GPU implementation of Extended Gaussian mixture model for Background subtraction

Vu Pham, Phong Vo, Vu Thanh Hung, Le Hoai Bac
Department of Computer Science
University of Science
Ho Chi Minh City, Viet Nam
{phvu, vdphong, vthung, lhbac}@fit.hcmus.edu.vn

***Abstract*—Although trivial background subtraction (BGS) algorithms (e.g. frame differencing, running average...) can perform quite fast, they are not robust enough to be used in various computer vision problems. Some complex algorithms usually give better results, but are too slow to be applied to real-time systems. We propose an improved version of the Extended Gaussian mixture model that utilizes the computational power of Graphics Processing Units (GPUs) to achieve real-time performance. Experiments show that our implementation running on a low-end GeForce 9600GT GPU provides at least 10x speedup. The frame rate is greater than 50 frames per second (fps) for most of the tests, even on HD video formats.**

***Index Terms*—Gaussian mixture model, Background subtraction, GPU**

I. Introduction

Background subtraction (BGS) is a process to segment out the foreground objects from the background of a video. In order to take account of many challenges in real-life scenarios such as illumination change, camera moving... numerous BGS algorithms have been developed. Recently, there are some surveys and comparative studies examined a wide-range of BGS methods [1], [2], [3], [4]. In those studies, it can be seen that none of the common BGS methods consistently outperform any other, the choice of BGS algorithms depends on the specific context that we are working on.

For most of common pixel-level BGS methods, the scene model maintains a probability density function for each pixel individually. A pixel in a new image is regarded as background if its new value is well described by its density function. Since the intensity value of pixels can change frame by frame, one might want to estimate appropriate values for mean and variance of underlying distribution of pixel intensities. This single Gaussian model was used by Wren et al. [5]. However, pixel value distributions are often more complex, hence more sophisticated models are necessary to get around under-fitting issue. In 1997, Friedman and Russell proposed Gaussian mixture model (GMM) approach for BGS [6]. Later, efficient update equations are given in [7] by Stauffer and Grimson, and this is often considered as the exemplary version of GMM for BGS. All these GMMs use a fixed number of components. Stenger et al. [8] suggested a method to select the number of components of a hidden Markov model in an off-line training procedure. The GMM update equations are continuously improved to simultaneously select the appropriate number of components for each pixel on-the-fly, which not only makes the model adapt fully to the scene, but also reduces the processing time and improves the segmentation [9], [10]. We consider this Extended GMM to be the state of the art in parametric background subtraction techniques. Nonetheless, we observed that the original implementation of Zivkovic [10] is still slow in demanding applications, especially for high-resolution video sequences.

The contemporary Graphics processing units (GPUs) utilize the so-called 'single instruction multiple data' (SIMD) architecture to perform massively parallel operations. In order to obtain maximum speedup on GPU, care must be taken to coalesce memory requests and to minimize the branching factor. The Compute Unified Device Architecture (CUDA) [11] and the Close-To-Metal [12] are some easy-to-use interfaces for modern GPUs. In this report, we present a fast processing version of Extended GMM using CUDA. We use global memory and constant memory on GPU for effective computation. We optimize memory access pattern to maximize memory throughput, which helps to improve the overall performance dramatically. CUDA streams are also utilized to execute kernel function asynchronously.

The first GPU implementation of Stauffer's and Grimson's algorithm was proposed by Lee and Jeong [13] in 2006 using Nvidia's Cg. Later in 2008, Peter Carr [14] reported an implementation of Stauffer's and Grimson's algorithm using Apple's Core Image Kernel Library. He achieved a speedup of 5x with 58 frames per second (fps) on the 352x288 video sequence. However, those works focused on improving the original Stauffer's and Grimson's method, while we investigate on Zivkovic's Extended GMM. On the 400x300 benchmark video, our implementation executes in more than 980 fps, 13 times faster than the original implementation of Zivkovic.

