Parallel Implementation of Video Surveillance Algorithms on GPU Architecture using CUDA

Sanyam Mehta[‡], Arindam Misra[‡], Ayush Singhal[‡], Praveen Kumar[†], Ankush Mittal[‡], Kannappan Palaniappan[†]

[‡]*Department of Electronics and Computer Engineering, Indian Institute of Technology, Roorkee, INDIA*

[†]*Department of Computer Science, University of Missouri-Columbia, USA*

E-mail: san01uec@iitr.ernet.in,ari07uce@iitr.ernet.in,ayush488@gmail.com, praveen.kverma@gmail.com, ankumfec@iitr.ernet.in, palaniappank@missouri.edu

Abstract

At present high-end workstations and clusters are the commonly used hardware for the problem of real-time video surveillance. Through this paper we propose a real time framework for a 640×480 frame size at 30 frames per second (fps) on a low costing graphics processing unit (GPU) (GeForce 8400 GS) which comes with many low-end laptops and personal desk- -tops. The processing of surveillance video is computationally intensive and involves algorithms like Gaussian Mixture Model (GMM), Morphological Image operations and Connected Component Labeling (CCL). The challenges faced in parallelizing Automated Video Surveillance (AVS) were: (i) Previous work had shown difficulty in parallelizing CCL on CUDA due to the dependencies between sub-blocks while merging (ii) The overhead due to a large number of memory transfers, reduces the speedup obtained by parallelization. We present an innovative parallel implementation of the CCL algorithm, overcoming the problem of merging. The algorithms scale well for small as well as large image sizes. We have optimized the implementations for the above mentioned algorithms and achieved speedups of 10X, 260X and 11X for GMM, Morphological image operations and CCL respectively, as compared to the serial implementation, on the GeForce GTX 280.

Keywords: GPU, thread hierarchy, erosion, dilation, real time object detection, video surveillance.

1. Introduction

Automated Video Surveillance is a sector that is witnessing a surge in demand owing to the wide range of applications like traffic monitoring, security of public places and critical infrastructure like dams and bridges, preventing cross-border infiltration, identification of military targets and providing crucial evidence in the trials of unlawful activities [11][13]. Obtaining the desired frame processing rates of 24-30 fps in real-time for such algorithms is the major challenge faced by the developers. Furthermore, with the recent advancements in video and network technology, there is a proliferation of inexpensive network based cameras and sensors for widespread deployment at any location. With the deployment of progressively larger systems, often consisting of hundreds or even thousands of cameras distributed over a wide area, video data from several cameras need to be captured, processed at a local processing server and transmitted to the control station for storage etc. Since there is enormous amount of media stream data to be processed in real time, there is a great requirement of High Performance Computational (HPC) solution to obtain an acceptable frame processing throughput.

The recent introduction of many parallel architectures has ushered a new era of parallel computing for obtaining real-time implementation of the video surveillance algorithms. Various strategies for parallel implementation of video surveillance on multi-cores have been adopted in earlier works [1][2], including our work on Cell Broadband Engine [15]. The grid based solutions have a high communication overhead and the cluster implementations are very costly.

The recent developments in the GPU architecture have provided an effective tool to handle the workload. The GeForce GTX 280 GPU is a massively parallel, unified shader design consisting of 240 individual stream processors having a single precision floating point capability of 933 GFlops. CUDA enables new applications with a standard platform for extracting valuable information from vast quantities of raw data. It enables HPC on normal enterprise workstations and server environments for data-intensive applications, eg. [12]. CUDA combines well with multi-core CPU systems to provide a flexible computing platform.

In this paper the parallel implementation of various video surveillance algorithms on the GPU architecture is presented. This work focuses on algorithms like (i) Gaussian mixture model for background modeling, (ii) Morphological image operations for image noise removal (iii) Connected component labeling for identifying the foreground objects. In each of these algorithms, different memory types and thread configurations provided by the CUDA architecture have been adequately exploited. One of the key contributions of this work is novel algorithmic modification for parallelization of the divide and conquer strategy for CCL. The speed-ups obtained with GTX 280(30 multiprocessors or 240 cores) were very significant, the corresponding speed-ups on 8400 GS (2 multiprocessors or 16 cores) were sufficient enough to process the 640×480 sized surveillance video in real-time. The scalability was tested by executing different frame sizes on both the GPUs.

