High Performance Object Detection on Big Video Data using GPUs

Praveen Kumar
Dept. of Computer Science and Engineering
The LNM Institute of Information Technology
Jaipur, India
praveen.kverma@gmail.com

***Abstract*— High resolution cameras have become inexpensive, compact and ubiquitously present in smart phones and surveillance systems. As a result huge volumes of images and video data are being generated daily. This availability of big video data has created challenges to video processing and analysis. Novel and scalable data management and processing frameworks are needed to meet the challenges posed by the big video data. This paper focuses on the first step in meeting this challenge that is to have high performance processing of big video data using GPUs. Parallel implementation of video object detection algorithm is presented along with fine grain optimization techniques and algorithm innovation. Experimental results show significant speedups of the algorithms resulting in real time processing of HD and beyond HD (like panoramic) resolution videos.**

Keywords-Big video data, video surveillance, foreground object detection, GPU implementation

I. INTRODUCTION

The growth in sensor and network technology has opened up different alternatives to greatly enhance the surveillance capabilities. Cameras with HD and beyond HD (like panoramic) resolution are increasingly being used for surveillance purpose. Recent developments in network technology and IP based cameras have further increased the flexibility of installation, anywhere at considerably low cost. This has resulted in deployment of large scale Video Surveillance (VS) systems with potentially thousands of cameras distributed over widespread geographical locations [1]. Thus, VS data is becoming the biggest big data [2]. The availability of massive images has created the need to address fundamental challenges in three key aspects of image processing and analysis, namely storage, processing and transport.

Video processing and analytics of such large amount of data becomes the bottleneck. VS algorithms represent a class of problems that are both computationally intensive and bandwidth intensive. For example, a benchmark done by a team of researchers in Intel [3] reported that major computationally expensive module like background modelling and detection of foreground regions consumes 1 billion microinstructions per frame of size 720x576. On a sequential machine with 3.2 GHz Intel Pentium 4 processor, it takes 0.4 sec to process one frame. Obtaining the desired frame processing rates of 20-30 fps in real-time for such algorithms is the major challenge faced by the developers. For processing video data from several camera feeds with high resolution HD (1920x1080) and beyond, it becomes imperative to investigate approaches for high performance by parallel processing.

We have also seen remarkable advances in computing power and storage capacity of modern architectures. Multi-core architectures and GPUs provide energy and cost efficient platform and some efforts (including our previous works) have leveraged these coprocessors to meet the real-time processing needs [4, 5, 6]. However, the potential benefits of these architectures can only be realized by extracting data level parallelism and developing fine-grained parallelization strategies. This paper presents high performance implementation of video object detection which forms the core of visual computing in a number of applications including video surveillance [7, 8]. The paper focuses on algorithms like (i) Gaussian mixture model for background modelling, (ii) Morphological image operations for image noise removal, (iii) Connected Component Labelling (CCL) for identifying the foreground regions which are used at successive stages of moving object detection and tracking algorithm. 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 CCL algorithm where parallelism is limited by various data dependencies. The scalability was tested on different frame sizes by executing the parallel code on Tesla C2070 GPU. Speedups obtained for different algorithms are impressive and yield real-time processing capacity of beyond HD resolution videos.

II. MOVING OBJECT DETECTION ALGORITHM

Figure 1 shows the multistage algorithm for video object detection. The different stages are briefly outlined as follows:

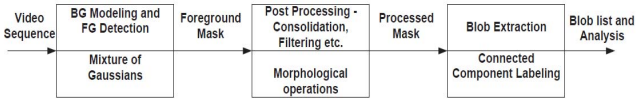

Fig 1. Overview of moving object detection algorithms in video surveillance workload

978-1-4799-8688-0/15 $31.00 © 2015 IEEE
DOI 10.1109/BigMM.2015.65

A. Background modelling and detection of foreground regions

This step forms the bulk of computation which varies depending on the complexity and robustness of the algorithm used. We use pixel-level Gaussians mixture background model which has been used in a wide variety of systems because of its efficiency in modeling multi-modal distribution of backgrounds and its ability to adapt to a change of the background. It models the intensity of every pixel by a mixture of K Gaussian distribution and hence becomes computationally very expensive for large image size and value of K. Furthermore, there is high degree of data parallelism in the algorithm as it involves independent operations for every pixel. Thus, compute intensive characteristic and available parallelism makes GMM suitable candidate for parallelization. However, to meet the memory requirement of this algorithm in the constraint memory of embedded cores becomes a challenge.

