GPU ACCELERATED REAL-TIME OBJECT DETECTION ON HIGH RESOLUTION VIDEOS USING MODIFIED CENSUS TRANSFORM

Salih Cihan Tek and Muhittin Gökmen
Department of Computer Engineering, Istanbul Technical University, Maslak, Istanbul, Turkey

Keywords: Object Detection, Face Detection, GPU Acceleration, CUDA.

Abstract: This paper presents a novel GPU accelerated object detection system using CUDA. Because of its detection accuracy, speed and robustness to illumination variations, a boosting based approach with Modified Census Transform features is used. Results are given on the face detection problem for evaluation. Results show that even our single-GPU implementation can run in real-time on high resolution video streams without sacrificing accuracy and outperforms the single-threaded and multi-threaded CPU implementations for resolutions ranging from 640×480 to 1920×1080 by a factor of 12-18x and 4-6x, respectively.

1 INTRODUCTION

Real-time object detection has been a very active topic of research in the last decade. The primary reason of this interest on the subject is the number of its possible real-world applications both in commercial and non-commercial systems. Most of the complicated applications like virtual reality, surveillance, video conferencing and robotics require the system to be able to run in real-time, making the speed of the object detector as important as the accuracy.

Numerous algorithms have been developed that can run in real-time provided that the input resolution is relatively small. Most of them are descendants of the object detection algorithm proposed in (Viola and Jones, 2004), which is based on scanning the image with a sliding window at multiple scales and using a cascaded classifier on each location to obtain the locations containing the searched object. One example of these algorithms is (Fröba and Ernst, 2004), the algorithm this research is based on. Even though these algorithms are fast, they are not fast enough to run in real time on video streams having a high resolution like 1280×720 without sacrificing the accuracy or generalization ability of the system in question. Considerable amount of effort has been made to speed up these algorithms, but algorithmic modifications by themselves are not enough to get drastic speed improvements required.

Another approach that is becoming increasingly common is to accelerate the algorithms using the GPU. In (Sharma et al., 2009), the authors followed this approach by implementing the Viola-Jones algorithm using CUDA. They achieved a detection rate of 19 Frames Per Second (FPS) on a video stream of resolution 1280×960 on a GTX 285 GPU. Although there is an important increase in speed, having %81 accuracy and 16 false positives on the CMU test set shows that a sacrifice has been made in the accuracy. In (Hefenbrock et al., 2010), a multi-GPU implementation of the Viola-Jones algorithm is presented that runs at 15.2 FPS at 640×480 resolution on 4 Tesla GPUs. Another CUDA accelerated Viola-Jones algorithm given in (Obukhov, 2004) performs at 14 and 8 FPS on a GTX 480 for resolutions 1280×720 and 1920×1080, respectively. The scaling factors are limited to be integers, which means that the detector will not be able to detect small faces accurately. In (Harvey, 2009), the author achieved 2.8 FPS on a single GTX 280 GPU and 4.3 FPS on a dual GTX 295 GPU on VGA image sizes with another CUDA implementation of the Viola-Jones algorithm. (Herout et al., 2010) is the only GPU accelerated object detection algorithm to date that uses a different feature and different classifier structure than (Viola and Jones, 2004). The reported frame rate is 58 FPS for 1280×720 resolution on a GTX 280 GPU with no information about the accuracy.

In this paper, we follow a similar approach, but use a different algorithm. We present a GPU implementation of a boosting based object detection algorithm that uses illumination-robust MCT (Modified Census Transform) (Fröba and Ernst, 2004) based classifiers using CUDA. Our implementation per-

Cihan Tek S. and Gökmen M..
GPU ACCELERATED REAL-TIME OBJECT DETECTION ON HIGH RESOLUTION VIDEOS USING MODIFIED CENSUS TRANSFORM.
DOI: 10.5220/0003821606850688
In *Proceedings of the International Conference on Computer Vision Theory and Applications* (VISAPP-2012), pages 685-688
ISBN: 978-989-8565-03-7

forms all major steps of the algorithm on the GPU and hence requires minimum amount of memory transfers between the host and the GPU. We evaluate the performance of our implementation on face detection.