2. GPU Architecture and CUDA

NVIDIA’s CUDA [14] is a general purpose parallel computing architecture that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems. The programmable GPU is a highly parallel, multi-thread, many core co-processors specialized for compute intensive highly parallel computation.

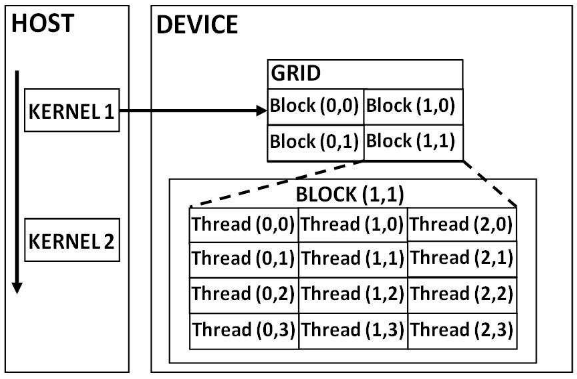


Fig 1 Thread hierarchy in CUDA

The three key abstractions of CUDA are the thread hierarchy, shared memories and barrier synchronization, which render it as only an extension of C. All the GPU threads run the same code and, are very light weight and have a low creation overhead. A kernel can be executed by a one dimensional or two dimensional grid of multiple equally-shaped thread blocks. A thread block is a 3, 2 or 1-dimensional group of threads as shown in Fig. 1. Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. Threads in different blocks cannot cooperate and each block can execute in any order relative to other blocks. The number of threads per block is therefore restricted by the limited memory resources of a processor core. On current GPUs, a thread block may contain up to 512 threads. The multiprocessor SIMT (Single Instruction Multiple Threads) unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*.

The constant memory is useful only when it is required that the entire warp may read a single memory location. The shared memory is on chip and the accesses are 100x-150x faster than accesses to local and global memory. The shared memory, for high bandwidth, is divided into equal sized memory modules called banks, which can be accessed simultaneously. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The banks are organized such that successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per two clock cycles. For devices of compute capability 1.x, the warp size is 32 and the number of banks is 16. The texture memory space is cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. The local and global memories are not cached and their access latencies are high. However, coalescing in global memory significantly reduce the access time and is an important consideration (for compute capability 1.3, global memory accesses are easily coalesced than earlier versions). Also CUDA 2.2 release provides page-locked host memory helps in increasing the overall bandwidth when the memory is required to be read or written exactly once. Also, it can be mapped to device address space – so no explicit memory transfer required.

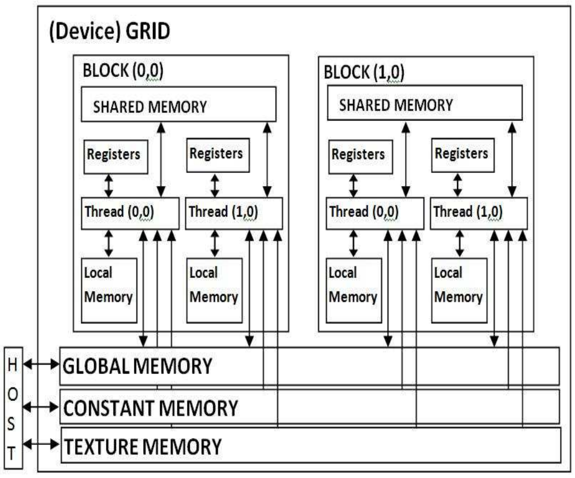


Fig. 2 The device memory space in CUDA

3. Our approach for the Video Surveillance Workload

A typical Automated Video Surveillance (AVS) workload consists of various stages like background modelling, foreground/background detection, noise removal by morphological image operations and object identification. Once the objects have been identified other applications can be developed as per the security requirements. Fig. 3 shows the multistage algorithm for a typical AVS system. The different stages and our approach to each of them are described as follows:

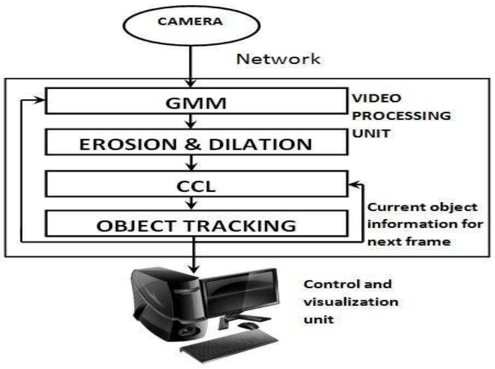


Fig. 3 A typical video surveillance system

3.1 Gaussian Mixture Model

Many approaches for background modelling like [4][5]have been proposed. Here, Gaussian Mixture model proposed by Stauffer and Grimson [3] is taken up, which assumes that the time series of observations, at a given image pixel, is independent of the observations at other image pixels. It is also assumed that these observations of the pixel can be modelled by a mixture of *K* Gaussians (currently, from 3 to 5 are used). Let x^t be a pixel value at time *t*. Thus, the probability that the pixel value x^t is observed at time t is given by:

$$(x^t) = \sum_1^K w_k^t \, N(x^t|\mu_k^t \sigma_k^t) \qquad (1)$$

Where, w_k^t, μ_k^t and σ_k^t are the weights, the mean, and the standard deviation, respectively, of the *k*-th Gaussian of the mixture associated with the signal at time *t*. At each time instant *t* the *K* Gaussians are ranked in descending order using *w=0.75* value (the most ranked components represent the "expected" signal, or the background) and only the first *B* distributions are used to model the background, where

$$B = \arg min_b \left\{\sum_{r=1}^{b} > T\right\} \qquad (2)$$

T is a threshold representing the minimum fraction of data used to model the background. As the parameters of each pixel change, to determine the value of Gaussian that would be produced by the background process depends on the most supporting evidence and least variance. Since the variance for a new moving object that occludes the image is high which can be easily checked from the value of σ.

GMM offers pixel level data parallelism which can be easily exploited on CUDA architecture. Since the GPU consists of multi-cores which allow independent thread scheduling and execution, perfectly suitable for independent pixel computation. So, an image of size m × n requires m × n threads, implemented using the appropriate size blocks running on multiple cores. Besides this the GPU architecture also provides shared memory which is much faster than the local and global memory spaces. In fact, for all threads of a warp, accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts [14] between the threads. In order to avoid too many global memory accesses, the shared memory was utilised to store the arrays of various Gaussian parameters. Each block has its own shared memory (upto 16 KB) which is accessible (read/write) to all its threads simultaneously, which greatly improves the computation on each thread since memory access time is significantly reduced. The value for K (number of Gaussians)is selected as 4, which not only results in effective coalescing [14] but also reduces the bank conflicts. As shown in the Table 1 the efficacy of coalescing is quite prominent.

The approach for GMM involves streaming (Fig. 4) i.e. processing the input frame using two streams allows for the memory copies of one stream to overlap with the kernel execution of the other stream.

```
for i <=2
   do
      create stream i  //cudaStreamCreate
for each stream i
   do
      copy half the image from host  to device
      each stream i. //cudaMemcpyAsync

for each stream i
   do
       kernel execution for each stream i.
       (half image processed)  //gmm<<<….>>>

cudaThreadSynchronize();
```

Fig. 4 Algorithm depicting streaming in CUDA

Streaming resulted in significant speed up in the case of 8400 GS, where the time for memory copies was closely matched to the time for kernel execution, while in case of GTX 280, the speed up was not so significant as the kernel execution took little time, being spread over 30 multiprocessors.

