GPU implementation of Extended Gaussian mixture model for Background subtraction

Vu Pham, Vo Dinh Phong, 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

In many computer vision applications such as human-computer interaction, traffic monitoring, video surveillance... background subtraction (BGS) is widely used as a preprocessing step. This is a process to separate or segment out the foreground objects from the background of a video. The foreground objects are defined as the parts of the video frame that changes and the background is made out of the pixels that stay relatively constant. In real-life scenarios, however, there are many factors can affect the video: illumination change, moving objects in the background, camera moving... In order to take account of those issues, numerous BGS algorithms have been developed. These algorithms are often divided into two classes: *parametric* models and *non-parametric* models. Parametric models maintain a single background model that is updated with each new video frame, while non-parametric models store a buffer of n previous video frames and estimate a background model based solely on statistical properties of these frames. Approximated median filtering (AMF) [1], Running Gaussian average [2], Gaussian mixture model (GMM) [3]... are some of common parametric models. Some of popular non-parametric models are Eigenbackgrounds [4], Kernel density estimator (KDE) [5], Median filtering [6]... Recently, there are some surveys and comparative studies examined a wide-range of BGS methods [7], [8], [9], [10]. Some of them [9] focused only on computational complexity, memory requirement and theoretical accuracy, but some others [7], [10], [8] also investigated the practical performance and post-processing techniques. 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. [2]. 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 GMM approach for BGS [11]. Later, efficient update equations are given in [3] by Stauffer and Grimson, and this is often considered as the exemplary version of GMM for BGS. In [12], the GMM is extended with a hysteresis threshold. The standard GMM update equations are extended in [13] to improve the speed of alteration of the model. All these GMMs use a fixed number of components. Stenger et al. [14] 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 [15], [16]. 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 [16] is still slow in demanding applications, especially for high-resolution video sequences.

The contemporary Graphics processing units (GPUs) are optimized for the specific task of rendering images, and can be very efficient on many data-parallel tasks. However, as opposed to CPUs, which provide more flexibility in program control, GPUs utilize the so-called 'single instruction multiple data' (SIMD) architecture to perform massively parallel operations. Programs which run on GPU must follow the traditional graphics pipeline. This makes difficulties in implementing general algorithms on GPU. In order to obtain maximum speedup on GPU, one cannot write a multithreaded program as for CPU. Care must be taken to coalesce memory requests and to minimize the branching factor, as threads on the same

multiprocessor operate concurrently. The Compute Unified Device Architecture (CUDA) from Nvidia [17] and the Close-To-Metal (CTM) from ATI/AMD [18] are such interfaces for modern GPUs. Lately, many modern applications are increasingly turning to GPUs for specialized computations [19].

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. These optimization techniques are commonly used in CUDA developer community and they have very good effect on performance of CUDA algorithms.

The first GPU implementation of Stauffer's and Grimson's algorithm was proposed by Lee and Jeong in 2006 using Nvidia's Cg library [20]. Later in 2008, Peter Carr [21] reported an implementation of Stauffer's and Grimson's algorithm using Apple's Core Image Kernel Library (CIKL) [22]. 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. In the next section, we review Zivkovic's Extended GMM algorithm. Section III discusses the implementation on CUDA and various optimization techniques which we have used in order to reach the maximum speedup. In Section IV, we present the experimental results and analyze them. Some concluding remarks and directions for future work are given in Section V.

II. Extended Gaussian Mixture Model

The Gaussian Mixture Model stores M separated normal distributions for each pixel (parameterized by mean μ_i, variance σ_i^2 and mixing weight w_i, where $i = 1, 2, ..., M$) with M typically between 3 and 5 (depending on the complexity of the scene). This leads to the following probability distribution for pixel value I_t

$$P(I_t) = \sum_{i=1}^{M} w_i \mathcal{N}(I_t; \mu_i, \sigma_i^2 I) \tag{1}$$

where the estimated mixing weights w_i are non-negative and add up to one (see Figure 1).

