Real-time GPU-based Face Detection in HD Video Sequences

David Oro † Carles Fernández † Javier Rodríguez Saeta † Xavier Martorell ‡ Javier Hernando ‡
† Herta Security, Barcelona, Spain
‡ Universitat Politècnica de Catalunya, Barcelona, Spain
david.oro@hertasecurity.com xavim@ac.upc.edu javier@tsc.upc.edu

Abstract

Modern GPUs have evolved into fully programmable parallel stream multiprocessors. Due to the nature of the graphic workloads, computer vision algorithms are in good position to leverage the computing power of these devices. An interesting problem that greatly benefits from parallelism is face detection. This paper presents a highly optimized Haar-based face detector that works in real time over high definition videos. The proposed kernel operations exploit both coarse and fine grain parallelism for performing integral image computations and filter evaluations, thus being beneficial not only for face detection but also for other computer vision techniques. Compared to previous implementations, the experiments show that our proposal achieves a sustained throughput of 35 fps under 1080p resolutions using a sliding window with step of one pixel.

1. Introduction

In recent years, face detection and identification technology has experienced a huge leap in terms of improved accuracy and throughput. Part of this success is due to the fact that the semiconductor industry has been able to deliver faster microprocessors clocked at higher frequencies. This increase in raw performance for single-threaded applications also has its roots in the growing complexity of the microarchitectures of general purpose processors. Techniques such as deep pipelining, out-of-order and superscalar execution, and simultaneous or speculative multithreading have enabled sustained increases in both the instruction level parallelism and the issue width of these highly complex cores.

At the same time, computer vision algorithms have successfully exploited the increase of serial performance, by carrying out hundreds of billions of operations to match specific patterns within an image. Applications ranging from classic object recognition to advanced video analytics were made possible. Particularly, smart video surveillance algorithms aim at analyzing 24/7 continuous video broadcasted from multiple IP CCTV cameras, thus requiring a high amount of computing power, especially when it is required to meet specific latency and accuracy constraints. From the perspective of latency, a video surveillance system has to analyze real-time image sequences without missing any video frame. In order to guarantee SLAs and trigger the appropriate alarms, computer vision algorithms should meet a tight deadline ensuring that all required computations can sustain a frame rate of at least 25 fps.

On the other hand, recent advances in CCD and active-pixel CMOS sensors have substantially reduced the cost of deploying 1080p HDTV cameras. By using high resolution images, it is now possible to detect features of distant faces in highly crowded environments (*e.g.* stadiums, airports or train stations) and use such increased resolutions to effectively improve face identification algorithms.

Unfortunately, it is not expected anymore to dramatically increase the throughput of single-threaded object detectors just by executing them in the latest available CPU. The so-called *power wall* and the impossibility of clocking out-of-order microarchitectures at ever-increasing frequencies have been the main reasons for the major multicore shift experienced by the CPU industry.

Unlike multicore CPUs, GPU microarchitectures do not rely on big L2 and L3 caches for hiding latencies when accessing DRAM memories. They spend instead a large extension of the die area in ALUs rather than in caches, and hide memory access latencies simply by overlapping them with arithmetic computations from multiple threads. In order to exploit these throughput-oriented architectures, the programmer must explicitly expose data-level parallelism by mapping *kernel* functions to a collection of data records or *streams*.

In this work we present a massively parallel stream-based implementation of a Haar-based face detector that targets NVIDIA GPUs. The proposed parallel processing pipeline is designed from scratch for detecting faces in real-time in H.264 high definition 1080p video sequences, and returning their precise location within the image for further identification. Our implementation processes HDTV video sequences at a sustained rate of 35 fps using hard runtime

2011 IEEE International Conference on Computer Vision Workshops
978-1-4673-0063-6/11/$26.00

constraints (sliding windows with step of one pixel and 32 different scales).

This paper is structured as follows: Section 2 discusses recent advances and state-of-the-art implementations of low-latency face detection algorithms based on the framework originally described by Viola and Jones [1]. Section 3 describes the proposed GPU-based parallel pipeline for performing face detection. The optimized implementation of this pipeline is described in two parts: Section 4 proposes an implementation of the integral image computation based on *multiscan* operations, and Section 5 discusses the parallel implementation strategy followed for the Haar cascade evaluation process. An intensive evaluation of these optimizations is carried out in Section 6. To conclude, Section 7 draws some final conclusions and describes the future work.