B. Consolidation, filtering and elimination using binary morphology

With a suitable threshold operation a binary image or mask corresponding to moving regions is created. Morphological operations "opening" and "closing" are applied to clean-up spurious responses to detach touching objects and fill in holes for single objects. There is high degree of parallelism in this step, and it is computationally expensive step as these operators have to be applied in several passes on the whole image. Therefore, faster parallel implementation is not only intuitive but indispensable for real-time processing.

C. Detection of connected components

In principle, the binary image resulting from above step must have one connected region for each moving object. These regions or blobs must be uniquely labelled, in order to uniquely characterize the object pixels underlying each blob. Since there is spatial dependency at every pixel, it is not straightforward to parallelize it. Although the underlying algorithm is simple in structure, the computational load increases with image size and the number of objects - the equivalence arrays become very large and hence the processing time. Furthermore, with all other steps being processed in parallel with high throughput, it becomes imperative to parallelize this step so as to avoid it from becoming a bottleneck in the processing stream.

III. GPU IMPLEMENTATION

A. Gaussian Mixture Model (GMM)

A GMM is a statistical model that assumes that the data originates from a weighted sum of several Gaussian distributions. Stauffer and Grimson [9, 10] presented an adaptive GMM method to model a dynamic background in image sequences. If K Gaussian distributions are used to describe the history of a pixel, the observation of the given pixel will be in one of the K states at any time [9]. The details of the procedure of foreground labelling using GMM can be found in the above referred papers.

GMM offers pixel level data parallelism which can be easily exploited on CUDA architecture. Since the GPU consists of multiple cores which allow independent thread scheduling and execution, it is 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 between the threads. So in order to avoid too many global memory accesses, we utilized this shared memory to store the arrays of various Gaussian parameters. Each block has its own shared memory which is accessible (read/write) to all its threads simultaneously. This greatly improves the computation on each thread since memory access time is significantly reduced. In our approach we have used K (number of Gaussians) as 4 which not only results in effective coalescing but also reduces the bank conflicts.

Our approach for GMM involves streaming (similar to the popular double buffering mechanism) i.e. we process the input frame using two streams. As a result, the memory copying of one stream (half the image) overlaps (in time) with the kernel execution (application of GMM) of the other stream. Streaming gives good results because the time for memory copying was closely matched to the time for kernel execution.

B. 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 [11]. Each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors, depending on the structuring element.

As the texture cache is optimized for 2-dimensional spatial locality, in our approach we have used the 2-dimensional texture memory 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 doesn't arise, preventing the warps from becoming divergent and adding a significant overhead.

As shown in Figure 2, 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 straightforward convolution was done with one thread running on two neighboring pixels.

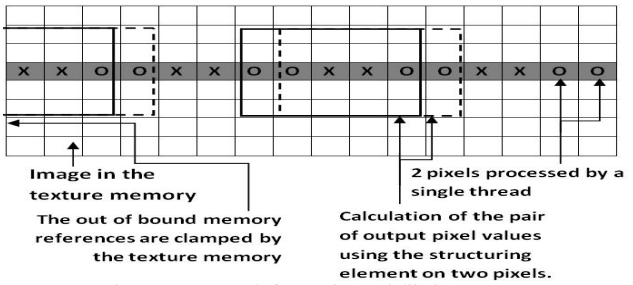

Fig 2. Our approach for erosion and dilation on GPU

C. 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 [12, 13]. 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. Then, the labels are put under equivalence class, pertaining to their belonging to the same object.

The second step resolves equivalence class for the intermediate labels generated in the first scan. We explore two algorithms for our parallel implementation. One algorithm uses *Union-Find* operations that resolves equivalence class labels using a set of trees often implemented as an array data structure with additional operations to maintain a more balanced or shallow tree [14]. We also explored another fundamental algorithm from graph theory known as the *Floyd-Warshall* (F-W) algorithm that expresses equivalent relations as a binary matrix and then resolves equivalences by obtaining transitive closure of the matrix.