The paper is organized as follows. Section II discusses the implementation on CUDA and various optimization techniques

Vu Pham, Phong Vo, Hung Vu Thanh, Bac Le Hoai, "*GPU Implementation of Extended Gaussian Mixture Model for Background Subtraction*", IEEE-RIVF 2010 International Conference on Computing and Telecommunication Technologies, Vietnam National University, Nov. 01-04, 2010. DOI: 10.1109/RIVF.2010.5634007.

which we have used in order to reach the maximum speedup. In Section III, we present the experimental results and analyze them. Some concluding remarks are given in Section IV.

II. Extended GMM on CUDA

A. Basic implementation

In both CPU and GPU implementations, each pixel is processed independently from each other. This makes Extended GMM an embarrassingly parallel algorithm, which can be parallelized by devoting a thread for each pixel. The number of pixels for each thread is automatically determined by the algorithm right before the first frame is processed. When a new frame arrives, the algorithm is run and the background model is updated. Pixels in output image are classified as either foreground or background, depends on their intensity value.

High-level data flow of the CUDA implementation is given in Figure 1. The input, output frames and the background model are stored in global memory, while the configuration parameters are stored in constant memory to ensure fast access from threads. Although shared memory is much faster than global memory, its capacity is not enough to store input and output data for all threads in the block. In the implementation, we use only 4 bytes of dynamic shared memory to store common parameters for the block.

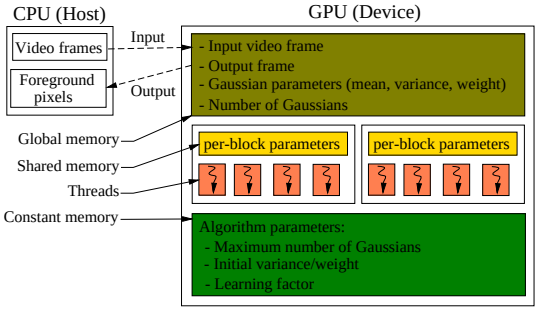


Figure 1: Data flow of the CUDA implementation.

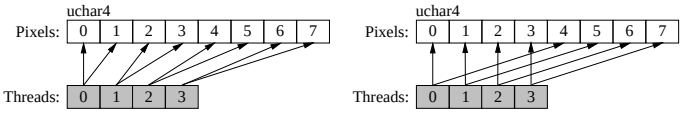
This basic implementation only provides 3x speedup over the original CPU implementation for VGA (640x480) video. Although the speedup is not substantial, the implementation is still helpful since the CPU is offloaded of the processing of video frames.

B. Different levels of optimization

Since the low memory bandwidth is one of the most crucial bottlenecks in GPU computing, our optimization mainly focused on improving the overall bandwidth usage of the algorithm. This is achieved by applying various techniques to the basic implementation (in turn of discussion here), which can help to improve performance significantly. The efficiency of these techniques is examined in Section III.

1) Pinned (non-pageable) memory: Using pinned memory not only makes the memory transfers can be performed concurrently with kernel execution (discussed in Section II-B3), but also help increase the bandwidth between host memory and device memory [15]. In our implementation, pinned memory is used to transfer the input and output image data between host and device. This not only improves memory throughput, but also allows launching the kernel asynchronously.

2) Memory coalescing: In CUDA, when threads in a half-warp (16 continuous threads in the same block) access an aligned contiguous region in global memory, all of the individual transfers are automatically combined into a single transfer [15]. In order to take advantage of CUDA memory coalescing, we convert input images into four channels (RGBA) mode, which uses 4 bytes for each pixel. By this way, each thread can access its pixels using `uchar4` structures. We also distribute pixels to threads so that all threads in the same block always access the neighbour pixels in each processing step. In Figure 2, a simple case of four threads accessing eight words (2 words for each thread) is depicted. In our implementation, a block has 128 threads, and threads access to pixels in step of 128.

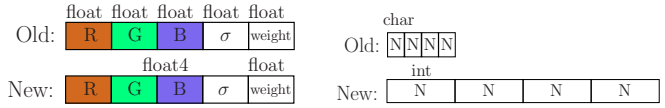


(a) Uncoalesced memory access pattern. (b) Coalesced memory access pattern.

Figure 2: Improvements when accessing to the input image. As in (b), 4 neighbour threads always access to 4 contiguous words, which enables memory coalescing.