2. Related work

There has been little work in the literature during the last years about real-time face detection at HDTV resolutions. Cho *et al*. [2] presented a FPGA-based face detection system based on Haar classifiers capable of detecting faces at 640×480 resolutions at 7.5 fps. Hefenbrock *et al*. [3] proposed a stream-based multi-GPU implementation on 4 cards that achieved 15.2 fps. However, the integral image computation was not parallelized, and Haar features were accessed from the shared memory of each streaming multiprocessor (SM) and not from the constant memory, which is specifically designed for broadcasting values to a thread warp. Kong *et al*. [4] described another GPU-based implementation that offered a latency of 197 ms (0.5 fps) when detecting 48 faces at 1280×1024 resolution. Herout *et al*. [5] proposed a GPU-based face detector based on *local rank* patterns as an alternative to the commonly used Haar wavelets [6].

Finally, Sharma *et al*. [7] presented a working CUDA implementation that achieved a peak throughput of 19 fps under a resolution of 1280×960 pixels. They proposed a naive parallel integral image kernel to perform both row-wise and column-wise prefix sums, by fetching input data from the off-chip texture memory cached in each SM. Unlike other implementations, the Haar cascade evaluation was parallelized using a fixed-size sliding window, and the input images were sequentially resized to deal with multiple scales.

In the present work we overcome the bottlenecks and limitations of the parallel integral image computation used by Sharma *et al*. [7]. We also propose a parallel algorithm for evaluating Haar filters that fully exploits the underlying microarchitecture of the NVIDIA GF100 core. The experimental results show that our parallel face detection framework beats in performance all implementations found to date in the literature, achieving a sustained frame rate of 35 fps at a resolution of 1920×1080 while decoding H.264 video in real-time.

3. Proposed face detection pipeline

In order to implement face detection, we follow a parallelization strategy that tries to simultaneously maximize GPU occupancy and minimize the amount of memory transfers between CPU and GPU. In addition, since the targeted GPU microarchitecture features a heterogeneous cache memory hierarchy [8], the Haar feature set and the image to be computed should be manually transferred to the appropriate cache type to achieve the highest throughput.

As shown in Figure 1, the pipeline starts from a video input. If the input is encoded using standard codecs, *e.g*. MPEG2 or MPEG4-AVC/H.264, it is then possible to use the NVCUVID API for decoding the video in the GPU [9]. This API exposes the on-die video decoder processor (VP) to the programmer, and allows interoperability between the hardware-decoded video frames and CUDA kernel computations.

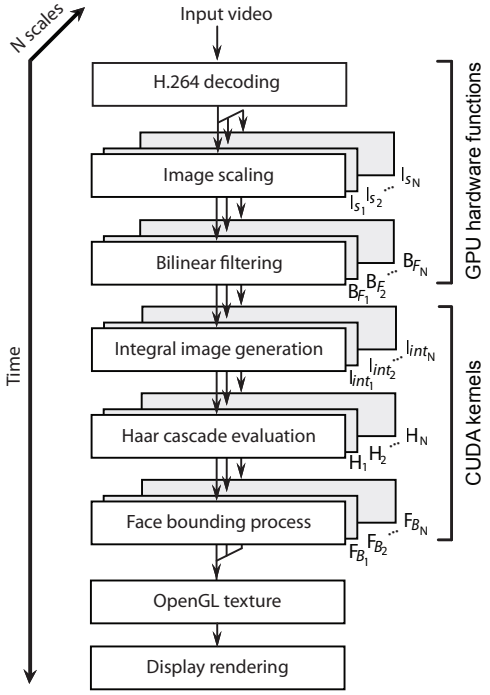

Figure 1. Proposed pipeline for parallel face detection

The following step after decoding the video frame is computing the integral image I_{int} in parallel. At this point, there are two possible implementations for the process H that evaluates the cascade of Haar filters. The first alternative scales the sliding window and requires resizing each filter accordingly, whereas the second one scales the image. As it will be discussed in Section 5, the best option is rescaling images instead of filters. Each newly scaled image I_s is filtered to avoid aliasing, *e.g*. by using the bilinear filtering fixed-functions B_F available in the texturing units of the

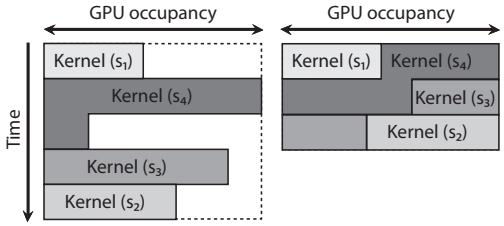

Figure 2. Serial (left) vs. concurrent (right) kernel execution

GPU, and its corresponding integral image is recomputed for each scale.

The last step of the pipeline returns the coordinates of detected faces to the CPU for further processing. The detected faces are bounded using a CUDA kernel F_B, and these results are bypassed to the conventional OpenGL pipeline using a pixel buffer object to display the results.

