Foreground-Background Separation on GPU using order based approaches

Raj Gupta[*]
Indian Institute of Technology[†]
Madras, Chennai,India
gupta.raj@gmail.com

Sailaja Reddy M
Indian Institute of Technology[‡]
Madras, Chennai, India
sailu.frns@gmail.com

Swagatika Panda
Indian Institute of Technology[§]
Madras, Chennai, India
swagatika.panda.rpnnss@gmail.com

Sushant Sharma
Indian Institute of Technology[¶]
Madras, Chennai, India
sushantsha@gmail.com

Anurag Mittal
Indian Institute of Technology
Madras, Chennai, India
amittal@cse.iitm.ac.in

ABSTRACT

Background modeling has been a challenging task in computer vision applications. Most of the approaches use the intensity space to do the background modeling. This basic assumption is not valid in the case of illumination changes. So, we have changed from the intensity space to the order space. We look on the patch (neighborhood of pixels) and build the model using the order among the pixels in it. The model built by us has more impact on the center part of the patch. Hence we have used the concept of overlapping of patches to make it more robust. The results have been shown on the standard PETS dataset as well as dataset collected by us using the camera setup in the outdoor environment. We have implemented it on GPU(NVIDIA Tesla C1060 Processor) to increase the throughput and we are able to achieve the 25X speed compared to CPU.

1. INTRODUCTION

Foreground-background separation is a very crucial step in the field of computer vision with very wide range of application area, especially in tracking and surveillance. This is essentially a preprocessing task which precedes other high level tasks such as blob detection, tracking, object detection and recognition etc. Hence efficiency and accuracy of the Foreground-background separation method implemented has a lot of significance. Here, we propose an illumination-invariant background subtraction algorithm in which every frame is divided into two sets of multiple patches. Each patch is compared to the corresponding patch of the background model in order to classify it as foreground or background. Then the overlapped region of two patches is selected as foreground if either of the patches are classified as foreground. We have further implemented our algorithm in GPU (Tesla C1060 Processor), in order to improve the performance in terms of speed and stability, and to offload the CPU at the same time.

The paper is organized as follows. Section 2 describes the algorithm for foreground-background separation. Section 3 focuses on the GPU-based implementation and section 4 gives the results on the proposed algorithm. Finally, we conclude the work in section 5.

[*]Corresponding author
[†]IBM India Research Lab, New Delhi, India
[‡]National Informatics Center, New Delhi, India
[§]International Institute of Information Technology, Hyderabad, India
[¶]PEC University of Technology, Chandigarh(UT), India

1.1 Related Work

The idea of using ranks rather than raw intensities has been studied for some time now. The Census algorithm[21] transforms the intensity space to an “order” space, where a bit pattern is formed by looking at the orders of a given pixel with its neighbors. This is quite similar to the Local Binary Patterns [12][5][15][19] approach. In order to match a block, this algorithm essentially counts the number of flipped points pairs in the block. Bhat and Nayar[6] use an improved version of this algorithm where they somewhat alleviate the problem of counting even one salt-and-pepper error in a pixel multiple times. Mittal and Ramesh[16] proposed a method in which the penalty for an order flip is proportional to the intensity difference between the two flipped pixels. This reduces the error due to pixels whose order may have got flipped due to Gaussian noise. Singh et al[17] present a statistically better approach for matching image patches.

We have looked at previous methods for foreground background separation using a range of HPC architectures including symmetric multiprocessing (SMP), massively multiprocessing (MMP) and architectures with distributed memory (DM) and non-uniform memory access (NUMA). The majority of recent research in multi-core adaptation of foreground background separation has focused on GPUs [7][9].

There are several reasons for the interest in GPUs. Thanks to fierce competition and driven by the gaming industry, GPUs today provide some of the highest performance and the lowest power consumption per FLOPS of any computing platform. GPU implementations tend to be more challenging than multi-core CPU implementations and are more rewarding in terms of achievable performance gains. The modern software platforms for general purpose programming on the GPU currently are NVIDIA's CUDA and AMD/ATI's Brook+[4][1]. While the exposure of architecture validates the argument in favor of using the graphics pipeline for general purpose programming, this has given better performance gains.

2. THE ALGORITHM

The algorithm for foreground background separation is described in the following order. The subsection 2.1 describes the matching technique algorithm to classify a particular region of the frame as foreground or background. The subsection 2.2 describes the method that the background model uses to learn and adapt dynamically.