In Zivkovic's method, incoming data is weighted by a learning factor α and assigned to the "close" component with the largest mixing weight w_i by a binary ownership label o_i which equals 1 for pixels belonging to the i-th "close" component, and 0 otherwise. The updating equations are

$$w_i \leftarrow w_i + \alpha\,(o_i - w_i) \tag{2}$$

$$\mu_i \leftarrow \mu_i + o_i \frac{\alpha}{w_i}\,(I_t - \mu_i) \tag{3}$$

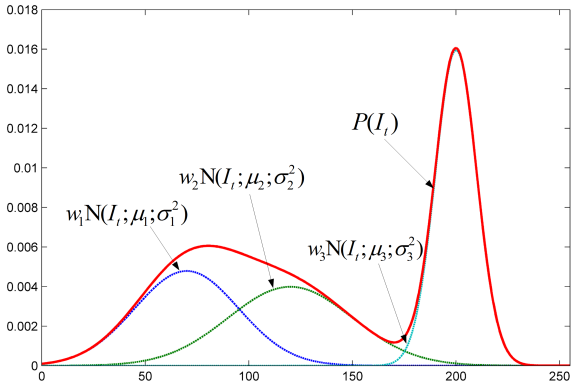


Figure 1: The pixel value probability $P(I_t)$ in equation (1) is illustrated for 1D pixel value $I_t \in \{0, 1, .., 255\}$, $M = 3$, $w_i = \{0.3, 0.3, 0.4\}$, $\mu_i = \{70, 120, 200\}$, $\sigma_i^2 = \{25, 30, 10\}$.

$$\sigma_i^2 \leftarrow \sigma_i^2 + o_i \frac{\alpha}{w_i}\left((I_t - \mu_i)^2 - \sigma_i^2\right) \tag{4}$$

As defined in [16], a sample is "close" to a component if the Mahalanobis distance from the component falls within, for example, three standard deviations. If there are no "close" components, a new Gaussian component is created with $w_{M+1} = \alpha, \mu_{M+1} = I_t$ and $\sigma_{M+1} = \sigma_0$ where σ_0 is a large initial variance. If the maximum number of components is exceeded, the component with lowest weight will be discarded.

In this method, a foreign object appearing in the scene will be represented by some additional components with low weights and high variances. This leads to the conclusion that background-foreground segmentation can be achieved by selecting the mixture of the N components of the highest weight to variance ratios as background model, and the remainder as foreground. Because of this acceptance criterion, the GMM contains the intrinsic assumption that background pixels have low variance.

Zivkovic's and Heijden's Extended GMM improves on this by using normalized mixture weights w_i to define an underlying multinomial distribution describing the probability that a sample pixel belongs to the *i*-th component of the GMM. This way w_i is determined by the product of the number of samples assigned to component i and the learning factor α:

$$w_i = \alpha n_i = \alpha \sum_{j=1}^{T} o_i^{(j)} \tag{5}$$

This estimate is further improved upon by introducing prior knowledge in the form of the conjugate prior of the multinomial distribution. This is expressed in the update equation (2) as a negative weight $-c$ imposing a minimal amount of evidence required from the data before a component can be allowed to exist. A new weight update is defined

$$w_i \leftarrow w_i + \alpha\,(o_i - w_i - c_T) \tag{6}$$

with $c_T \approx \alpha c$. After each iteration, the weights must be normalized to ensure a proper probability distribution.

This method can adapt to changing circumstances and a wide range of irregular foreground objects being introduced in the scene, while retaining control over the number of Gaussians in the mixtures.

III. Extended GMM on CUDA

The CUDA environment exposes the SIMD architecture of the GPUs by enabling the operation of program *kernels* on data *grids*, divided into multiple *blocks* consisting of several *threads*. The highest performance is achieved when the threads avoid divergence and perform the same operation on their data elements. The GPU has high computational power but low memory bandwidth. CUDA threads can access data from several memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block. All threads have access to the same global memory. There are two additional read-only memory spaces: constant and texture memory spaces. Accessing to constant memory and shared memory is *generally* much faster than global memory. However, shared memory is limited in capacity and it is only fast as long as there are no bank conflicts between the threads. This basic knowledge is significant in developing and considering an implementation on CUDA. The details of CUDA memory hierarchy and CUDA heterogeneous programming model can be found in [17].

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. However, the number of threads on GPU is limited, while the number of pixels per frame is various for each video. Our implementation deals with this by automatically choosing an appropriate number of pixels for each thread. 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. This process is depicted in Figure 2.

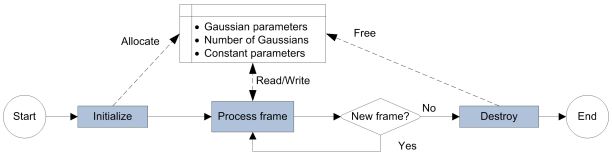


Figure 2: General execution flow chart of the CPU/GPU implementation.