2 DETECTION ALGORITHM

MCT (Fröba and Ernst, 2004) is a transform that transforms the pixel values in a given neighborhood N to a binary string. The binary values corresponding to a pixel location is obtained by comparing the pixel values in the neighborhood with their mean. Let $N(\mathbf{x})$ be a spatial neighborhood centered at the pixel location $\mathbf{x}$ and $\bar{\mathbf{I}}(\mathbf{x})$ be the mean of the pixel intensity values on this neighborhood. If $\bigotimes$ is the concatenation operator, then the MCT can be defined as follows:

$$\Gamma(\mathbf{x}) = \bigotimes_{\mathbf{y}\in\mathcal{N}} \zeta(\bar{\mathbf{I}}(\mathbf{x}),\mathbf{I}(\mathbf{y})) \quad (1)$$

where the comparison function $\zeta(\mathbf{I}(\mathbf{x}),\mathbf{I}(\mathbf{y}))$ is defined as

$$\zeta(\mathbf{I}(\mathbf{x}),\mathbf{I}(\mathbf{y})) = \begin{cases} 1, & \mathbf{I}(\mathbf{x}) < \mathbf{I}(\mathbf{y}) \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

In order to obtain memory efficient classifiers, we use a neighborhood size of 3×3, which leads to 9 bit MCT values in the range of $[0,510]$. These values correspond to indices γ of local structure kernels that can encode structural information about the image like edges, ridges, etc. in binary form.

A MCT based weak classifier $h_{\mathbf{x}}$ consists of a coordinate pair $\mathbf{x} = (x,y)$ relative to the origin of the 24×24 fixed size scanning window and a 511 element lookup table. The lookup table contains a weight for each γ such that $0 \le \gamma \le 510$. The output a weak classifier gives on a window is determined by calculating γ from the neighborhood centered at $\mathbf{x}$ and getting the corresponding value from the lookup table.

The detector has a cascaded structure, where each stage contains a strong classifier containing a number of weak classifiers, each of which having a different $\mathbf{x}$ position. $H_j(\Gamma)$, the strong classifier of the j. stage, is defined as follows:

$$H_j(\Gamma) = \sum_{\mathbf{x}\in W'} h_{\mathbf{x}}(\Gamma(\mathbf{x})) \quad (3)$$

where $W' \subseteq W$ is the set of unique positions used by the weak classifiers $h_{\mathbf{x}}$ in the strong classifier. A window W passes the j. stage if the sum of the responses of the weak classifiers on that window is less than or equal to the stage threshold T_j. A window is classified as the searched object if it passes all stages. The cascade used in this work has 5 stages utilizing 10, 20, 40, 80, and 217 positions.

The detection process involves using the classifier cascade at each location on the input image at multiple scales. As in any other sliding window approach, there are scanning parameters like horizontal and vertical step sizes (Δx and Δy), the scale factor (s) and the starting scale. In this work, we choose $\Delta x = \Delta y = 1$, $s = 1.15$ and use a starting scale of 1, which results in a computationally demanding, fine-grained scanning process and makes it possible to detect faces as small as 24×24 accurately.

3 CUDA IMPLEMENTATION

3.1 Preprocessing

In order for the detector to detect objects at various sizes, the input image needs to be scanned in multiple scales. In (Viola and Jones, 2004), this is performed by scanning the same input image with up-scaled classifiers. The approach we followed is to construct an image pyramid and scan it using a fixed size window to prevent sparse global memory accesses that would have been required by the former approach at larger scales. The whole image pyramid is stored as a single image as shown on the left in Figure 1. Even though this layout has empty regions that increase the number of windows scanned, it greatly simplifies the scanning process described in the next section. The regions having a constant gray level are easily eliminated by the first stage of the cascade.

The image at each level of the pyramid is constructed by binding the image from the previous level to a texture to take advantage of the hardware bilinear interpolation capability of the GPU, and then downsampling it according to the scale factor.