Here the approach for parallelizing CCL on the GPU belongs to the class of divide and conquer algorithms [15]. The proposed implementation divides the image into small tile like regions. Then labelling of the objects in each region is done independently by parallel threads running on GPU. The result after the divide phase is shown in figure 3 (a). Then in the conquer phase the local labels in the regions are merged into global labels such that if any object spans over multiple regions, all the pixels that are part of one object are assigned unique label globally with respect to the entire image. The result after the merge phase is shown in figure 3 (b).

In the first phase, we divide the image into NxN smaller regions such that each region has same number of rows and columns. The value of N, number of rows and columns in each region is chosen according to the size of image and the number of CUDA cores on the GPU. Each pixel is labelled according to its connectivity with its neighbors as described earlier. In case of more than one neighbor, one of the neighbor's labels was used and rest were marked under one equivalence class. This was done similarly for all blocks that were running in parallel.

Our procedure for merge phase is inspired from [15]. In this phase, we connect each region with its neighbor regions to generate the actual label within the entire image. We use NxN pointers Global_List[i] to point to arrays that maintain the global labels with respect to the entire image. Global_List[i] points to the array for Region [i] where each array element is the global label within the entire image and the index for each array element is the local label within Region[i]. Memory allocation for each array pointed to by Label_List[i] can be done dynamically according to the maximum local label in Region[i].

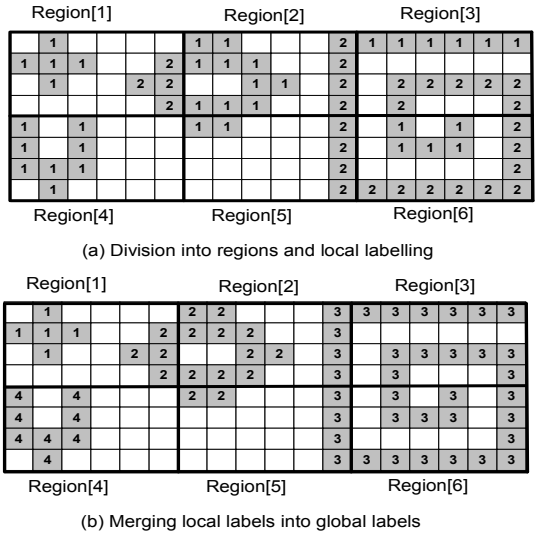

Fig 3. A sample example of dividing image into tile like regions and then merging. (a) shows the local labels given to each region by independent threads (b) shows the global labels after merging the local labels from all the regions.

In the merge phase, we compare the labels in the first row and column of a region with the labels in the adjoining row or column of the adjacent regions. There are three cases to be handled. The first case is of the starting pixel of each region whose label has to be merged with the labels of five adjacent pixels (for 8 connectivity) in Region [i-1], Region [i-N] and Region [i-N-1] as shown in figure 4 (a). The second case is of the remaining pixels in the first column of the region and third case is of the remaining pixels in the first row of the region as shown in figure 4(b) and (c) respectively.

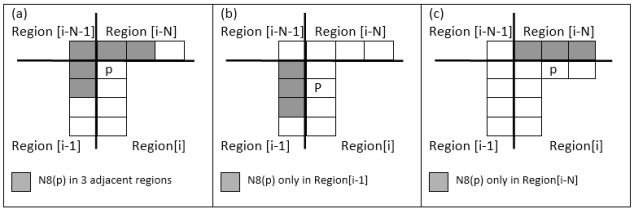

Fig 4. Merge takes place in three ways at Region[i]. a) Merge at the first pixel in Region[i] b) Merge at remaining pixels in the first column in Region[i] and the last column in region[i-1] c) Merge at pixels in the first row in Region[i] and the last row in Region[i-N].