All kernel computations in the GPU exploit both coarse-grain and fine-grain parallelism. Coarse-grain parallelism is achieved by simultaneously launching and executing the same kernel operation for each scaled image from the same CUDA context. Additionally, fine-grain parallelism is also exploited at thread-level within each CUDA kernel. Since the Fermi microarchitecture supports concurrent kernel execution [8], the occupancy of GPU resources is maximized even for those kernels dealing with small scales, by executing them in parallel as seen in Figure 2.

4. Parallel integral image computation

Computing integral images is an expensive task within the face detection process. There exists literature on how to optimize this task via parallelization under both stream-based and multi-core processors [10, 11, 12]. However, most existing work relies on the observation made by Meson *et al*. [11] that integral images can be computed using standard exclusive prefix sum operations followed by matrix transpositions. Since this algorithm is meant to be executed on GPUs, it is possible to preserve data locality in on-die caches by performing row-wise prefix-sums and two matrix transpositions, as opposed to the row-wise and column-wise prefix sum operations initially proposed by Sharma *et al*. [7].

The prefix sum or *scan* is a data-parallel primitive applied to a given stream, in which each element is generated by summing the elements up to its index. A naive parallel implementation computes the prefix-sum in $\log_2 n$ steps, by first summing in parallel all pairs of contiguous elements to produce $n/2$ additions, and recursively summing the produced pairs until a final single sum remains [13]. This naive implementation performs $O(n \log_2 n)$ total additions, whereas a single-threaded CPU implementation would only require $O(n)$ operations.

For better exploiting the underlying GPU microarchitecture, we take an additional effort and follow a divide-and-conquer hierarchical approach based on the Hillis-Steele algorithm [14, 15], which implements the scan at different granularities. In this implementation, input data is fetched in parallel from the GPU DRAM and then stored in the on-die shared memory of each SM before starting any computation.

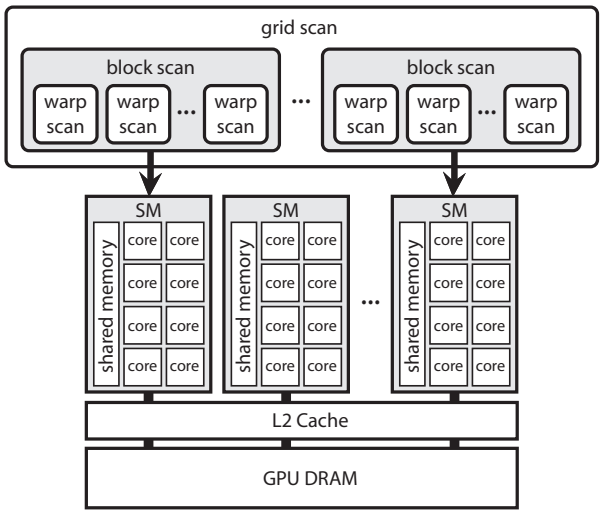

Figure 3. Divide-and-conquer approach for the parallel prefix sum (*scan*) operation and the underlying GPU microarchitecture

As Figure 3 shows, the high-level *block scan* operation is composed of several *warp scan* primitives. The purpose of this block-wise operation is to compute the *scan* across a block of threads. A grid of intra-block primitives (*grid scan*) is then used to finish the prefix sum computation of a stream of arbitrary length. In order to exploit coarse-grain parallelism at row level, the integral image computation relies on the *multiscan* kernel operation. This operation carries out the *scan* in parallel for each row of the input image.

At the lowest level, a *warp scan* primitive computes the prefix sum only for the threads referenced within the warp. As depicted in Figure 4, the *warp scan* primitive performs the prefix sum operation in parallel for an input vector of 32 elements in $\log_2 32$ steps. For each step, a subset of threads in the warp performs a partial sum. At the end of the 5th step, the output of the algorithm will be the prefix sum for the 32-element vector.

In the Fermi core, SIMT instructions perform the same computation across a warp or group of 32 threads and simultaneously store the results in registers without needing additional thread barrier instructions [8]. For this reason, each step of the *warp scan* primitive can be implemented using a sequence of `ld.shared.u32`, `add.u32` and `st.shared.u32` SIMT instructions. The parallel access pattern depicted in Figure 4 ensures memory coalesc-