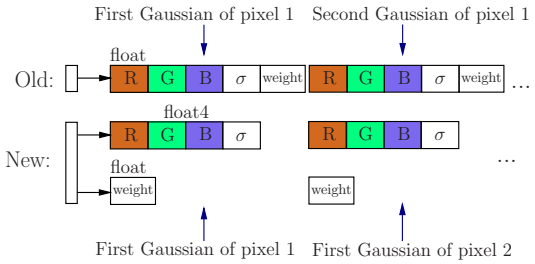
In addition, we also reorganize the Gaussian component parameters. Firstly, we compact the Gaussian parameters as depicted in Figure 3a. We can further compact the Gaussian mean values into an `uchar4` structure, but we discover that the algorithm's accuracy decreases, as the Gaussian mean values are not precise enough. Secondly, the array of the number of Gaussian components per pixel is also inflated to 4 bytes for each element, allows memory coalescing with 4 bytes for each thread (Figure 3b). Thirdly, we follow the Structure of Arrays (SoA) pattern to store the background model, instead of the easy to use Array of Structures (AoS) pattern. As described in Figure 3c, the array of Gaussian means and variances is stored separately from the weights. Since they have different access patterns, separating these two arrays helps to avoid unnecessary data transfers. We also rearrange the Gaussian components so that the first components of all the pixels come before the second components. This is expected to increase the chance of memory coalescing when neighbor threads access to contiguous Gaussian components. Lastly, as in the input image array, the array of Gaussian parameters and the array of "the number of Gaussian components per pixel" are accessed in step of 128 from threads.

3) Asynchronous execution: A CUDA *stream* is a sequence of commands that execute in order on the device. Using streams



(a) Compaction of Gaussian mean values into a float4 structure.

(b) Inflation of the "number of Gaussian components per pixel" array.



(c) Reorganizing the background model using the SoA pattern.

Figure 3: Improvements on storing the Gaussian parameters arrays.

and pinned memory, we are able to interleave the CPU code execution with the kernel launches and memory transfers as described in Figure 4. For an effective implementation, we use a second buffer memory, and define two CUDA streams, *execStream* and *copyStream*. While the kernel is busy with processing image data in the first buffer, the copy stream keeps transferring the next frame into the second buffer. When the kernel finishes its processing, the buffer pointers are swapped and the kernel is launched again. This double buffering technique allows to interleave the CPU code with the two CUDA stream executions.

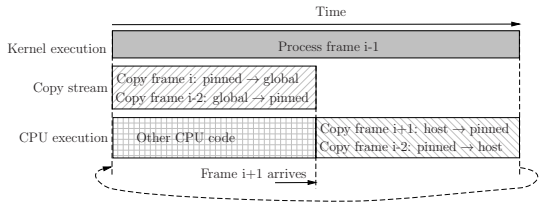


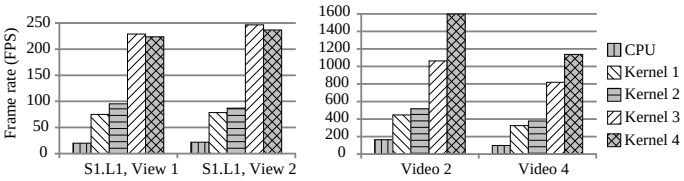
Figure 4: Timeline for asynchronous execution.

III. Experimental results

The CPU and GPU implementations are executed on the dataset from the VSSN 2006 competition [16] to compare the robustness, and on the PETS2009 [17] dataset to evaluate the speedup. The VSSN2006 dataset contains various 384x240 semi-synthetic video sequences and the corresponding ground truth. We chose video 2, video 4 and video 8 for evaluation. PETS2009 is a set of 768x576 image sequences in outdoor condition. We choose view 1 and view 2 from S1.L1 scenario, view 1 from S1.L2 scenario to test the performance of the implementations. All experiments are performed on a system running Core 2 Quad Q9400 2.6 GHz CPU (4 GB of RAM) and GeForce 9600GT GPU. Although the CPU implementation is not multithreaded, we still execute it on four cores to get the best performance.