2.1 Foreground/Background Classification

The frame is divided into a grid of blocks or patches and the following algorithm is applied for each patch of the frame in order to match it with the corresponding patch in the background model. The background model is learned adaptively as explained later in 2.2. This method is based on a stable monotonic-change-invariant feature descriptor (Refer [10] for details). This feature descriptor consists of point pairs. In order to obtain invariance to Gaussian noise, the points in a pair are chosen such that they have a certain minimum difference between their intensities. In addition to this, robustness to changes in the scale and localization of the feature point is obtained by picking point pairs such that moving the points a certain distance in their neighborhood does not change the order of the intensities of the pair. Therefore, the point pairs are relatively stable in their intensity order with respect to both intensity noise and localization error. Further, we allow a point to repeat only a certain number of times in the pairs in order to improve the independence between the different point pairs.

2.1.1 Computation of Extremal Regions

The first step in our algorithm is the computation of extremal regions. Extremal regions are regions that have intensities above or below a given threshold. Given that the points in the point pairs must have a given difference of intensity δ_I between them, we compute extremal regions with two thresholds T_1 and T_2 such that $T_1 - T_2 = \delta_I$:

$$\mathcal{R}^+ = Thresh^+(I, T_1)$$
$$\mathcal{R}^- = Thresh^-(I, T_2)$$

where $Thresh^+(I,T)$ is the set of all points in the Image I that are above a given threshold T and $Thresh^-(I,T)$ is the set of all points I below T. As should be obvious, all points in $\mathcal{R}^+$ are greater than all points in $\mathcal{R}^-$ by atleast an intensity difference of δ_I.

2.1.2 Computation of Point Pairs

Given a pair of extremal regions $\mathcal{R}^+$ and $\mathcal{R}^-$, we compute points that are as far as possible from the boundaries of these

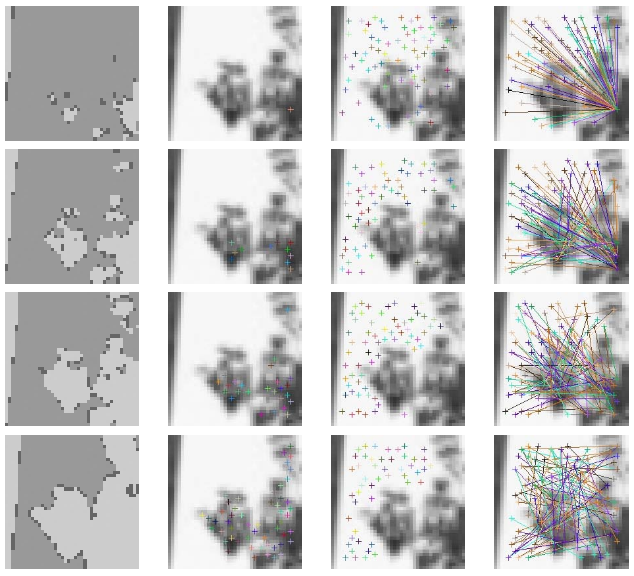
Figure 1: An Example of pair extraction at different levels. First column shows the extracted extremal regions where the brighter gray is the "Min" extremal region, darker gray is the "Max" extremal region and black are the boundary points. The second and third columns show the extraction of some points in the min and max regions respectively, super-imposed on the original patch. Finally, the last column shows the pairs formed at each level, again super-imposed on the original patch. These are combined by throwing away pairs having common points in order to obtain the final pairs shown in Fig. 2.

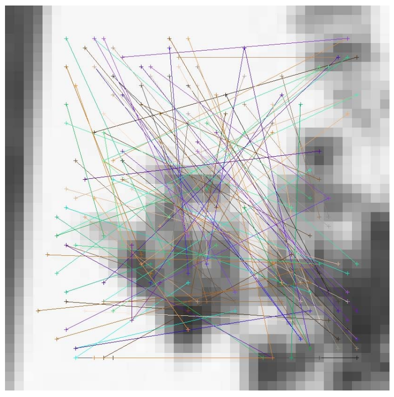
Figure 2: An Example of pairs extracted in a patch.

regions using the distance transform. We use the one based on the Euclidean distance measure[8]. Points that have high distances from the boundaries are selected as possible candidates and points from $\mathcal{R}^+$ are matched with points from $\mathcal{R}^-$. Once a pair is selected for a given region pair, we mark these points as boundary points so that the points that are selected next are those that have the largest distance not only from the original boundaries but from the already selected points as well. This procedure is repeated till we cannot obtain points that have a certain minimum "distance" from the boundaries. After obtaining the set of point pairs, we select the most stable ones based on the "distance" value associated with the points as per the distance transform. The minimum of the distance values of the two points in a pair is taken as the *stability factor* for that pair. Then, using a greedy approach, we select the most stable point-pairs one by one while taking care that any one point in the patch does not have too many close-by points (maximum 3) in the already existing point-pairs. This is done in order to ensure some independence between the different point pairs and to allow the pairs to spread out in the patch so as to obtain discriminability. The output of this procedure is a set of point pairs along with their stability factors. Features which do not produce a certain minimum number of stable pairs are discarded as being unreliable for matching. This set of extracted point pairs, thus, forms the feature descriptor for matching the patches of the two frames (current frame and the background model). Fig. 1 shows the above process on an image patch, where the different rows show the computation of the extremal regions and point pairs at different thresholds. These are combined in the end in order to obtain the final point pairs shown in Fig. 2.