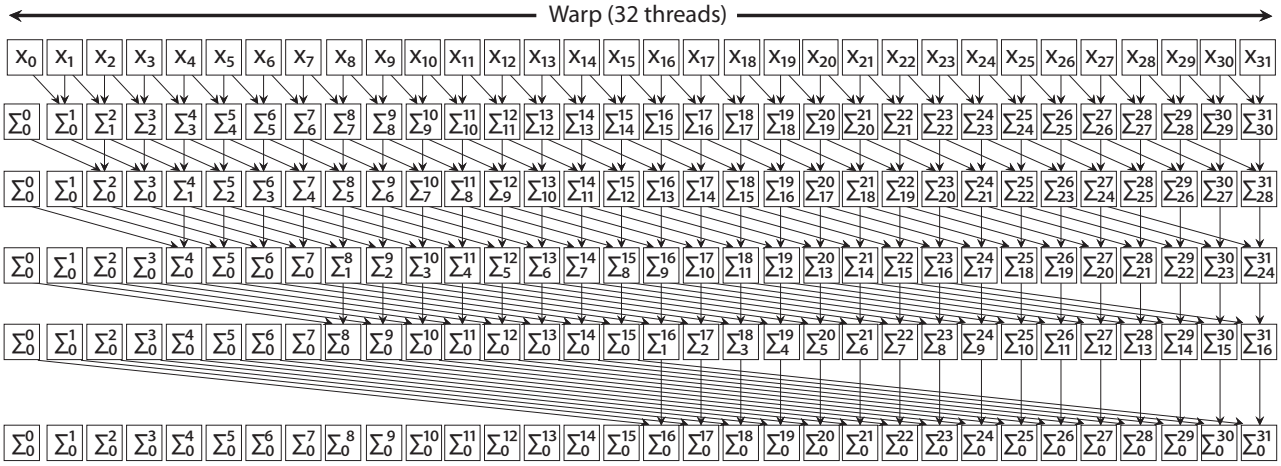

Figure 4. Parallel *warp scan* primitive for a 32-element input vector x

ing, thus maximizing both bandwidth and throughput.

The integral image I_{int} is obtained by subsequently performing a *multiscan* operation, a parallel matrix transposition, a second *multiscan* and a second transposition:

$$I_{int} = multiscan\left(multiscan\left(I\right)^T\right)^T$$

Transposing the matrices is necessary to avoid column-wise scanning, so that data locality is better exploited when accessing the shared memory. Memory coalescing and bulk transfers are required to achieve the maximum bandwidth for the GPU, so a natural way of transposing matrices is to divide them into equally-sized blocks and to copy data from the off-chip DRAM to the on-die shared memories. This involves using a barrier instruction after data is transferred to the shared memory of each SM. Synchronization is required to guarantee that all threads constituting the CUDA block start with the transposition immediately after all memory transfers have concluded.

Generally, the input matrix is not a multiple of the warp width. To solve this issue, before transferring memory from the CPU address space to the GPU, the matrix is zero-padded until becoming square. Even though this padding process causes additional memory transfers, the final throughput of the transposition will be much better, since all accesses will be properly coalesced and aligned.

5. Parallel Haar cascade evaluation

Once the integral image of a given frame is computed, the cascade of Haar filters has to be evaluated. This step is the most resource-intensive part of the face detection algorithm, since it involves hundreds of billions of operations per image frame in HD video sequences.

There are two alternative parallelization strategies to tackle scaling during filter evaluation. The first one is to parallelize the serial algorithm proposed by Viola and Jones [1], which consists of resizing the filters to the desired scanning resolution. Unfortunately, this technique is inefficient when implemented in CUDA. Since all filters must be scaled up to the size of the sliding window, the occupancy of the CUDA cores will be extremely low if each thread evaluates the Haar filter cascade for a given sliding window position and resolution. As shown in the following equation, the number of potential simultaneous threads $N_{threads}$ quadratically decreases as the size of the sliding window $W \times W$ increases by a $scale$ factor:

$$N_{threads} = \left\lceil \frac{I_{width} \cdot I_{height}}{scale^2 \cdot W^2} \right\rceil$$

Therefore, when detecting faces of 96×96 pixels in a 1920×1080 image, there are only 225 threads competing for the hardware resources. This amount of threads is clearly insufficient for keeping the 512 cores of the GF100 microarchitecture busy, especially when threads stall due to data dependencies or pending load instructions. The situation is even worse for higher resolutions of faces.

Hence, the main objective is to create as many threads evaluating the cascade as possible. In order to meet this goal, we propose using small fixed-sized sliding windows and scaling instead the input image. As a consequence, integral images need to be computed for each newly scaled image, whereas the traditional algorithm computes it once.

In order to obtain the highest performance during the evaluation of the cascade, the integral image data should be moved to the on-die shared memory before any computing-intensive operation starts. The best way to do it is using the divide-and-conquer approach, where each thread from the cooperative array transfers a portion of image from the global memory to the shared memory of the corresponding SM. The size of memory chunks has to be at least four times

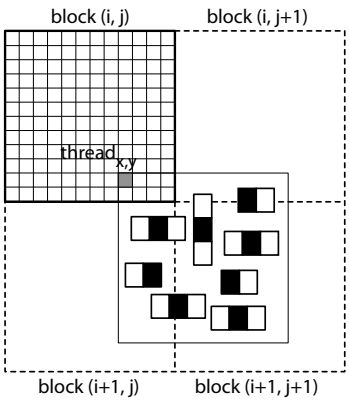

Figure 5. Evaluation area considered by a thread in a CUDA block

larger than that of the sliding window. This is because the threads of a block (i, j) in the grid have to bring pixels from contiguous blocks to the shared memory in order to evaluate the sliding window, see Figure 5.