3.2 Morphological Image Operations

After the identification of the foreground pixels from the image, there are some noise elements (like salt and pepper noise) that creep into the foreground image. They essentially need to be removed in order to find the relevant objects by the connected component labelling method. This is achieved by morphological image operation of erosion followed by dilation [6]. Each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbours, depending on the structuring element (Fig. 5) used. In case of dilation (denoted by Θ) the value of the output pixel is the *maximum* value of all the pixels in the input pixel's neighbourhood. In a binary image, if any of the pixels in the neighbourhood corresponding to the structural element is set to the value 1, the output pixel is set to 1. With binary images, dilation connects areas that are separated by spaces smaller than the structuring element and adds pixels to the perimeter of each image object.

In erosion (denoted by Θ), the value of the output pixel is the *minimum* value of all the pixels in the input pixel's neighbourhood. In a binary image, if any of the pixels in the neighbourhood corresponding to the structural element is set to the value 0, the output pixel is set to 0. With binary images, erosion completely removes objects smaller than the structuring element and removes perimeter pixels from larger image objects. This is described mathematically as:

$$A\Theta B = z\left(\hat{B}\right)_z \cap A \neq \emptyset \qquad (3)$$

and

$$A\Theta B = z|(B)_z \subseteq A \qquad (4)$$

where $(\hat{B})$ is the reflection of set B and $(B)_z$ is the translation of set B by point z as per the set theoretic definition.

0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
0	1	0	1	0
0	0	1	0	0

Fig. 5 A 5×5 structuring element

As the texture cache is optimized for 2-dimensional spatial locality, the 2-dimensional texture memory is used to hold the input image; this has an advantage over reading pixels from the global memory, when coalescing is not possible. Also, the problem of out of bound memory references at the edge pixels are avoided by the *cudaAddressModeClamp* addressing mode of the texture memory in which out of range texture coordinates are clamped to a valid range. Thus the need to check out of bound memory references by conditional statements never arose, preventing the warps from becoming divergent and adding a significant overhead.

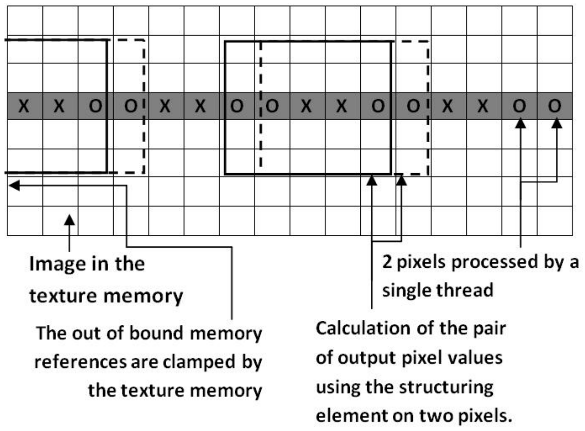

X X O O X X O O X X O O X X O O
Image in the texture memory
The out of bound memory references are clamped by the texture memory
2 pixels processed by a single thread
Calculation of the pair of output pixel values using the structuring element on two pixels.

Fig. 6 Approach for erosion and dilation

As shown in Fig. 6 a single thread is used to process two pixels. A half warp (16 threads) has a bandwidth of 32 bytes/cycle and hence 16 threads, each processing 2 pixels (2 bytes) use full bandwidth, while writing back noise-free image. This halves the total number of threads thus reducing the execution time significantly. A structuring element of size 7×7 was used both in dilation and erosion. A straightforward convolution was done with one thread running on two neighbouring pixels. The execution times for the morphological image operations for the GTX 280 and the 8400 GS are shown in Table 2.

3.3 Connected Component Labelling

The connected component labelling algorithm works on a black and white (binary) image input to identify the various objects in the frame by checking pixel connectivity [8]. The image is scanned pixel-by-pixel (from top to bottom and left to right) in order to identify connected pixel regions, i.e. regions of adjacent pixels which share the same set of intensity values and temporary labels are assigned. The connectivity can be either 4 or 8 neighbour connectivity (8-connectivity in our case). Then, the labels are put under equivalence class, pertaining to their belonging to the same object. After constructing the equivalence classes the labels for the connected pixels are resolved by assigning label of the equivalence class to all the pixels of that object.

