan Algorithms for GPUs. Technical report, NVIDIA Corporation, 2008.

[19] G. Ruetsch and P. Micikevicius. Optimizing Matrix Transpose in CUDA. Technical report, NVIDIA Corporation, 2009.

[20] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast Summed-Area Table Generation and its Applications. In *Computer Graphics Forum*, volume 24, pages 547–555, 2005.

[21] C.H. Messom and Barczack A.L. High Precision GPU based Integral Images for Moment Invariant Image Processing Systems. *Electronics and New Zealand Conference*, 2008.

[22] B. Bilgic, B.K.P. Horn, and I. Masaki. Efficient Integral Image Computation on the GPU. In *Intelligent Vehicles Symposium*, pages 528–533. IEEE, 2010.

[23] D. Oro, C. Fernández, J. R. Saeta, X. Martorell, and J. Hernando. Real-time GPU-based Face Detection in HD Video Sequences. In *IEEE International Conference on Computer Vision Workshops*, pages 530–537. IEEE, 2011.

[24] Eigen C++ Template Library. *http://eigen.tuxfamily.org/*.

[25] iTunes Movie Trailers. *http://trailers.apple.com/*.

[26] libavcodec. *http://www.libav.org/*.

[27] Compute Command Line Profiler. *NVIDIA Corporation*, 2011.

[28] R. Lienhart and J. Maydt. An Extended Set of Haar-like Features For Rapid Object Detection. In *Proceedings of the International Conference on Image Processing*, volume 1, pages 900–903. IEEE, 2002.

[29] V. Jain and E. Learned-Miller. FDDB: A Benchmark for Face Detection in Unconstrained Settings. Technical report, University of Massachusetts-Amherst, 2010.

[30] H.W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[31] Surveillance Cameras Face Database. *http://www.scface.org/*.

[32] L. Bourdev and J. Brandt. Robust Object Detection Via Soft Cascade. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 236–243, 2005.