IV. Results and Discussion

The parallel implementation of the above algorithms in moving object detection was executed on NVIDIA Tesla C2070 GPU on board Lenovo D20 Thinkstation - a 3.2 GHz Intel® Xeon® X5650 machine with 6 core processor @ 2.66 GHz and 8 GB of RAM. The Tesla C2070 GPU has a staggering 448 cores delivering peak performance of over a

teraflop for single precision and 515 GFlops of double precision along with a huge 6GB of dedicated memory facilitating storage of larger data sets in local memory to reduce data transfers. The image sizes that have been used range from standard dimension of 320×240, 720×480, XGA size of 1024x768, half HD 1360x768, full HD 1920x1080 and beyond HD 2560x1440 (as in panoramic images).

For the experiment purpose, high resolution video data collection was done using professional quality HD camera (Sony HXR MC-50P) and the panoramic camera in several real life surveillance scenarios, including the college campus. Figure 5 shows some sample frames at different time instant selected from HD video data collected at the college campus.

Fig 5. Sample frames from HD video data collected at college campus.

As can be seen from figure 9, HD camera could view only a portion of the wide campus at any instant of time. To capture the entire campus view in single video, Arecont Vision AV20185DN panoramic camera was used. AV20185DN uses four sensors to capture footage over a 180° panoramic view at a combined resolution of 20 megapixels, equivalent to 65 analog cameras. Fig 6 shows sample video frames coming from four different channels of the AV20185DN camera which are archived in separate folders. These four frames correspond to same point of time and they were concatenated to get one panoramic video frame of resolution 10240x1920 (~20 megapixel) covering the entire campus area as shown in figure 7.

Fig 6. Video frames from 4 different channels of the AV20185DN camera taken at college campus

A. *Qualitative Analysis*

Figure 7 shows the output on a sample panoramic video frame (10240x1920) from the college dataset. The first row shows the original input, the second shows the output of background subtraction using GMM and the third row shows the output after morphological processing which has noise removed and blobs more consolidated. The final output is after CCL showing the bounding box around the detected objects.

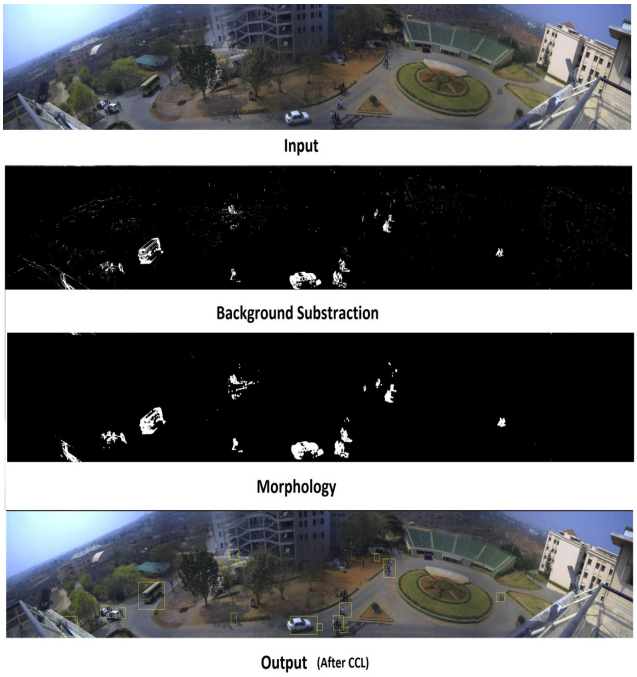

Fig 7: A sample output on the panoramic video frame (10240x1920) extracted from our college dataset.

B. *Parallel GMM Performance*

Figure 8 compares the performance of sequential and parallel implementations of GMM algorithm on different frame sizes. Result shows significant speedup for parallel GPU implementation going upto 15.8x when compared to sequential execution on Intel Xeon processor and upto 70.7x in comparison to sequential execution on Intel core2 duo processor.

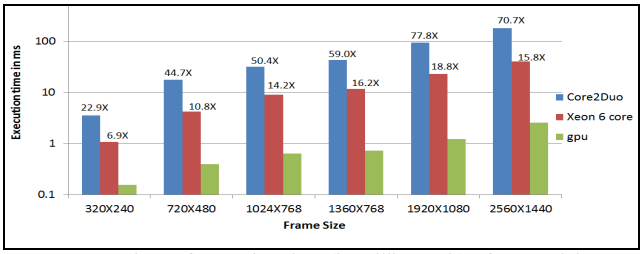

Fig 8. Comparison of execution time (in milliseconds) of sequential versus parallel GPU implementation of GMM algorithm on different video frame sizes. The time axes is plotted on logarithmic scale.