A. Optimization levels and Occupancy

To estimate the effectiveness of various optimization techniques, we run the implementation with different levels of optimization on the VSSN2006 and PETS2009 datasets. The results are shown in Figure 5. Kernel 1 is the GPU basic implementation using constant memory as described in Section II-A. Kernel 2 is more optimized with memory coalescing on 4-channel input images. Kernel 3 uses the SoA pattern, memory coalescing on Gaussian parameters, and pinned memory when transferring data between host and device. Kernel 4 is the asynchronous implementation as in Section II-B3.



(a) PETS2009. (b) VSSN06.

Figure 5: Effectiveness of optimization techniques.

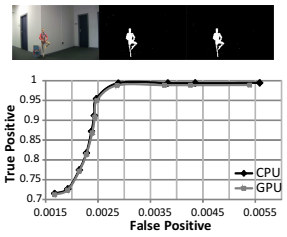
On VSSN 2006 dataset, kernel 4 outperforms all others, but on PETS2009, it is slightly slower than kernel 3. This can be roughly explained by the difference in frame size of these two datasets: PETS2009 has 768x576 sequences, while VSSN2006 contains smaller 384x240 sequences. The asynchronous implementation in kernel 4 requires more time to transfer large data between host and device, leads to the decrement in frame rate. Nevertheless, we prefer kernel 4 since it is better in interleaving the CPU processing into the GPU processing time. We use kernel 4 for all remaining experiments.

In CUDA, the multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU [11]. Our implementation uses 128 threads per block, 20 registers per thread and 36 bytes of shared memory per block. The implementation is executed on GeForce 9600GT which supports compute capability 1.1. Thus, the kernel has occupancy of 0.5. This configuration is determined experimentally. When attempting to increase the occupancy by using less registers per thread and more/less threads per block, we notice that the overall performance falls.

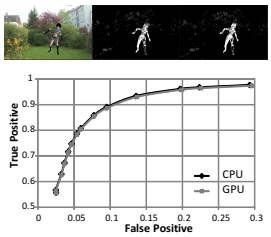
B. Robustness

The VSSN 2006 dataset is used to analyze the precision of the two implementations. At the beginning of those sequences, there are several hundreds of training frames which contain the background only. Since the extended GMM is online, these frames were not used for evaluation. Therefore, the first 150 frames in video 2, 305 frames in video 4 and 655 frames in

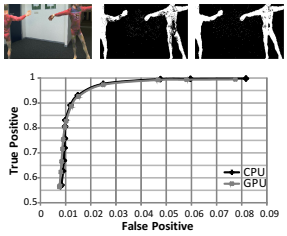
video 8 are ignored. In the tests, the leaning factor (see [10], [9]) is varied in the range from 0.0005 to 0.1, we measured the true positives - percentage of the foreground pixels that are correctly classified, and the false positives - percentage of the background pixels that are incorrectly classified as the foreground. These results are plotted as the receiver operating characteristic (ROC) curves and are given in Figure 6.



(a) VSSN06, Video 2.



(b) VSSN06, Video 4.



(c) VSSN06, Video 8.

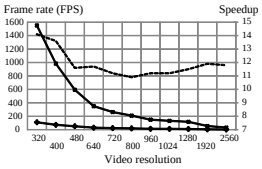
Figure 6: Comparision of the GPU implementation to the CPU implementation. The ROC curves show that the GPU implemetation has nearly the same robustness with the original implementation. The top lines are sample input frame, CPU result and GPU result respectively.

C. Frame rate and speedup

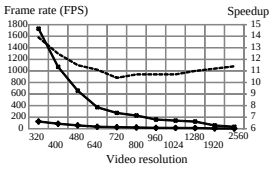
The PETS2009 is used to test the speedup of the GPU implementation over various frame resolution. We down-sample and up-sample frames to get 11 sequences in different resolutions. At each resolution level, we run both implementations, measure the frame rate and speedup. The results are provided in Figure 7. The CPU implementation performs pretty well on the low-resolution sequences, but for sequences larger than 640x480, its frame rates are lower than 30 fps, and for HD sequences (1280x720, 1920x1080, 2560x1440) the frame rates are below 10 fps. For the GPU version, even with HD sequences, the frame rate is always greater than 30 fps.