2.1.3 Matching

For each of the features obtained, we have a set of point-pairs along with their stability factors $\{(p_i^1, p_i^2, s_i), i = 1 \dots n\}$. For these point pairs, we test if the order of the pixels has changed in the other patch. Then, we calculate a weighted sum of the order flips, giving each point pair a weight that depends on its stability factor. Since higher stability points are very important for stable matching and should be given a higher weight, we use the square of the stability factor (s^2) as the weight for a pair that has stability factor s. The final weighted matching score is obtained using the pairs from both the feature points as follows:

$$M = \frac{\Sigma_{i=1}^{n} s_i^2 \, \mathrm{sgn}(I_o(p_i^1) - I_o(p_i^2))}{\Sigma_{i=1}^{n} s_i^2} \tag{1}$$

where $I_o(p)$ is the intensity of point p in the patch "other" than the one in which the point pair was computed (i.e. if the pair was computed in the first patch, then I_o is the intensity in the second patch and if the pair was computed in the second patch, then I_o is the intensity in the first patch). sgn is the *sign* function:

$$\mathrm{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases} \tag{2}$$

We assume in Eq. 1 that the pair (p_i^1, p_i^2, s_i) is stored such that the first point has higher intensity than the second in the original patch in which this pair was computed.

2.1.4 Patch-Overlap Method

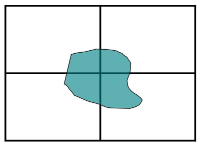

Figure 3: an object in the corner of the patches.

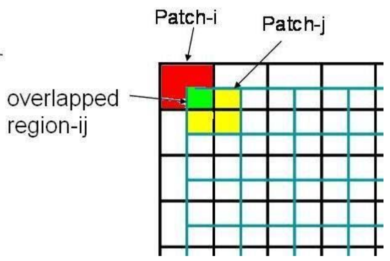

Figure 4: overlapped patches in an image.

The stable monotonic change invariant matching technique weighs points towards the center of the patch as more stable. In such a case, if any object remains in the corner of a patch as depicted in Fig. 3, there is a chance that it is ignored being a less stable point. In order to solve this problem and to achieve robustness in the background subtraction, we define another patch that covers the corner regions of four patches as shown in Fig. 4 and apply the matching technique on this overlapping patch also.

The entire frame is, at first, divided into a grid of smaller patches (say, 10x10). Then, a portion of width equal to half of the size of the patch is excluded from the boundary of the frame and rest of the area of the frame is divided into another set of patches of the same size as before. The new set of patches overlap on the first set of the patches as shown in Fig.4. The individual patches are matched with the existing background model as per the algorithm described in the previous subsections.

2.1.5 Augmenting results with patch information

The overlapped region of two patches (here, of the size 5x5 as marked by green region in Fig.4) is declared as foreground or background based on the result of matching for each of the patches. It is decided as background, if and only if both of the patches are backgrounds. If either of the patches are foreground, then the overlapped region is declared as foreground. Total stability measure of each patch is calculated as the sum of the squares of stability factors of individual pixels in the patch. If both the patches, patch$_i$ and patch$_j$ have very less "total stability measure" (2.1), i.e., the patches are homogeneous, then the average value of intensity and RGB values of the pixels in patch$_i$ of the current frame is compared with the corresponding values of the same patch in background image. If the difference exceeds certain heuristically chosen threshold, then the overlapped region$_{ij}$ is marked as "foreground". On the other hand if either of the two patches have significant total stability measure, then a higher value of sum of order flips M (Eq. 1) in either patch$_i$ or patch$_j$ marks the overlapped region$_{ij}$ as "foreground".

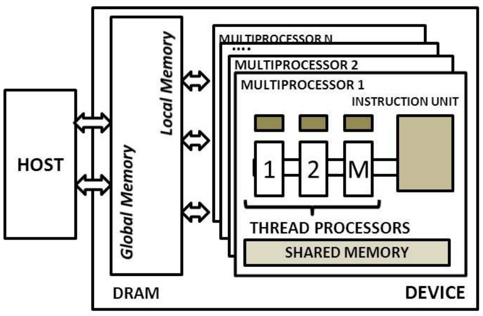

Figure 5: Physical Memory layout of Tesla Processors.