High-level data flow of the CUDA implementation is given in Figure 3. The background model (Gaussian parameters, the number of Gaussians array) is stored in global memory, while the system configuration parameters are stored in constant memory to ensure fast access from threads. Input and output frames are also allocated in global memory. A new frame will be processed as follows: (1) the input frame is copied from host memory to global memory on GPU, (2) the kernel function is executed on threads, (3) each thread process its pixel(s) using the parameters in constant memory, (4) each thread copies its processed pixel(s) to global memory, and (5) the output image is copied from global memory to host memory. Although shared memory is much faster than global memory, its capacity (16 KB for most of modern NVIDIA GPUs) 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.

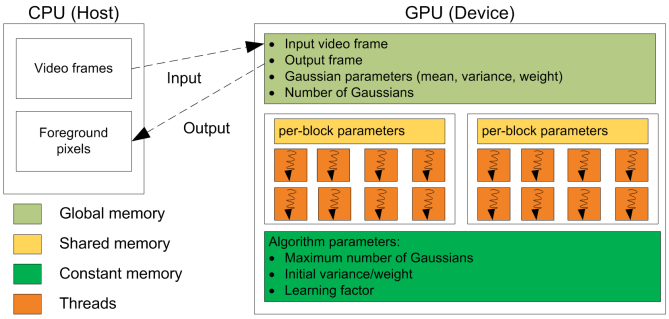


Figure 3: Data flow of the CUDA implementation.

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 with CUDA, our optimization mainly focused on improving the overall bandwidth usage of the algorithm. This is achieved by applying various techniques which are discussed in this section. These techniques, when applying on the basic implementation (in turn of discussion here), help to improve performance significantly. The efficiency of these techniques is examined in Section IV.

1) Pinned (non-pageable) memory: Pinned memory is located on host storage and is not subject to page in the virtual memory storage system. The CUDA runtime provides some functions to allocate and free pinned host memory. Using pinned memory has several benefits:

- The memory transfers can be performed concurrently with kernel execution (discussed in Section III-B3).
- Transfer bandwidth between host memory and device memory is higher when using pinned memory. To clarify this, we make a quick test of memory transfer on a system running Core 2 Quad Q9400 2.6 GHz CPU, 4 MB of RAM and NVIDIA GeForce 9600GT GPU. As reported in Figure 4, pinned memory outperforms pageable memory when copied from CPU to GPU; but when copied from GPU to CPU, pinned memory is slightly slower.

In our implementation, pinned memory is used to transfer the input and output image data between host and device.

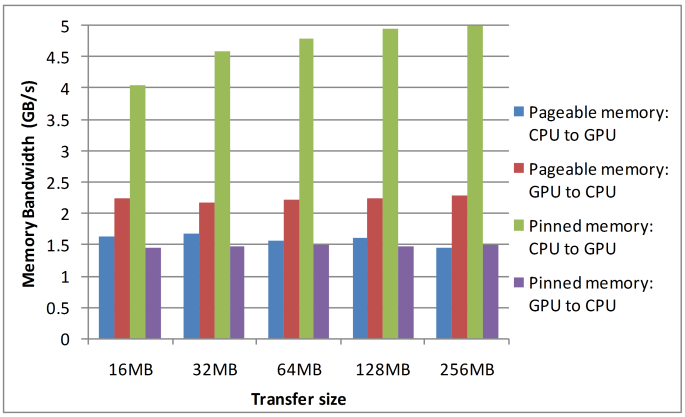


Figure 4: Differences in memory bandwidth when using pinned memory.

This not only improves memory throughput, but also allows launching the kernel asynchronously.

2) Memory coalescing: In CUDA, global memory accesses which are performed by neighbored threads can be coalesced automatically into as few as one transaction when certain requirements are met. These requirements are not the same for all of CUDA devices. Generally, 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 combined into a single transfer (not all threads need to participate) [23]. A simple coalesced access pattern is given in Figure 5.