IV. Conclusions

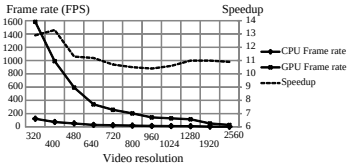
Since background subtraction is a basic task in many computer vision applications, it is preferred to be as fast as possible. We have observed that the SIMD architecture can help to improve the performance of this algorithm significantly. Besides the speedup, GPU implementation also helps to offload CPU from the heavy burden of processing each pixel consecutively.



(a) S1.L1, View 1.



(b) S1.L1, View 2.



(c) S1.L2, View 1.

Figure 7: Comparision of the CPU version and GPU version in term of frame rate over different resolutions. Dash lines are the speedup. The average speedup is about 11x.

In addition, we have also checked the efficiency of various CUDA optimization techniques. We are able to achieve high speedup when apply various optimization techniques such as memory coalescing and asynchronous execution[1].

References

[1] M. Piccardi, "Background subtraction techniques: A Review," in *IEEE International Conference on Systems, Man and Cybernetics*, 2004, pp. 3099–3104.

[2] Y. Benezeth, P. Jodoin, B. Emile, H. Laurent, and C. Rosenberger, "Review and evaluation of commonly-implemented background subtraction algorithms," in *IEEE International Conference on Pattern Recognition*, December 2008, pp. 1–4.

[3] R. J. Radke, S. Andra, O. Al-Kofahi, and B. Roysam, "Image change detection algorithms: a systematic survey," *IEEE Transactions on Image Processing*, vol. 14, no. 3, pp. 294–307, March 2005.

[4] D. H. Parks and S. S. Fels, "Evaluation of Background Subtraction Algorithms with Post-Processing," in *IEEE Fifth International Conference on Advanced Video and Signal Based Surveillance*, September 2008, pp. 192–199.

[5] C. Wren, A. Azarbayejani, T. Darrell, and A. Pentland, "Pfinder: real-time tracking of the human body," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 7, pp. 780–785, July 1997.

[6] N. Friedman and S. Russell, "Image segmentation in video sequences: A probabilistic approach," in *Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 1997, pp. 175–181.

[7] C. Stauffer and W. E. L. Grimson, "Adaptive Background Mixture Models for Real-Time Tracking," in *IEEE Conference on Computer Vision and Pattern Recognition*, 1999, pp. 246–252.

[8] B. Stenger, V. Ramesh, N. Paragios, F. Coetzee, and J. Buhmann, "Topology free hidden Markov models: application to background modeling," in *Proc. IEEE International Conference on Computer Vision*, 2001, pp. 294–301.

[9] Z. Zivkovic, "Improved adaptive gaussian mixture model for background subtraction," in *Proc. IEEE International Conference on Pattern Recognition*, no. 2, 2004, pp. 28–31.

[10] Z. Zivkovic and F. van der Heijden, "Efficient adaptive density estimation per image pixel for the task of background subtraction," *Pattern Recognition Letters*, vol. 27, no. 7, pp. 773–780, May 2006.

[11] NVIDIA Corporation, *CUDA Programming Guide*. NVIDIA, 2007.

[12] AMD/ATI, *ATI CTM (Close to Mental) Guide*. AMD/ATI, 2007.

[1]To facilitate the applying our implementation, we have published the source code at http://www.fit.hcmus.edu.vn/~vdphong/.

[13] S.-j. Lee and C.-s. Jeong, "Real-time Object Segmentation based on GPU," in *Proc. International Conference on Computational Intelligence and Security*, November 2006, pp. 739–742.
[14] P. Carr, "GPU Accelerated Multimodal Background Subtraction," in *Proc. Digital Image Computing: Techniques and Applications*, December 2008, pp. 279–286.
[15] NVIDIA Corporation, *CUDA Best Practices Guide*. NVIDIA, 2010.
[16] "VSSN 2006 Competition," 2006. [Online]. Available: http://mmc36.informatik.uni-augsburg.de/VSSN06_OSAC/
[17] "PETS 2009 Benchmark Data," 2009. [Online]. Available: http://www.cvg.rdg.ac.uk/PETS2009/a.html