After these memory transfers have been completed, the filters of the Haar cascade need to be evaluated. This evaluation is both a memory and an arithmetic intensive operation. Therefore, the right approach should involve moving the Haar features to the on-die shared memory. Unfortunately, this is unfeasible for two reasons. First, the selected feature set [16] has a size of 106 KB and the available shared memory in a SM is 48 KB. Second, all warp threads will stall, since each thread will try to access the same memory address (*e.g.* the first filter of the first stage), thus producing bank conflicts. To deal with these situations, the GF100 microarchitecture provides the constant memory, which is specifically designed for broadcasting values for all warp threads that simultaneously access the same address.

Unlike the shared memory, the constant memory is read-only and must be allocated and initialized before launching a kernel. Although the aggregated size of the constant memory is 128 KB (8 KB per SM), the CUDA programming model restricts the available size to 64 KB, so the feature set must be stored compressed. Since thresholds are encoded using double precision floating-point numbers, they can be re-encoded in 32 bits at the cost of slightly losing precision. Similarly, not all the bits required for the coordinates, dimensions and weight values are significant. Given that the training images used for the cascade had a size of 24×24 pixels, two 16-bit unsigned integers suffice for encoding a feature.

6. Experimental results

In order to evaluate the performance of the previously described parallel algorithms, multiple execution tests were performed in the same computer. The selected platform featured an Intel Core i5-760 2.8 GHz, quad-core CPU and an NVIDIA GTX 470 graphics card.

All GPU applications were compiled for the CUDA Toolkit 4.0 with the `-O3` flag and targeted the `sm_20` architecture. The underlying OS was powered by the Linux Kernel 2.6.35 and compiled for the x86-64 architecture. On the other hand, GCC v.4.4.5 was used for linking the final application and for building the CPU tests. It should be noted that all GPU benchmarks were performed without considering the time spent on memory transfers between the CPU and the GPU. This assumption is valid since the final GPU-based face detector will start performing computations only once the image frame is available in the off-chip GPU DRAM memory.

The cascade used for the Haar evaluation process was the frontal feature set developed by Lienhart *et al*. [16] and distributed in the OpenCV framework. It has 2913 filters and is organized into 25 stages. Finally, the size of the training images was 24×24 pixels.

6.1. Integral image

In order to evaluate our parallel integral image implementation, we executed the *multiscan* algorithm for different image sizes in both CPU and GPU. The CPU implementation was a single-threaded application that carried out a recursive scan in $O(n)$ steps for an $n \times n$ matrix whereas the GPU implementation was the same as described in Section 4. Since each row involves $O(\log_2 n)$ steps, the GPU implementation is asymptotically fitted by $O(n \log_2 n)$. On the other hand, input images were matrices randomly populated with unsigned 32-bit integers that ranged from 30×30 to $10^4 \times 10^4$ pixels.

As depicted in Figure 6, the GPU *multiscan* scales well with the image size. For a 100 megapixel image it is executed in only 43 ms, 66 times faster than the CPU version.

Although the GPU algorithm scales well with the size of the input image, it is slower than the CPU version for small images. This effect can be clearly seen in Figure 7 where the execution time for the GPU is constant for images smaller or equal than 100×100 pixels. The reason for this behavior is related to the overhead produced by the CUDA kernel launching process. If the image to be processed has few elements, it is not possible to hide the latencies of kernel launches with arithmetic computations. As a result, the CPU will beat the GPU for images smaller than 60×60 pixels where the time spent by the CUDA runtime for allocating all required resources and internal data structures is higher than the time spent by the CUDA cores for computing the data stream.