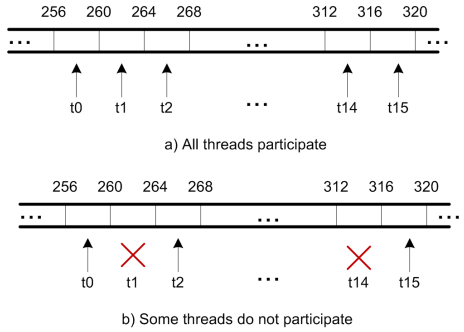


Figure 5: A simple coalesced global memory access pattern. Each thread reads a 4-byte word, the contiguous region has 64 bytes in length and it is aligned to 4 bytes. In a) 16 threads participate in the transfer, but in b) there are some threads do not. In both cases, the device will automalically coalesces the transfers into one transaction, increasing the bandwidth significantly.

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 (an `uchar4` is a quaternion of unsigned bytes). 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 6, 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.

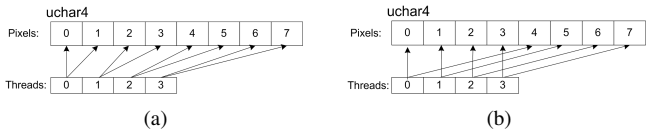
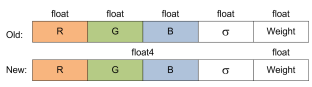


Figure 6: Improvements when accessing to the input image. (a) Uncoalesed memory access pattern where each thread accesses 2 adjacent words. (b) Coalesced memory access pattern where each thread accesses 2 words in step of 4. As in (b), 4 neighbour threads always access to 4 contiguous words, which enables memory coalescing.

In addition, we also reorganize the Gaussian component parameters. Firstly, we compact the Gaussian parameters as depicted in Figure 7a. 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 7b). 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 in the original CPU implementation. As described in Figure 7c, the array of Gaussian means and variances is stored separately from the weights. Since they have different access patterns (discussed in Section II), 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.

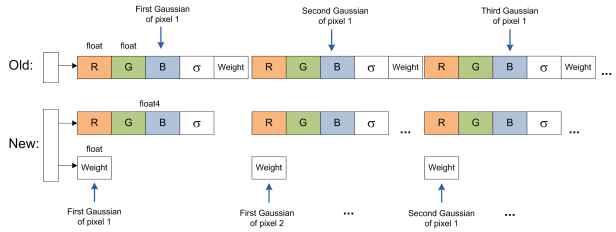
3) Asynchronous execution: A CUDA *stream* is a sequence of commands that execute in order on the device. Commands in difference streams, however, may execute out of order. In CUDA, a stream is defined by creating a stream object and specifying it as the stream parameter to kernel launches and memory transfers between host and device. Using streams and pinned memory, we are able to interleave the CPU code execution with the kernel launches and memory transfers as described in Figure 8.

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. The detailed implementation is given in Algorithm 1.



(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. Separating weights from means and variances helps avoid unnecessary memory transfer, since they have different access patterns from threads. Reordering Gaussian components so that the first components come before the second components helps to increase the chance of memory coalescing.

Figure 7: Improvements on storing the Gaussian parameters arrays.

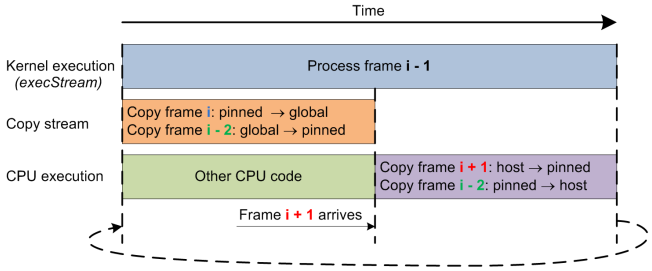


Figure 8: Timeline for asynchronous execution. In the copy stream (the second row), transfers are performed by `cudaMemcpyAsync`; while CPU (the third row) uses `memcpy` as usual.

IV. Experimental Results

The CPU and GPU implementations are executed on the dataset from the VSSN 2006 competition [24] to compare the robustness, and on the PETS2009 [25] dataset to evaluate the speedup. The VSSN2006 dataset contains various 384x240 semi-synthetic (synthetic foreground object moving over a real background) AVI video sequences and the corresponding ground truth sequences. We chose video 2, video 4 and video 8 for evaluation. Video 2 has 500 frames with a relatively static indoor scene. Video 4 has 820 frames and is more dynamic with swaying leafs in the background. Video 8 (1920 frames) uses the same background with video 2, but at the end of the sequence, lamps are turned on and the illumination is changed. PETS2009 is a set of 768x576 JPEG image sequences in outdoor condition. It has 4 subsets, each subset contains several sequences and each sequence contains from 4 up to 8 views. We choose view 1 and view 2 from S1.L1 (subject 1, sequence 1) scenario, view 1 from S1.L2 scenario to test the performance of the implementations. All experiments in this section are performed on a system running Core 2 Quad Q9400 2.6 GHz CPU (4 GB of RAM) and GeForce 9600GT