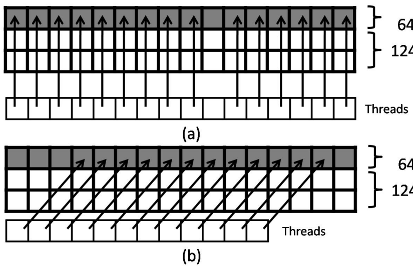

Figure 6: (a) Coleased Global memory access to avoiding redundant transactions. (b) Number of threads in the warp less than *warp-size*/2 causes redundant transactions.

2.2 Learning the Background

The background in any real-world scenario is not static; it keeps changing with time. For example, change in illumination and presence of shadows according to the time of the day, addition or removal of movable objects like vehicles which remain in the region of interest for a longer period of time, rain, clouds etc. The background subtraction algorithm that we have implemented is adaptive and keeps learning and dynamically upgrading itself with time [18]. The pixels of the background model are updated periodically as follows:

$$bk = bk * (1 - \alpha) + fg * \alpha \tag{3}$$

where, α is the learning rate, bk is the pixel intensity of the existing background model and fg is the pixel intensity of the current frame. The advantage of the method of learning and upgrading the background is that the existing model of background is not destroyed when an object is allowed to become a part of the background, if it remains stationary for sufficiently longer duration. Later, if the object starts moving, the previous background is quickly recovered.

3. GRAPHICS PROCESSING UNITS

A graphics processing unit or GPU is a specialized microprocessor that offloads and accelerates 3D or 2D graphics rendering. NVIDIA's Tesla Architecture exposes the computational Horse power of the NVIDIA's GPU. Here, we present an overview of the NVIDIA GPU's hardware. GPUs are designed such that more transistors are devoted to data processing rather than data caching and flow control. GPU is well-suited for problems expressed as data-parallel computations with high arithmetic intensity[1]. Fig 5 shows the general physical layout of NVIDIA GPUs. The Device has its own Global Memory, which all the cores (Thread processors) can access. It contains N multiprocessors which have M cores each. Cores share an instruction unit with other cores in a multiprocessor. Each processor has its own local memory, separate register set and all the M cores share an on-chip memory called shared memory. NVIDIA Tesla C1060[2] follows 10 series NVIDIA architecture and has 30 multiprocessors. Each multiprocessor has 8 cores, a double precision unit and an on-chip shared memory.

[1]Arithmetic Intensity is defined as the ratio of arithmetic operations to memory operations

3.1 CUDA: a General-Purpose Parallel Computing Architecture

NVIDIA GPUs can be interfaced by CUDA, developed by NVIDIA Corporation. CUDA is a scalable heterogeneous serial-parallel programming model and a software environment for parallel computing on multicore CPUs and GPUs.

3.2 Performance Optimization

We use the following performance optimization[13] techniques to overcome possible bottlenecks:

3.2.1 Maximizing parallel execution

Our aim is to increase the fraction of parallel code and hence increase the speed-up of the implementation. We parallelize the computation to an extent to keep all GPU cores busy.

3.2.2 Optimizing memory transfer

Device to Host memory bandwidth is much lower than Device to Device memory bandwidth[2]. Hence, it is important to minimize data transfer, even if that means running code on GPU that does not demonstrate any relative speed-up. Large transfer is preferred over many small ones. To hide the transfer delays we have used concurrent kernel execution[4].

3.2.3 Optimizing memory usage

Global Memory Read/Write has highest latency (400-600 clock cycles) which is likely to be a performance bottleneck. A multiprocessor partitions threads into warps[3]. When a half-warp[4] accesses a contiguous region in the global memory, the 16 individual transfers are combined into a single transfer(fig 6.a). Note that even though one word is not requested, all data in the segment are fetched in a 64 byte transaction. Therefore the number of threads per block should be in multiple of 16 whenever possible and the data structure to be accessed should be designed so as to encourage coalesced memory access[3] and reduce redundant data transferred(fig 6.b).

Each Multiprocessor has a Shared Memory associated with it. Being on-chip, it is much faster than local and global

[2]Host⇔Device tranfers over PCI-ex16(4GB/s) vs Device⇔Device (80GB/s) for C1060

[3]A warp is a group of 32 threads executed on a multiprocessor

[4]Half-warp can be first half or the second half of a warp

memory. Moreover it allows cooperation between a restricted group of parallel threads (block), i.e., sharing memory accesses and sharing results to avoid redundant computation. The image/patch data should first be loaded into the shared memory and then processed upon if that reduces the global memory access.