Here the approach for parallelizing CCL on the GPU belongs to the class of divide and conquer algorithms [7]. The proposed implementation divides the image into small parts and labels the objects in those small parts. Then in the conquer phase the image parts are stitched back to see if the two adjoining parts have the same object or not.

For initial labelling the image was divided into N×N small regions and the sub-images were scanned pixel by pixel from left to right and top to bottom. These small regions were executed in parallel on different blocks (32×32 in case of 1024×1024 images).

Each pixel was labelled according to its connectivity with its neighbours. In case of more than one neighbour, one of the neighbour's labels was used and rest were marked under one equivalence class. This was done similarly for all blocks that were running in parallel. The equivalence class array was stored in shared memory for each block which saved a lot of memory access time. The whole image frame was stored in texture memory to reduce memory access time, as global memory coalescing was not possible due to random but spatial accesses.

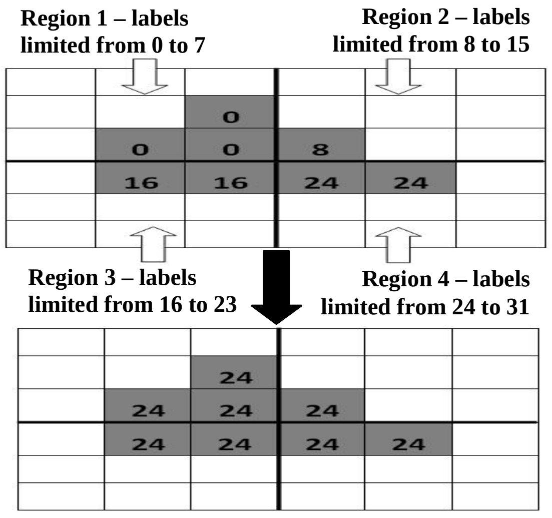

Region 1 – labels limited from 0 to 7
Region 2 – labels limited from 8 to 15
0
0
0
8
16
16
24
24
Region 3 – labels limited from 16 to 23
Region 4 – labels limited from 24 to 31
24
24
24
24
24
24
24
24

Fig 7. The connected components are assigned the maximum label after resolution

In earlier works on CCL like [9][10], the major limitation was that the sub-blocks into which the problem was broken had to be merged serially, the reason being each sub-block had blobs with serial labels and while merging any two connected sub-blocks, the labels in all the other sub-blocks had to be modified.

A new approach to enable parallelization of CCL is presented in this paper. The code (as indicated in Fig. 7) labels the blobs (objects) independent of the other sub-blocks, but according to the CUDA thread ids (i.e. 1st sub-block can label the blobs from 0 to 7, the 2nd sub-block can label the blobs from 8 to 15 and so on). So in this case no sub-block can detect more than 8 blobs (which is generally the case, but one may easily choose to have a higher limit). In order to avoid conflicts between sub-blocks, connected parts of the image, in different regions, were given the highest label from amongst the different labels in different regions, as shown in the fig. 7.

So, as a result of making the entire code 'portable on GPU', the speed up obtained was enormous – the entire processing being split and made parallel to be executed on the GTX 280, resulting in the entire CCL (i.e. including merge) code to be executed in just 2.4 milliseconds (Table 3) for a 1024×768 image, a speed-up of 11x as compared to sequential code.

Table 1 CUDA Profiler output for 1024X768 image size

Function	Occup-ancy	Coalesced global memory loads	Coalesced global memory stores	Total branches	Divergent Branches	Static Memory per block	Total global memory loads	Total global memory stores
GMM	0.75	5682	826	829	27	3112	5682	347
GMM	0.75	5094	186	500	5	3112	5094	93
Erode	1	0	102	408	2	20	0	37
Dilate	1	0	154	4975	31	20	0	77
CCL	0.25	2657	1902	4128	0	536	2657	823
Merge	0.25	9898	36	1936	2	1064	9898	18