Apart from the image data, GMM stores the background model information of mean(μ), sigma(σ) and weight (w) values in the memory which needs to be transferred back and forth from host to device memory for successive frame computation. This causes lot of delay due to memory copy (cudaMemcpy) operations. However, by streaming concept, we try to overlap the communication with computation time and hence were able to overcome memory latency. It can be observed that the speedup increases with image sizes because of increase in the total number of CUDA threads which keeps the cores busy. For optimization, we used shared memory for storing various Gaussian mixture values which reduced the memory access

time but it had a side effect of decreasing the occupancy. To balance the use of shared memory, we chose 192 threads to be executed per block and created blocks in 1D grid. It was noted that shared memory can be used with only floating point and also the data types which are type compatible with float (like int, short etc) and not with unsigned char data types.

C. *Parallel Morphology Performance*

Figure 9 compares the performance of sequential and parallel implementations of Morphology algorithm (time measured is for one erosion and dilation operation with 5x5 structuring element) on different frame sizes. As shown in the figure, we are able to get enormous speedup for parallel GPU implementation going upto 253.6x when compared to sequential execution on Intel Xeon processor and upto 696.7x in comparison to sequential execution on Intel core2 duo processor. We are able to get very high speedup for this because of several factors. First of all, the input data to this algorithm is binary values for each pixel which needs very less memory for storing the entire frame and thus the time for memory copy operation is reduced to minimal. Moreover, we could optimize memory access time during computation by storing the binary image in texture memory which is a read only memory and also cached. As the morphology operation on neighboring pixel use data which have spatial locality, this optimization reduces access time considerably without having to use shared memory. Thus we could also get maximum occupancy of 1. Also we made the implementation generic such that it will work for structuring element of any size. Again the speedup increased with increase in image size due to the generation of more number of CUDA threads thus utilizing the 448 cores of Tesla card fully.

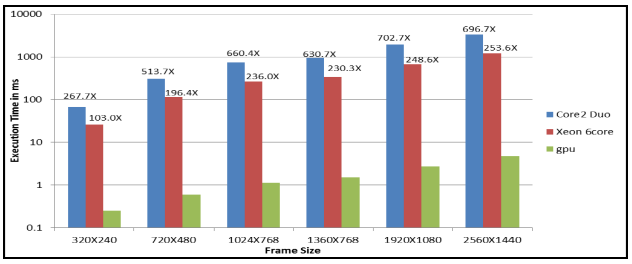

Fig 9. Comparison of execution time (in milliseconds) of sequential versus parallel GPU implementation of Morphology algorithm on different video frame sizes. The time axes is plotted on logarithmic scale.

D. *Parallel CCL Performance*

We implemented the first pass of CCL labeling and resolving equivalence labels using both Union-Find (UF) and Floyd-Warshall (FW) algorithm, resulting in two different sequential implementations. For the parallel implementation also we use these two algorithms for local labeling on individual GPU core and the same divide and merging strategy as discussed in section III (C). Thus we have two different parallel implementations on GPU for performance measurement against their sequential counterparts. Figure 10 and 11 compares the performance of sequential and parallel implementations of CCL algorithm using UF and FW respectively, on different frame sizes.

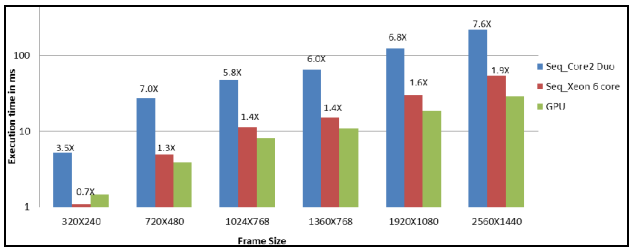

Fig 10. Comparison of execution time (in milliseconds) of sequential versus parallel GPU implementation of *CCL Union-Find* algorithm on different video frame sizes.The time axes is plotted on logarithmic scale.