3.2.4 *Optimizing instruction usage*

Multiprocessors have a single instruction set (Refer Section 3) and employs SIMT[5] architecture. A warp executes one common instruction at a time. Any flow control instruction (if, switch, do, for, while) may cause threads of the same warp to diverge, making it serialized until they converge, increasing the total number of instructions executed for the warp. Hence, threads of a warp should agree on their execution path as far as possible.

3.3 Implementation

Many foreground-background algorithms work on blocks of pixels (patches). The following approach can be used to implement them on GPU, parallelizing w.r.t. the blocks and gaining significant speed up. Our implementation is described in the following order. The subsection 3.3.1 explains how CUDA scales on GPUs and introduces some related terminologies. Subsection 3.3.2 describes the program flow of the foreground background separation algorithm on GPU and sections 3.3.4-3.3.7 explain how they are parallelized.

3.3.1 *CUDA enabled on GPU*

CUDA allows the programmer to define kernels[6]. CUDA threads are grouped in blocks organized into a one dimensional or two-dimensional grid. The CUDA architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel-grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

3.3.2 *Program Flow of the Implementation*

Figure 7 gives an overview of the program flow of our GPU implementation. The different kernels are *Conversion to grey*, *image to patch*, *Create feature*, *calculate measure*, *create binary image*. *Conversion to grey* converts the input images into grey scale, *image to patch* creates the overlap image and categorize both the original and overlap image into patches(Section2.1.4), *Create feature* does the computation of the Extremal regions(Section 2.1.1) and point pairs (Section 2.1.2), *calculate measure* calculates the total stability measure and does the matching(Section 2.1.3), *create binary image* creates binary image for the overlap regions.

The configuration arguments (grid-size,block-size) are shown as <grid, size>. Each kernel reads image/patch information from global device memory space and writes the result back into the same. Threads in a block may make use of the shared memory for the intermediate storage and cooperation (Refer Section 3.2.3). A copy command represents Host ⇔ Device memory transfers. Red arrows depict concurrent transfer and kernel execution. Synchronize command is used to make sure all the data transfer and kernel execution is synchronized before further execution.

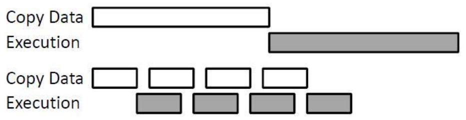

Figure 8: Timeline for a non-concurrent and a concurrent execution. No kernel will launched until the data transfers in their respective streams complete

3.3.3 *Representation of the Image Structure*

The image/patch information is represented as a structure of arrays (SoA) and not as array of structures (AoS). The pictorial representation is shown in the fig 7. This representation allows efficient memory coalescing (Refer Section 3.2.3) as the data required by the threads, executing a common instruction, will be lying in contigious memory locations.

3.3.4 *Achieving Concurrency in execution and data transfer*

Streams are used to have concurrency between the kernel execution and Host ⇔ Device Asynchronous data transfer[4]. A stream is a sequence of commands that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently. The kernel for *conversion to grey* uses streams and achieves concurrent transfer with execution. Number of stream objects created is W_i, where W_i is the width of the image in pixels. Each stream object executes a kernel which runs for H_i number of threads, where H_i is the height of the image in pixels. The following sample shows use of streams to achieves concurrent transfer with execution of *conversion to grey*.

```
**for** $i = 0$ to $W_i$ **do**
create(stream[i]);
$copy(Devicepointer, Hostpointer, size, stream[i])$;
specify sequence of kernel launches each with H$_i$ threads
destroy(stream[i]);
**end for**
```

Fig 8 compares the timeline of a non-concurrent and a concurrent execution respectively. Note that none of the kernels will launch until the data transfers in their respective streams complete.