4. EXPERIMENTAL RESULTS

The parallel implementation of the above mentioned AVS workload was executed on two NVIDIA GPUs, the first GPU used is the GeForce GTX 280 on board a 3.2 GHz Intel Xeon machine with 1GB of RAM, the second one was the GeForce 8400 GS on board a 2 GHz Intel Centrino Duo machine. The GTX 280 has a single precision floating point capability of 933GFlops and a memory bandwidth of 141.7 GB/sec, it also has 1 GB of dedicated DDR3 video memory and consists of 30 multiprocessor with 8 cores each, hence a total of 240 stream processors. It belongs to a compute category 1.3 which supports many advanced features like page-locked host memory and those which take care of the alignment and synchronization issues. The 8400 GS has a memory bandwidth of 6.4 GB/sec and has two multiprocessors with 8 cores each, i.e. 16 stream processors, single precision floating point capability of 28.8 GFlops and 128 MB of dedicated memory. It belongs to the compute capability 1.2. The development environment used was Visual Studio 2005 and the CUDA profiler version 2.2 was used for profiling the CUDA implementation. The image sizes that have been used are 1600×1200, 1024×768, 640×480, 320×240 and 160×120. In the subsequent discussion we mention the results obtained for the image size of *1024×768* on the GTX 280 (30 multiprocessors).

The Gaussian Mixture Model, used for background modelling has the kind of parallelism that is required for implementation on a GPU. As evident by the Fig. 8, the time of execution increases with the increase in image size and the amount of speedup achieved also increases almost proportionately; this is due to the execution of a large number of threads that keeps the GPU busy. Hence, a significant speedup of 10x has been achieved for 320×240 image.

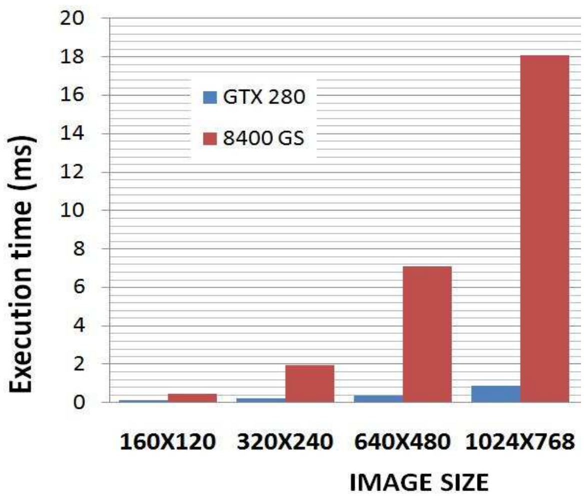


Fig. 8 Execution times for GMM (Speed up = 10 for image size 320×240, as compared to sequential code)

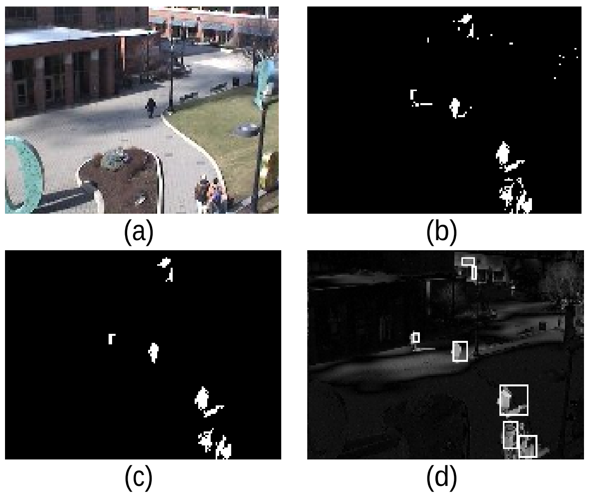
Table 2. Execution times for erosion and dilation for a 3×3 structuring element.