Evaluation of a weak classifier becomes a simple memory lookup when the MCT values are precomputed. To take advantage of this property, the whole image pyramid is transformed with MCT beforehand by launching a kernel with a block size of 16×16. Each thread block pulls in a patch from the pyramid in global memory to its dedicated shared memory before further processing. Each thread in a block pulls in a single pixel from the global memory. Since the MCT computation at the edges and corners requires additional pixels, threads at the edges and corners pull in additional pixels. After getting the data to shared memory, threads in each block are synchronized. Then each thread computes the value corresponding to its location and writes the result to the global memory. At this point, the original image pyramid is no longer needed and hence can be discarded. In the remaining sections, the term ”image pyramid”

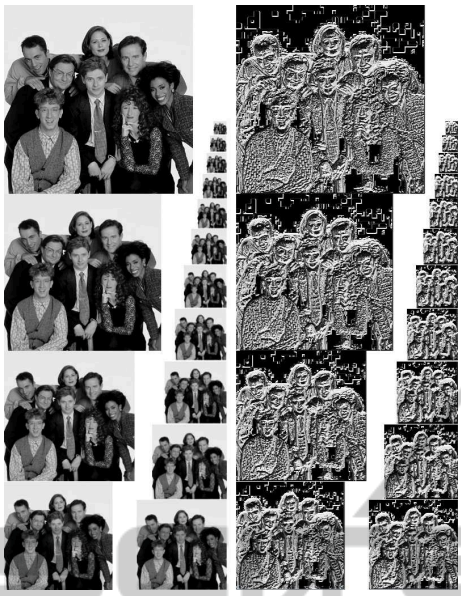

Figure 1: Image pyramid and the result of transforming it with MCT.

refers to the one containing the result of the MCT, which is shown on the right side of Figure 1.

During the scanning process, the cascade data will be heavily accessed and therefore the speed of accessing them is crucial for the performance. Since the size of the constant memory is too small store a cascade containing more than 32 weak classifiers, it is stored in the texture memory as a 2D floating point texture. Each row starts with a single floating point value containing the x and y coordinate of the weak classifier in its upper and lower 2 bytes, respectively. Rest of the row contains the weights of the lookup table. This approach helps to reduce texture cache misses by making the accesses to the feature weights as spatially local as possible in the 2D space.

3.2 Object Detection

The detection process involves scanning the image pyramid with a fixed size sliding window, classifying the window each location and writing results back to the device memory. We found that the best performance overall is obtained when making each thread classify a single window on the image pyramid. As a natural consequence of using a cascaded classifier structure, the number of stages that will be evaluated in a window cannot be predetermined. Threads that classify their windows as negative in early stages have to wait idle until all other threads in the same block finish their tasks, only after which the processing of a new block can be started. This results in under-utilization of GPU resources. In order to overcome this problem, we split the cascade into 2 smaller groups and use 2 different detection kernels for scanning. Each kernel uses one of the groups obtained in the previous step for classification. After performing a number of experiments, we found that the best location for the split is after the 2nd stage.

The first detection kernel is launched with a grid having as many threads as the number of windows that needs to be scanned and a block size of 16×16. When a window is classified as the searched object, its coordinates are written to the corresponding location in a preallocated 1D array of floats that resides in global memory. Each element of this array can store the x and y coordinates of a window in its upper and lower two bytes, respectively. Since detections are rare, this array contains sparse data after the execution of the kernel finishes. Most of its elements still contain dummy values that are set when it is first allocated. To solve this problem, we use a stream compaction algorithm that copies all elements having meaningful values to the beginning of the array. Then we launch another detection kernel with a 1D grid of thread blocks, each one containing a 1D array of 256 threads. Each thread classifies a single window whose coordinates are fetched from the array generated by the first kernel. The whole process is shown in Figure 2 for the first 10 windows in the input image.

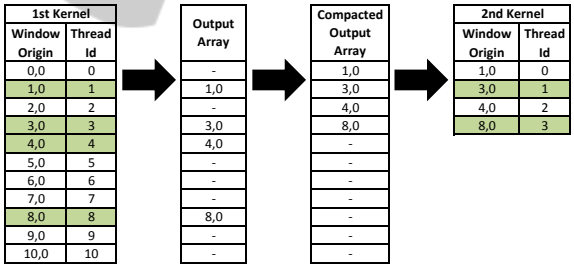

Figure 2: Steps of scanning with 2 kernels. Window locations that are detected as positive are highlighted.

3.3 Utilizing Multiple GPUs

We have extended our implementation to reduce the detection times even further on devices that have multiple GPUs. We spawn $M+1$ CPU threads, where M is the number of GPUs in the system. The main thread acquires frames from the video stream and does preliminary computations. Each one of the other threads performs kernel launches and memory transfers between the GPU assigned to it and the host. Each GPU generates several levels of the image pyramid and scans only those levels. The input image and the cascade data are copied to the memories of each GPU separately. Distributing different levels of the pyramid to different GPUs makes it possible to achieve

nearly linear speed-up when the levels each GPU will process is carefully determined.