3.3.5 *Gathering Patch Information*

Next task is to categorize image information according to patches. For this we can parallelize w.r.t to the pixels in the respective patch. We launch the kernel grid as 2-D group (of blocks) of size ($W_i/W_p, H_i/W_p$) corresponding to each patch and we declare each block as a 2-D group (of threads) of size (W_p, W_p) corresponding to each pixel in the patch. Here W_p corresponds to the size of the patch in pixels. The figure 9 depicts the Thread batching process. Each parallel thread makes coalesced access[7] to the pixel

[5]Single-Instruction Multiple-Thread

[6]Kernels are C functions that, when called, are executed N times in parallel by N different CUDA threads

[7]Here coleased access is achieved by choosing W_p as 8 or 16

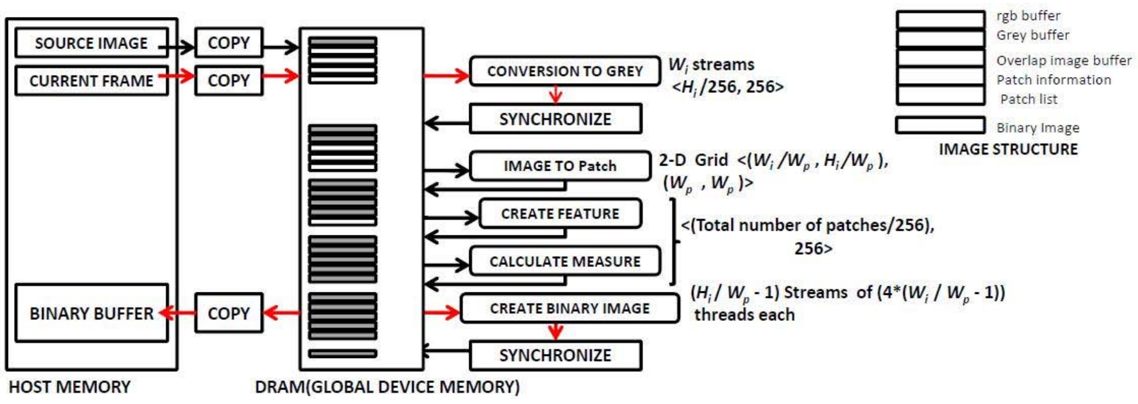

Figure 7: An overview of the memory management and program flow on the GPU.

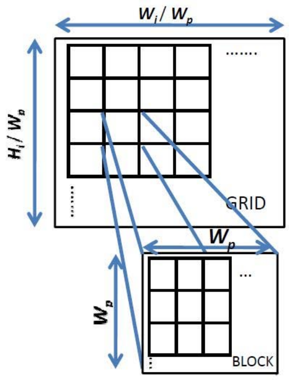

Figure 9: Thread Batching. Both Grid and Block are declared as 2-D.

information (rgb buffer and the grey buffer arrays in the global memory). It is followed by a call to kernel which spawns $(H_i - W_p)$ number of threads, each copies the rgb and grey information of $(W_i - W_p)$ overlapping pixels. This is a better alternative for Device ⇔ Device transfer than to call the copy command from the Host code with the cores idle. The patch information for the overlapped image is collected in the same fashion as did for the original image.

3.3.6 *Computation of extremal regions, point pairs and Matching*

Now we have information of patches for the original image and the overlapped image. The total number of patches is

$$(\frac{W_i}{W_p} * \frac{H_i}{W_p}) + ((\frac{W_i}{W_p} - 1) * (\frac{H_i}{W_p} - 1))$$

For the computation of the Extremal regions (Section 2.1.1), Point pairs (Section 2.1.2) and then the Matching (Section 2.1.3), the processing is done for each patch which is independent of the other patches. Therefore we can parallelize the algorithm w.r.t. the patches. We declare blocks, each of 256 threads as the optimal value and each thread processes one patch.

(Refer Section 3.2.3)

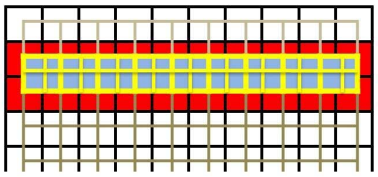

Figure 10: Parallelizing over overlapped regions. Each thread in a block process for one respective region (yellow).

3.3.7 *Creating Binary Image*

In order to create the binary image, we have to consider the overlapped regions as shown in green in figure 4. There are $4*(H_i/W_p-1)*(W_i/W_p-1)$ overlapped regions. We can parallelize w.r.t. these regions, i.e., we can use each thread corresponding to each overlapped region. As every region is an overlap of two patches, we have every thread accessing information of two patches. We have one patch shared by at most 4 regions, in other words, we can have at most 4 threads accessing the same patch-information(memory location) but they all are writing on different memory locations(in the binary image buffer) therefore making the algorithm parallelizable. Moreover as the accesses are shared, its profitable to make use of the *shared memory.* Let us declare the grid of $(H_i/W_p - 1)$ blocks with $(4 * (W_i/W_p - 1))$ threads each. Each thread in a block processes for one region (shown in yellow in Fig.10). For each block of threads we need information about $(2 * W_i/W_p)$ patches of the original image(shown in red) and $(W_i/W_p - 1)$ patches of the overlapping image (shown in blue in Fig.12). This data is moved to the shared memory which avoids re-accessing the global memory for other overlapped regions, reducing global memory accesses to around half. With the help of *streams* and *paged locked* host memory, we can concurrently copy the binary image data from the device to the host. Moreover CUDA allows us to do heterogeneous programming making it possible to read a new frame into the Host memory. Figure 12 shows the timeline for the concurrent asynchronous copying compared to serial execution and copying.