IMAGE SIZE	GTX 280 (ms)	8400 GS (ms)
160×120	0.0445	0.120
320×240	0.0586	0.465
640×480	0.1254	1.75
1024×768	0.2429	3.61
1600×1200	0.5625	11.7

Shared memory was used to reduce the global memory accesses keeping in view the shared memory size (16 KB). As can be seen from the Table 1 a total of 4 blocks (192×4 threads out of a maximum of 1024 threads) could be executed in parallel on a multiprocessor, giving an occupancy of 0.75.As a result of using K=4 all the global memory loads were coalesced as can be seen from the Table 1 also, there were lesser bank conflicts. The use of streaming reduced memory copy overhead but not to the extent anticipated, due to the efficient memory copying in GTX 280 - compute capability 1.3. This approach however, was of great help in the 8400 GS – compute capability1.2.

The morphological image operations contribute a major portion of the computational expense of the of the AVS workload. In our approach we are able to drastically reduce their execution time. The speedup scales with the image size both on the GTX 280 and the 8400 GS, the comparison of sequential code with the parallel implementation of the 1024×768 image size shows a significant speedup of 260X with a structuring element of 7×7. The time taken by the sequential implementation was 89.806 ms as compared to the 0.352 ms taken by the parallel implementation.

For this image size we were able to unleash the full computational power of the GPU with occupancy of 1. (i.e. neither shared memory, nor the registers per multiprocessor were the limiting factors) 280, as indicated in the Table 1. Moreover, the use of texture memory and address clamp modes have reduced the percentage of divergent threads to <1%. On the 8400 GS also a significant speedup has been achieved.



(a) Input Image (b) Foreground Image
(c) Image after noise removal (d) Output

Fig. 9 Image output of various stages of AVS

In CCL (i.e. CCL & Merge) 32×32 sized *independent* sub-blocks were assigned to each thread and 32 threads were run on one block (which was experimentally observed to be optimal). Since, the maximum number of active blocks on a multiprocessor can be 8, the total number of active threads per multiprocessor were 256 and hence an occupancy of 0.25. The optimal parallelization of the CCL algorithm was significant in itself, as the parallelization of CCL on CUDA has not been reported and was deemed very difficult. Apart from the code been parallelized, the use of shared memory and then texture memory to store appropriate data led to significant increases in speedup. The use of texture memory not only prevented any warps from diverging by avoiding the conditional statements (due to clamped accesses in texture memory) but also led to speedup due to the spatial locality of references in CCL. However, the implementation of CCL is block size dependent, which still remains a bottleneck.

Table 3 Execution times for CCL

IMAGE SIZE	GTX 280 (ms)	8400 GS (ms)
160×120	0.106	0.522
320×240	0.220	1.34
640×480	1.256	4.5
1024×768	2.494	14.1
1600×1200	2.649	46.2

In each of the above kernels page-locked host memory (a feature of CUDA 2.2) has been used whenever only one memory read and write were involved which increased the memory throughput.

Architectures dedicated to video surveillance cost as much as lakhs of rupees, while the GeForce GTX 280 costs Rs.17000 and the 8400 GS costs merely Rs.4000. Even for an image size of 640×480, 30 frames per second could be processed on 8400 GS, for an image size of 1024×768 close to 15 frames per second could be processed and for images of smaller size 30 frames could be easily processed as shown in the fig 10.

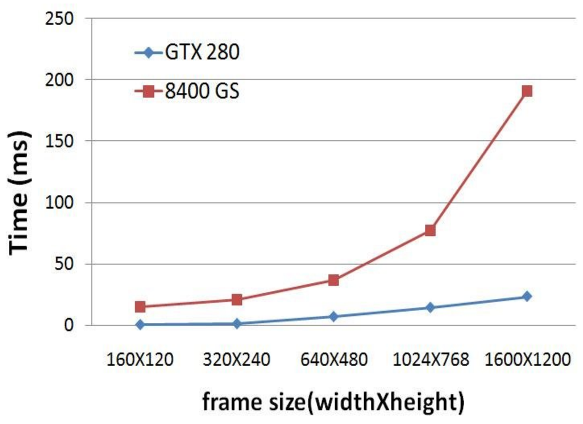


Fig. 10 Comparision of total time for image of different sizes

5. CONCLUSION AND FUTURE WORK

Through this paper, we describe the implementation of a typical AVS workload on the parallel architecture of NVIDIA GPUs to perform real time AVS. The various algorithms, as described in the previous sections are GMM for background modelling, morphological image operations, and CCL for object identification. In our previous work [15] a detailed comparison has been done between Cell BE and CUDA for these algorithms. During the implementation on the GPU architecture, major emphasis was given to selecting the thread configurations, the memory types for each kind of data, out of the numerous options available on GPU architecture, so that the memory latency can be reduced and hidden. Lot of emphasis was given to memory coalescing and avoiding bank conflicts.