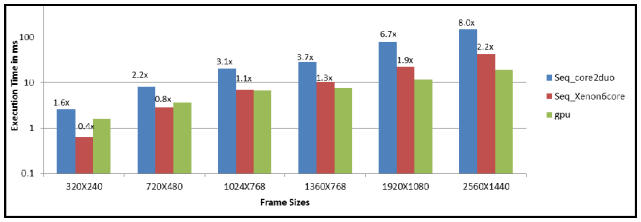

Fig 11. Comparison of execution time (in milliseconds) of sequential versus parallel GPU implementation of *CCL Floyd-warshall* algorithm on different video frame sizes.The time axes is plotted on logarithmic scale.

Although CCL algorithm was difficult to parallelize due to many dependencies, we could obtain reasonable speedups values for parallel GPU implementations, especially on large image sizes (upto 7.6-8.0x) as can be seen from the figure 10 and 11. This is because for smaller image sizes, the overhead involved in dividing the data and merging the labels, overshadows the gain by parallelizing the computation. The overhead ratio reduces when the amount of useful computation increases more than the overhead computation on larger image sizes. Again, we tried to extensively utilize both shared and texture memory for optimizing the memory accesses. We stored the input image pixel values in texture memory which is read only. We used shared memory for storing equivalence array which had to be frequently accessed during first labeling pass and also during local resolution of labels. We tried to reduce the number of branch conditions (which cause thread divergence) by replacing many if-else conditions with switch statements.

In order to increase the coalesced memory accesses and occupancy of blocks, we tried to increase the number of threads per block (value upto 128). Also for larger frame size, we could increase the number of blocks in grid which also resulted in increasing the total number of CUDA threads. However, it was observed that increasing the number of CUDA threads increased the performance only upto certain limit. This limit varied for different frame sizes. After the limit is reached, increasing the threads starts decreasing the performance. This phenomenon can be observed in fig 12 which shows how the performance of UF and FW CCL implementation on GPU varies by increasing the total number of thread. Both of them reach an optimum performance at certain limit after which the execution time starts increasing.

By experimentation, we determined this optimum limit value for different implementation on different frame sizes.

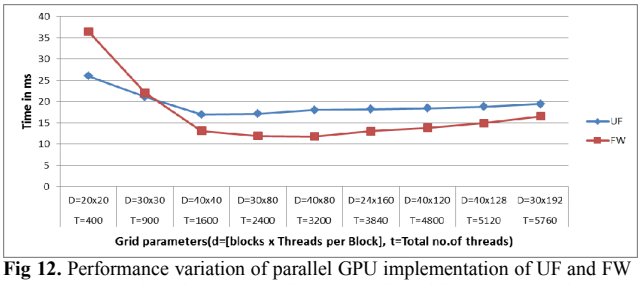

Fig 12. Performance variation of parallel GPU implementation of UF and FW CCL on Frame size of 1920x1080 by varying the grid parameters to increase the total number of CUDA threads.

E. End-to-End Performance

In order to test the performance of the end-to-end system, we connected the individual processing modules in the sequence of video object detection algorithm. Table 1 shows the time taken and the frame processing rate, that was achieved for sequential (running on Intel Xeon) and parallel GPU implementation on different video resolution. Since we used a typical video surveillance dataset, we used FW algorithm for CCL implementation which was performing better than UF as showed by our results above. We exclude the time taken for fetching the image frames from stream/disk IO in our performance measurement. As can be observed from the table, that the number of frames processed per second for sequential implementation is very low whereas the parallel GPU implementation can process even HD resolution videos and beyond HD(like panoramic) videos in real time at 22.3 and 13.5 frames per second respectively.

TABLE I. COMPARISON OF EXECUTION TIME (IN MILLISECONDS) AND FRAME PROCESSING RATE OF SEQUENTIAL VERSUS PARALLEL GPU IMPLEMENTATIONS OF END-TO-END SYSTEM OF MOVING OBJECT DETECTION.