The performance of the *multiscan* for non-square $n \times m$ matrices was also analyzed. Since kernel launches are expensive operations, in theory a *multiscan* of a 10000×100 matrix would yield a higher execution time than that of a 100×10000. Also, the first approach requires more ker-

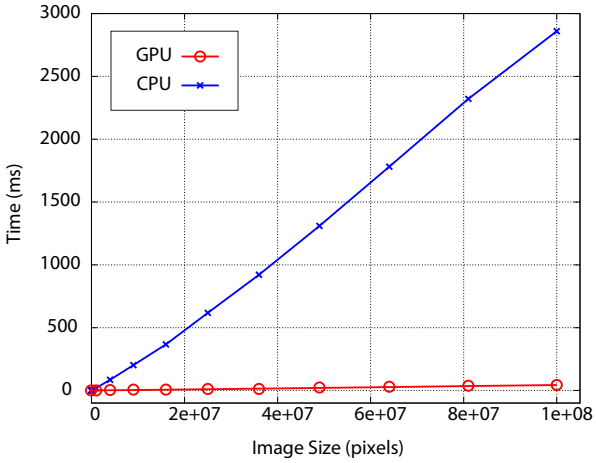

Figure 6. Execution time of the *multiscan* operation

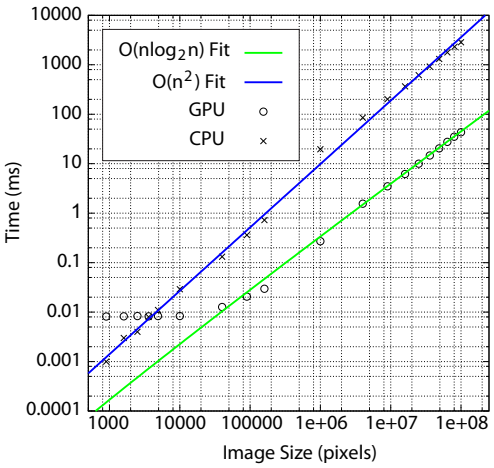

Figure 7. Execution time of *multiscan* (log scale for x and y axis)

nel launches than the second and computes less work in each kernel. As shown in Table 1, this does not behave as expected. The execution time grows slightly faster if the column size (m) is kept constant and the number of rows (n) increased, rather than the opposite, but only while $m \leq 1000$ and $n \leq 10000$. This effect is amplified as n (kernel launches) grows and may be related to the implementation of the CUDA runtime.

Rows (n)	Columns (m)				
	1	10	100	1000	10000
1	0.006860	0.006820	0.006780	0.006740	0.021770
10	0.006970	0.006880	0.006860	0.007010	0.024120
100	0.006900	0.007400	0.007290	0.014300	0.192690
1000	0.045060	0.049580	0.051780	0.248030	**3.874300**
10000	0.421470	0.47058	0.507210	**2.604200**	40.690800

Table 1. *Multiscan* execution time (ms) for the GPU

In addition to the *multiscan*, the matrix transposition was also evaluated. Due to the nature of our GPU parallel implementation, the optimal width of the CUDA cooperative thread array must be experimentally determined for achieving the maximum performance. Since the purposed face detection application should work at HDTV resolutions, we determined the block size yielding the best results for resolutions ranging from 1280×720 up to 2048×2048 pixels. Furthermore, the transposition algorithm was deliberately modified with the purpose of analyzing the effect of both coalesced and uncoalesced memory accesses under the GF100 microarchitecture.

As Figure 8 shows, the lowest execution time is obtained for blocks of 16×16 threads when performing coalesced memory accesses. The reason for this is related to the number of bank ports available in the on-die shared memory.

Finally, the parallel algorithms for *multiscan* and transposition are sequentially combined to obtain the $n \times m$ integral image. The performance of the whole process has been

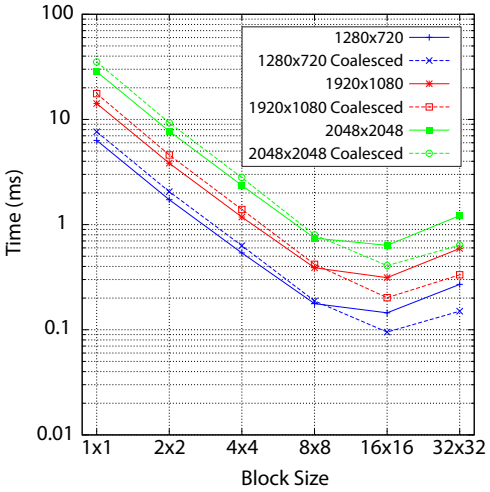

Figure 8. Matrix transposition for different CUDA block sizes

compared against a sequential $O(n \cdot m)$ CPU implementation, which does not perform any *scan* or transposition for computing the integral image. It is worth noting that sequentially adding each element of the matrix while keeping the accumulated sum in a variable avoids unnecessary loop iterations. Both implementations were tested with padded images ranging from 256×256 up to 8192×8192 pixels.

The obtained speed up for the GPU is not as high as expected if we take into account the results of each intermediate step of the parallel implementation (see Figure 9). On average, the GPU implementation is 2.5 times faster than the CPU. However, this does not happen with images smaller than 256×256 pixels; at these low resolutions, the CPU is on average 15% faster than the GPU. Again, the reason behind this behavior is that when the random values of the small input matrices are created from CPU registers, they may never leave the on-die L2 and L3 caches, and they

even fit in the L1 data cache. Nevertheless, the approach of caching the complete working set is not sustainable for high resolutions and even though the CPU still benefits from aggressive memory prefetching, the GPU speed up grows with the size of the input image.