4 EXPERIMENTAL RESULTS

We have tested both the CPU and GPU implementations on the same desktop PC that contains a Intel Core i5-2500k processor, 3 GTX 580 GPUs and 8GB RAM. Figure 3 shows the average number of frames processed per second by all implementations on video streams of various sizes. These measurements include the time required to perform preprocessing and memory copies between the host and the device. The multi-threaded CPU implementation uses OpenMP to distribute the processing to different cores.

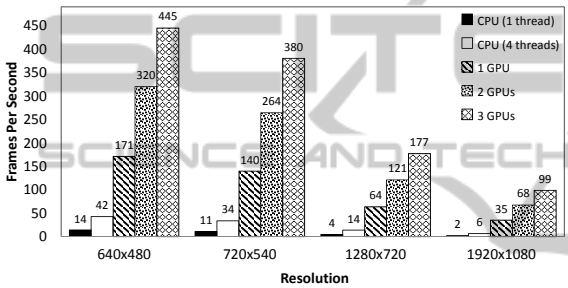

Figure 3: Frame rates of GPU and CPU implementations on various input resolutions.

As it can be seen from Figure 3, even the single-GPU implementation outperforms the single-threaded and multi-threaded CPU implementations by a factor of 12-18x and 4-6x, respectively. As the resolution increases, so does the difference between the speed of the GPU and CPU implementations, clearly showing that a GPU, as a massively parallel processor, is better suited to process high resolution videos than a CPU. These results also show that, in contrast to the CPU, the performance of the GPU based implementation scales nearly linearly with the number of GPUs.

We have also tested both our CPU and GPU implementations on the CMU+MIT frontal face test set (Rowley et al., 1998) to validate our results. The detection rates for both implementations are measured as %90.8 with 32 false positives, proving that our GPU implementation generates the exact same results with those of the CPU implementation.

5 CONCLUSIONS

We have presented an efficient GPU implementation of a boosting based, real-time object detection system utilizing MCT based classifiers using CUDA. We have shown that even our single GPU implementation outperforms both the single-threaded and multi-threaded CPU implementations running on a high-end CPU. We have pointed out that, because of their massively parallel architecture, GPUs are more suitable for working with high resolution videos than CPUs. Our implementation, with its ability to detect objects in high resolution video streams in real-time, can easily be used in modern multimedia, entertainment and surveillance systems.

ACKNOWLEDGEMENTS

This work is supported by ITU-BAP and TUBITAK under the grants 34120 and 109E268, respectively.

REFERENCES

Fröba, B. and Ernst, A. (2004). Face detection with the modified census transform. In *Proceedings of the Sixth IEEE international conference on Automatic face and gesture recognition*, FGR' 04, pages 91–96, Washington, DC, USA. IEEE Computer Society.

Harvey, J. P. (2009). Gpu acceleration of object classification algorithms using nvidia cuda. Master's thesis, Rochester Institute of Technology.

Hefenbrock, D., Oberg, J., Thanh, N. T. N., Kastner, R., and Baden, S. B. (2010). Accelerating viola-jones face detection to fpga-level using gpus. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 11–18, Washington, DC, USA. IEEE Computer Society.

Herout, A., Joth, R., Jurnek, R., Havel, J., Hradi, M., and Zemk, P. (2010). Real-time object detection on cuda. *Journal of Real-Time Image Processing*, 2010(1):1–12.

Obukhov, A. (2004). Haar classifiers for object detection with cuda. In Fernando, R., editor, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 33, pages 517–544. Addison Wesley.

Rowley, H., Baluja, S., and Kanade, T. (1998). Neural network-based face detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(1):23–38.

Sharma, B., Thota, R., Vydyanathan, N., and Kale, A. (2009). Towards a robust, real-time face processing system using cuda-enabled gpus. In *High Performance Computing (HiPC), 2009 International Conference on*, page 368377. IEEE.

Viola, P. and Jones, M. J. (2004). Robust real-time face detection. *Int. J. Comput. Vision*, 57:137–154.