Image Size	1024X768	1360x768	1920x1080	2560x1440
Seq. UF	T=28.565 Fps=35.0	T=131.659 Fps=7.6	T=299.941 Fps=3.3	T=398.765 Fps=2.5
Seq. FW	T=8.716 Fps=114.7	T=15.543 Fps=64.3	T=23.368 Fps=42.8	T=31.853 Fps=31.4
GPU UF	3.277	8.4706	12.8355	12.5189

V. CONCLUSIONS AND FUTURE WORK

In this paper, we describe parallel implementation of the core video object detection algorithm for video surveillance using GPUs to achieve real time processing on high resolution video data. The various algorithms described in this paper are GMM, morphological operations, and CCL. We came up with novel parallelization strategies for extracting data parallelism in these algorithms. For optimization, major emphasis was given to reduce the memory latency by extensively utilizing shared and texture memory wherever possible and optimizing the number of threads and block configuration for maximum utilization of all the GPU cores. Care was taken for memory coalescing and avoiding bank conflicts. We compared the performance of sequential and parallel GPU implementation of the GMM, Morphology and CCL algorithms. Experimental results show that the GPU parallel implementation achieved significant speedups upto a factor of ~250x for binary morphology, ~15x for GMM and upto ~2.2x for CCL in comparison to sequential implementation running on Intel Xeon 6 core processor. When compared with core2 Duo processor, the parallel GPU speedups reached very high speed-up upto factors of ~696x for binary morphology, ~70x for GMM and ~8x for CCL algorithm. The parallel GPU end-to-end system can process HD resolution videos and beyond HD (like panoramic) videos in real time at 22.3 and 13.5 frames per second respectively.

We have examined the performance on only Tesla C2070 GPU in this work. Our future work will include testing the current implementation on very high resolution (20 megapixel) panoramic videos and multiple feeds to measure its scalability and efficacy. Also we will attempt to implement these algorithms on AMD GPUs using OpenCL and compare the performance of multi-vendor GPUs. Intel MIC architecture (Knights corner) is another highly promising architecture for exascale computing which we intend to explore for big image and video data processing.

REFERENCES

[1] P. Kumar, S. Roy, A. Mittal and P. Kumar. OS-Guard: A Novel Framework for Multimedia Surveillance Data Management. Journal of Multimedia Technology and Application, 59(1): 363-382, 2012

[2] T. Huang, "Surveillance Video: The Biggest Big Data," Computing Now, vol. 7, no. 2, Feb. 2014, IEEE Computer Society [online].

[3] K. Jefferson and C. Lee. Computer vision workload analysis - casestudy of video surveillance systems. Intel Technology Journal, 09(02),2005.

[4] 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.

[5] P. Kumar, K. Palaniappan, A. Mittal and G. Seetharaman. Parallel Blob Extraction using Multicore Cell Processor. Advanced Concepts for Intelligent Vision Systems 2009. LNCS 5807, pp.320–332, 2009.

[6] P. Kumar, S. Mehta, A. goyal and A. Mittal. Real-time Moving Object Detection algorithm on High resolution Videos using GPUs. Journal of Real-Time Image Processing. DOI: 10.1007/s11554-012-0309-y)

[7] 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.

[8] C. Bibby and I.D. Reid. Robust real-time visual tracking using pixelwise posteriors. In European Conference on Computer Vision, pages II:831–844, 2008

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

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

[11] 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

[12] Fu Chang, Chun-Jen Chen, Chi-Jen Lu. A Linear-Time Component-Labeling Algorithm using Contour Tracing Technique. Computer Vision and Understanding, V93, I2, pp 206 220. 2004.

[13] Kesheng Wu, Ekow Otoo, and Arie Shoshani. Optimizing connected component labeling algorithms. SPIE Proceedings of Medical Imaging Conference 2005, San Diego, CA, 2005.LBNL report LBNL-56864.

[14] Christophe Fiorio and Jens Gustedt. Two linear time unionfind strategies for image processing. Theor. Comput. Sci.,154(2):165–181, 1996.

[15] Jung-Me Park, Carl G. Looney, Hui-Chuan Chen. Fast Connected Component Labeling Algorithm Using a Divide and Conquer Technique. Computer Science Dept University of Alabama and University of Nevada, Reno. 2004.