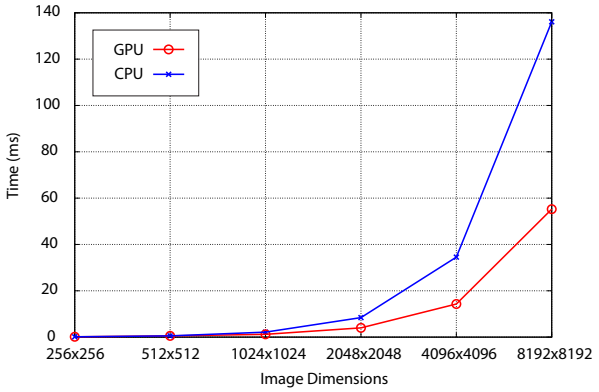

Figure 9. Integral image execution time

To conclude, the average latency obtained for computing the integral image at 1920×1080 pixels in the GPU was 2.3 ms where 0.61 ms were spent for each *multiscan* operation and 0.54 ms for each matrix transposition.

6.2. Optimal block size for filter evaluation

Prior to the performance evaluation of the parallelized Haar cascade filtering, the optimal block size has to be estimated. This size is also used for computing the dimensions of the memory chunks that each thread has to bring to the shared memory.

In this way, the occupancy of the GPU is computed by an analytical model that takes into account register and shared memory usage and the number of threads that constitute a block for a given kernel. By following this approach, we experimentally determined the GPU occupancy for the Haar evaluation kernel. As Figure 10 depicts, there are multiple combinations that achieve the maximum occupancy for this code. However, since the sliding window should be greater or equal than 24×24 elements and square, the only alternative is to use $28 \cdot 28 = 784$ threads per block.

6.3. Face detection

Once the optimal block size was selected, several benchmarks were conducted in order to characterize the latency of the face detection algorithm. These tests were executed in the same computer as that used in Section 6.1 and combine both the integral image computation and the Haar cascade process. In addition, the final algorithm carries out face detection by progressively downscaling the input image frame 32 times, and simultaneously launching the CUDA kernels for each one of the 32 considered scales in parallel.

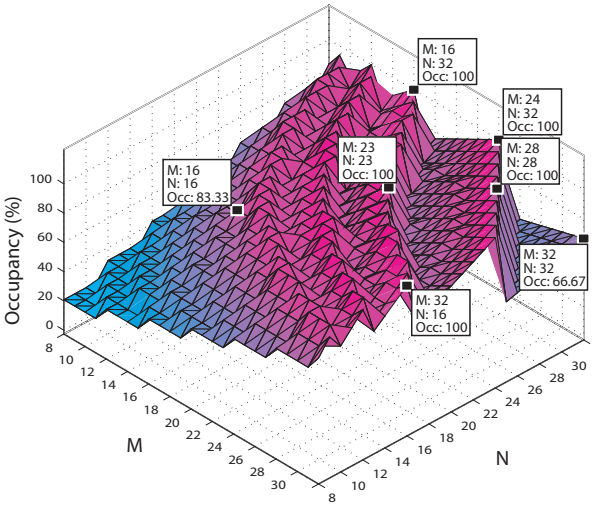

Figure 10. GPU occupancy for blocks of threads of different sizes

We were not able to find any publicly available databases of high definition videos for evaluating the performance of the final face detection system. The considered benchmarks are instead a collection of high definition H.264 movie trailers and music clips carefully selected for stressing the system, *i.e.* containing a large number of faces. All these videos were downloaded from the YouTube website and feature a resolution of 1920×1080 pixels.

Figure 11 shows the face detection execution time at each frame of two selected videos presenting different characteristics within the chosen collection. The first one (*Shakira – Waka Waka*) features panoramic views in highly crowded soccer stadiums, reaching hundreds of faces in some frames. On the other hand, the second video (*Andreea Balan – Trippin'*) peaks at 6 simultaneous faces at most. Both videos present very frequent transitions and parts with sudden motion, and the scales of faces roughly range from 30 up to 1000 pixels vertically. Even though the face detection slows down the video playback and violates the 40 ms deadline (25 fps) on a few occasions, most of the time the obtained throughput is close to 40 fps.

The performance of the algorithm regarding face detection is exactly equivalent to that shown by the detector in [16] when specifying the same tuning parameters. That includes a step size of one pixel for the sliding window, and a total of 32 scale reductions using a scale factor of 1.1.