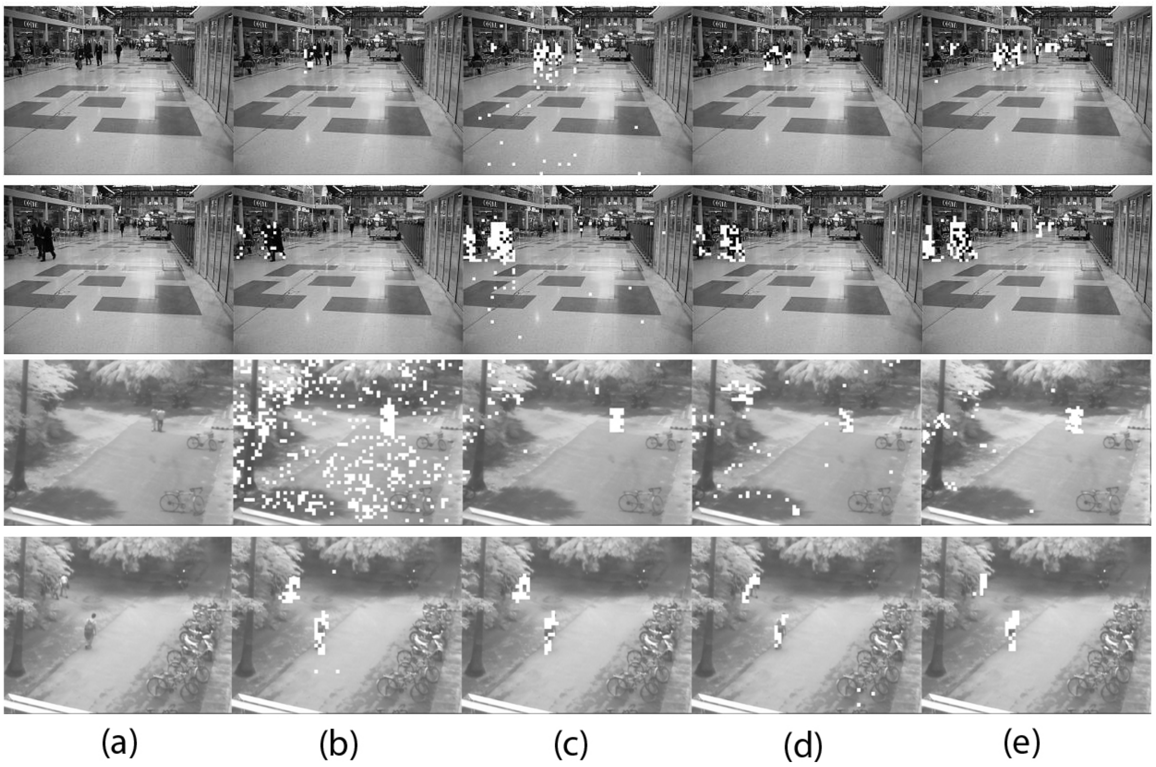

Figure 11: Results for foreground-background separation. Row1, Row2: indoor scene on the PETS database. Row3: outdoor scene on a sunny day with presence of shadows and leaf-movements. Row4: outdoor scene on a cloudy day with less illumination. (a)Input Image, background subtraction using (b) LTP technique (c)LTP technique (d) using GMM technique (e)using monotonic change invariant method

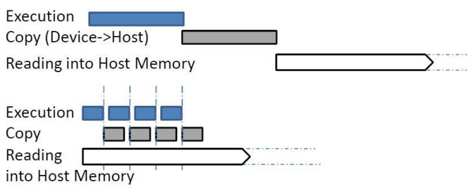

Figure 12: Timeline for serial execution and *copy* compared to concurrent asynchronous *copy*.

4. RESULTS

The algorithm was successfully implemented in the indoor as well as outdoor environment where it gave significant performance in various conditions such as bright scenes, scenes with less illumination, moving shadows of the trees, fluctuations in leaves and branches etc. The first row and 2nd row in Fig11 show results taken from PETS dataset in the indoor environment. The third row shows results of different background subtraction techniques on a sunny day with significant shadows and leaf-movements. The 4th row shows the results on a cloudy day with less illumination and less shadows. The result was appreciable when compared to various other techniques such as LTP technique [11], LBP technique [14] [20] and GMM [18] as shown in Fig11.