Algorithm 1 Code segment that implements the asynchronous execution in CUDA.

```
Input: GmmModel, inputImg
Output: outputImg
Begin
  // wait for the copyStream
  Synchronize(copyStream)

  // CPU code: copy to/from pinned memory
  copy(pinnedIn, inputImg, ...)
  copy(outputImg, pinnedOut, ...)

  // wait for the execStream
  Synchronize(execStream)

  // swap pointers
  swap(globalInMemory, globalInMemory2)
  swap(globalOutMemory, globalOutMemory2)

  // copyStream: copy to/from GPU global memory
  CopyAsync(globalInMemory, pinnedIn, ..., copyStream)
  CopyAsync(pinnedOut, globalOutMemory,..., copyStream)

  // execStream: launch the kernel
  UpdateGmmModel<<<..., execStream>>> (globalInMemory2,
       globalOutMemory2)
End
```

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 9. In this figure, kernel 1 is the GPU basic implementation using constant memory as described in Section III-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 III-B3.

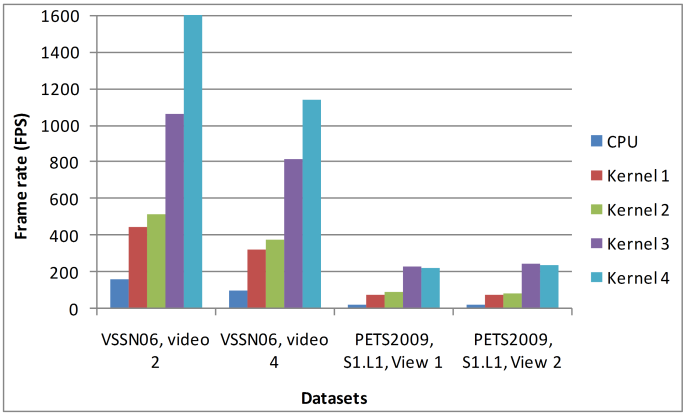


Figure 9: Effectiveness of optimization techniques.

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 video 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. In this section, 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. It depends on number of threads per block, number of registers per thread, capacity of shared memory per block that a kernel uses, and on compute capability of the device. 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 (α from equation (2)) 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 10.

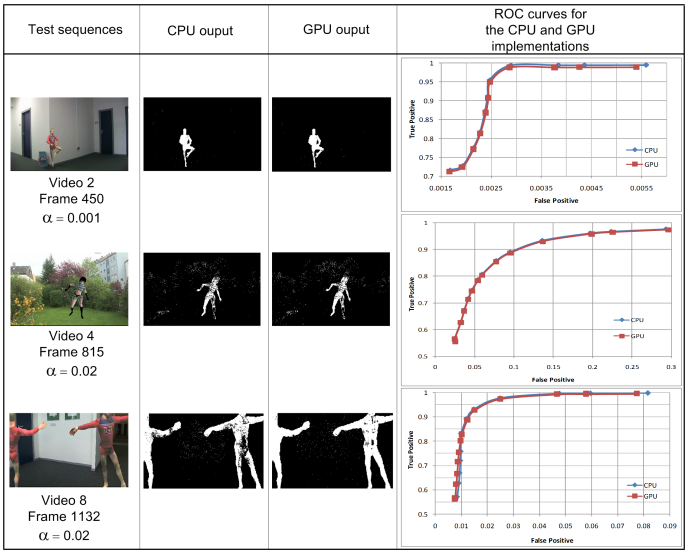


Figure 10: Comparision of the GPU implementation to the CPU implementation. The ROC curves show that the GPU implemetation has nearly the same robustness with the original implementation.

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 11. 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.