Efficient usage of the different kinds of memories offered by the CUDA architecture and subsequent experimental verification resulted in the most optimal implementations. As a result, significant overall speed-up was achieved.

Further testing and validations are going on. We have examined the performance on only 8400 GS(2 multiprocessors) and GTX 280(30 multiprocessors) in this paper, hence a range of intermediate devices are yet to be explored. Our future work will include the implementation of the AVS workload on other GPU devices to examine the scalability, as well as comparison with other parallel architectures to get an idea of their viability as compared to the GPU implementation.

6. REFERENCES

[1] S. Momcilovic and L. Sousa. A parallel algorithm for advanced video motion estimation on multicore architectures. Int. Conf. Complex, Intelligent and Software Intensive Systems, pp 831-836, 2008.

[2] M. D. McCool. Data-Parallel Programming on the Cell BE and the GPU Using the RapidMind Development Platform. *GSPx Multicore Applications Conference*, 9 pages, 2006.

[3] C. Stauffer and W. Grimson,Adaptive background mixture models for real-time tracking, In Proceedings CVPR,pp.246–252,1999.

[4] Zoran Zivkovic, Improved Adaptive Gaussian Mixture Model for Background Subtraction. In Proc. ICPR,pp 28-31 vol. 2,2004

[5] Toyama, K.; Krumm, J.; Brumitt, B.; Meyers, B., Wallflower: principles and practice of background maintenance. The Proceedings of the Seventh IEEE International Conference on Computer Vision, vol.1, pp.255-261, 20-25 September, 1999, Kerkyra, Corfu, Greece

[6] H.Sugano and R.Miyamoto. Parallel implementation of morphological processing on cell/be with opencv interface. Communications, Control and Signal Processing, 2008. ISCCSP 2008, pp 578–583, 2008.

[7] J. M. Park, G. C. Looney, and H. C. Chen, " A Fast Connected Component Labeling Algorithm Using Divide and Conquer", CATA 2000 Conference on Computers and Their Applications, pp 373-376, Dec. 2000.

[8] R. Fisher, S. Perkins, A. Walker and E. Wolfart Connected Component Labeling, 2003

[9] K.P. Belkhale and P. Banerjee, "Parallel Algorithms for Geometric Connected Component Labeling on a Hypercube Multiprocessor," IEEE Transactions on Computers, vol. 41, no. 6, pp. 799-709,1992

[10] M. Manohar and H.K. Ramapriyan. Connected component labeling of binary images on a mesh connected massively parallel processor. Computer vision, Graphics, and Image Processing, 45(2):133-149,1989.

[11] K.Dawson-Howe, "Active surveillance using dynamic background subtraction," Tech. Rep.TCD-CS-96-06, Trinity College, 1996.

[12] Michael Boyer, David Tarjan, Scott T. Acton†, and Kevin Skadron, Accelerating Leukocyte Tracking using CUDA:A Case Study in Leveraging Manycore Copocessors,2009.

[13] A.C. Sankaranarayanan, A. Veeraraghavan, and R. Chellappa. Object detection, tracking and recognition for multiple smart cameras. *Proceedings of the IEEE*, 96(10):1606–1624,2008.

[14] NVIDIA CUDA Programming Guide,Version 2.2, page10,27-35,75-97,2009.

[15] Praveen Kumar, Kannappan Palaniappan, Ankush Mittal and Guna Seetharaman. Parallel Blob Extraction using Multicore Cell Processor. Advanced Concepts for Intelligent Vision Systems (ACIVS) 2009. LNCS 5807, pp. 320–332, 2009.