7. Conclusions and future work

We have presented a highly optimized parallel implementation of a Haar-based face detector, which analyzes high-definition 1080p video at a sustained rate of 35 fps for generic sequences containing multiple faces. Our face detection implementation is, to the best of our knowledge, the first one processing real-time HDTV video with a sliding

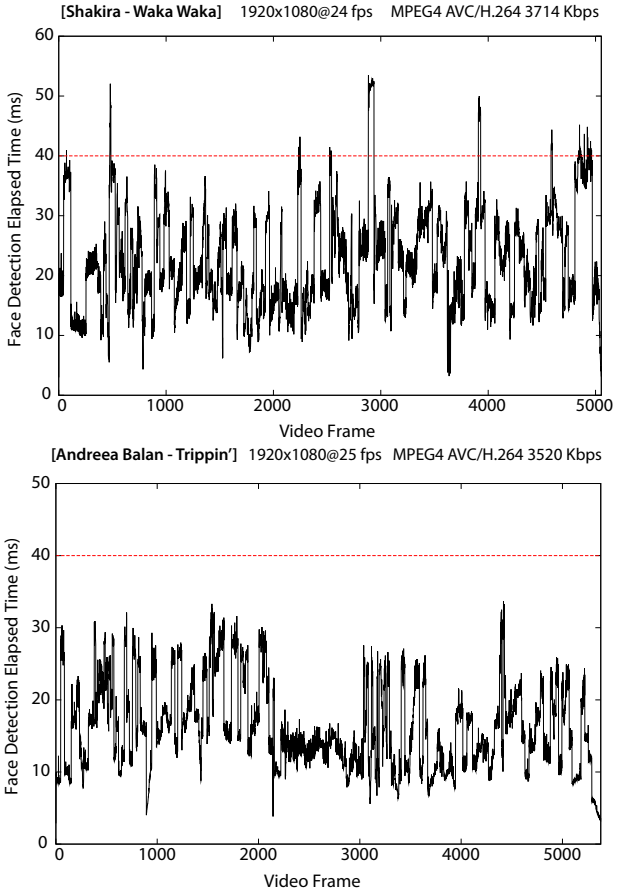

Figure 11. Execution time for two different HD music videos

window of dense step size (1 pixel).

The efficient implementation of the integral image paradigm via row *multiscan* does not restrict to Viola-Jones face detection. It directly benefits a wide variety of computer vision techniques, including Randomized Decision Trees and Forests and methods based on more complex descriptors like HOG and SURF. Moreover, we proved that the combination of a smart usage of the on-die caches, fixed-size sliding windows and image scaling based on texture fetches maximize the GPU occupancy, thus increasing the evaluation throughput of Haar filter cascades.

Further steps include an efficient implementation of the non-maxima suppression process. In addition, a previous stage for image enhancement is required in order to overcome challenging lighting conditions.

Acknowledgements

This work has been partially supported by the European Commission in the context of the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the Spanish Ministry of Science (TIN2007-60625, TEC2010-21040-C02-01), the Generalitat de Catalunya (2009-SGR-980), and Herta Security.

References

[1] P. Viola and M. Jones. Robust Real-time Object Detection. *International Journal of Computer Vision*, 57(2):137–154, 2002. 2, 4

[2] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner. FPGA-based Face Detection System Using Haar Classifiers. In *Proceeding of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 103–112. ACM, 2009. 2

[3] D. Hefenbrock, J. Oberg, N. Thanh, R. Kastner, and S.B. Baden. Accelerating Viola-Jones Face Detection to FPGA-level Using GPUs. In *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 11–18. IEEE, 2010. 2

[4] J. Kong and Y. Deng. GPU Accelerated Face Detection. In *International Conference on Intelligent Control and Information Processing*, pages 584–588. IEEE, 2010. 2

[5] 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, 2010. 2

[6] M. Hradis, A. Herout, and P. Zemcik. Local Rank Patterns: Novel Features for Rapid Object Detection. *Computer Vision and Graphics*, pages 239–248, 2009. 2

[7] 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. IEEE, 2009. 2, 3

[8] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, 2011. 2, 3

[9] F. Jargstorff and E. Young. CUDA Video Decoder API. *NVIDIA Corporation*, 2008. 2

[10] 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. 3

[11] 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. 3

[12] N. Zhang. Working Towards Efficient Parallel Computing of Integral Images on Multi-core Processors. In *International Conference on Computer Engineering and Technology*, volume 2, pages V2–30. IEEE, 2010. 3

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

[14] W.D. Hillis and G.L. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986. 3

[15] S. Sengupta, M. Harris, and M. Garland. Efficient Parallel Scan Algorithms for GPUs. *NVIDIA Technical Report NVR-2008-003*, 2008. 3

[16] 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. 5, 7