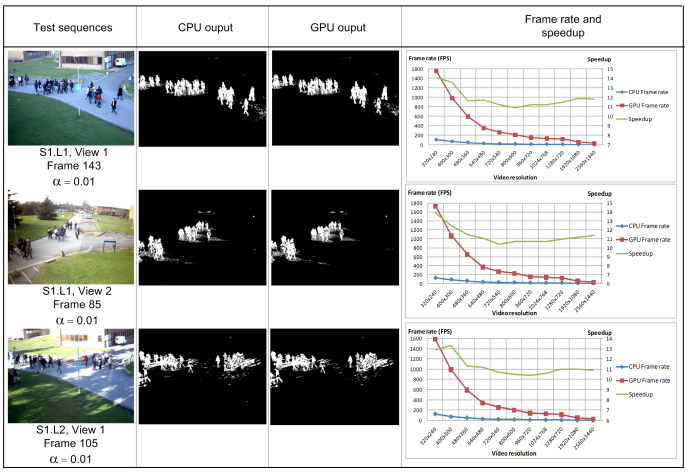


Figure 11: Comparision of the CPU version and GPU version in term of frame rate over different resolutions. Green lines are the speedup. The average speedup for each sequence is about 11x.

V. 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.

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] N. J. B. McFarlane and C. P. Schofield, "Segmentation and tracking of piglets in images," *Machine Vision and Applications*, vol. 8, no. 3, pp. 187–193, May 1995.

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

[2] 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.

[3] C. Stauffer and W. E. L. Grimson, "Adaptive Background Mixture Models for Real-Time Tracking," *Proc. CVPR*, 1999.

[4] N. Oliver, B. Rosario, and A. Pentland, "A Bayesian computer vision system for modeling human interactions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 831–843, 2000.

[5] A. Elgammal, D. Harwood, and L. Davis, "Non-parametric Model for Background Subtraction," in *ECCV*, 2000, pp. 751–767.

[6] S. Calderara, R. Melli, A. Prati, and R. Cucchiara, "Reliable background suppression for complex scenes," *Proceedings of the 4th ACM international workshop on Video surveillance and sensor networks - VSSN '06*, p. 211, 2006.

[7] Y. Benezeth, P. Jodoin, B. Emile, H. Laurent, and C. Rosenberger, "Review and evaluation of commonly-implemented background subtraction algorithms," *2008 19th International Conference on Pattern Recognition*, pp. 1–4, December 2008.

[8] D. H. Parks and S. S. Fels, *Evaluation of Background Subtraction Algorithms with Post-Processing*. IEEE, September 2008.

[9] M. Piccardi, "Background subtraction techniques: a review," *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, pp. 3099–3104, 2004.

[10] R. J. Radke, S. Andra, O. Al-Kofahi, and B. Roysam, "Image change detection algorithms: a systematic survey," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 14, no. 3, pp. 294–307, March 2005.

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

[12] P. W. Power and J. A. Schoonees, "Understanding Background Mixture Models for Foreground Segmentation," *Image and Vision Computing*, no. November, 2002.

[13] P. Kaewtrakulpong and R. Bowden, "An Improved Adaptive Background Mixture Model for Real- time Tracking with Shadow Detection," in *Proc. 2nd European Workshop on Advanced Video Based Surveillance Systems, AVBS01*. Kluwer Academic, 2001, pp. 1–5.

[14] B. Stenger, V. Ramesh, N. Paragios, F. Coetzee, and J. Buhmann, "Topology free hidden Markov models: application to background modeling," *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, vol. 00, no. C, pp. 294–301, 2001.

[15] Z. Zivkovic, "Improved adaptive gaussian mixture model for background subtraction," *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, no. 2, pp. 28–31, 2004.

[16] 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.

[17] NVIDIA Corporation, "NVIDIA CUDA Programming Guide," 2007.

[18] AMD/ATI, "ATI CTM (Close to Mental) Guide," 2007.

[19] J. Fung, "Advances in GPU-based Image Processing and Computer Vision," in *SIGGRAPH*, 2009.

[20] S.-j. Lee and C.-s. Jeong, "Real-time Object Segmentation based on GPU," *2006 International Conference on Computational Intelligence and Security*, pp. 739–742, November 2006.

[21] P. Carr, *GPU Accelerated Multimodal Background Subtraction*. Digital Image Computing: Techniques and Applications, December 2008.

[22] Apple Inc., "Core Image Programming Guide," 2008.

[23] NVIDIA Corporation, "NVIDIA CUDA Best Practices Guide," 2010.

[24] "VSSN 2006 Competition," 2006. [Online]. Available: http://mmc36.informatik.uni-augsburg.de/VSSN06_OSAC/

[25] "PETS 2009 Benchmark Data," 2009. [Online]. Available: http://www.cvg.rdg.ac.uk/PETS2009/a.html