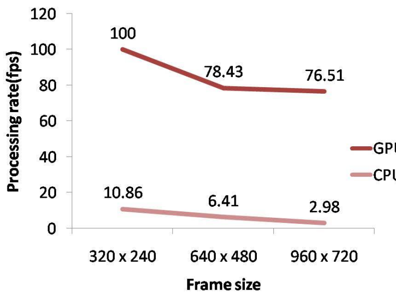

Figure 13: Processing rate of GPU compared to CPU

4.1 Performance Gains

The performance of the GPU implementation was evaluated on a 2.00 GHz Intel(R) Xeon(R) running MS XP Pro x64 with an NVIDIA Tesla C1060 GPU. An equivalent CPU version of the algorithm was implemented on the same system. The two implementations were tested using a variety of video frame sizes, and a summary of typical results is

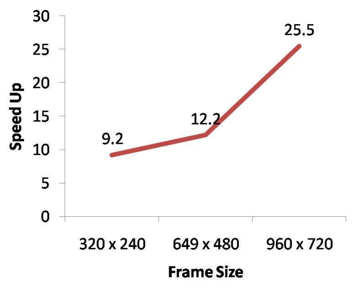

Figure 14: Performance gain from GPU implementation.

presented in Fig 13 and Fig 14. Although the parallelism based on patches and overlapped regions are well defined, many stages have branching conditions. Such aspects will hinder the performance of parallel execution. In this work, we made every effort to minimize the complexity of branching conditions making them more predictable. Fig 14 shows that performance gain increases with increase in size of the image. This is because for smaller images, the number of patches/regions are not large enough to keep the multiprocessors busy, in other words the occupancy[8] is less and the time taken to launch the kernels is not justified to a great extent. With the increase in the size of the image, fraction of parallel code increases thus increasing the speed up (Refer section 3.2.1).

5. CONCLUSION

In this paper, we described the foreground background separation method using the order of intensities. This approach is also robust towards Gaussian noise and distortion in spacial domain of pixels. It can be implemented quite efficiently using the fast algorithms that are available for extremal region and distance transform computation. It is effective in various cases of fluctuations in background such as shadows, weather change (clouds) and other features of real world. This algorithm also provides promising results in both indoor and outdoor environments. Since foreground-background separation is used as a pre-processing stage in many applications like tracking, object recognition etc., we have implemented the foreground-background separation on a GPU to achieve high throughput. We have achieved the speed up of 25X for 960 X 720 image resolution using GPU (NVIDIA Tesla C1060 Processor).

6. REFERENCES

[1] *ATI stream computing user guide, version 1.4.0.a.* AMD, 2009.

[2] *Board Specification- Tesla C1060 Computing Processor Board January 2009.* NVIDIA, 2009.

[3] *NVIDIA CUDA Best Practices Guide Version 3.0 2/4/2010.* NVIDIA, 2010.

[4] *NVIDIA Programming Guide Version 3.0 2/20/2010.* NVIDIA, 2010.

[5] T. Ahonen, A. Hadid, and M. Pietikainen. Face recognition with local binary patterns. In *Computer Vision, ECCV 2004, Lecture Notes in Computer Science.*

[6] D. Bhat and S. Nayar. Ordinal measures for image correspondence. In *PAMI.*

[7] P. Carr. Gpu accelerated multimodal background subtraction. In *DICTA '08: Proceedings of the 2008 Digital Image Computing: Techniques and Applications*, pages 279–286, Washington, DC, USA, 2008. IEEE Computer Society.

[8] R. Fabbri, L. da Fontoura Costa, J. C. Torelli, and O. M. Bruno. 2d euclidean distance transform algorithms: A comparative survey. *ACM Computer Survey*, 40(1):1–44, February 2008.

[9] Griesser, A. D. Roeck, S. Neubeck, A. V. Gool, and Luc. Gpu-based foreground-background segmentation using an extended colinearity criterion. In *10th international fall workshop on VMV*, pages 319–326, Erlangen, Germany, 2005. Proceedings 10th international fall workshop on VMV.

[10] R. Gupta and A. Mittal. Smd: A locally stable monotonic change invariant feature descriptor. In *Computer Vision - ECCV 2008.*

[11] R. Gupta, H. Patil, and A. Mittal. Robust order-based methods for feature description. In *CVPR*, June 2010.

[12] A. Hadid, M. Pietikäinen, and T. Ahonen. A discriminative feature space for detecting and recognizing faces. In *CVPR.*

[13] M. Harris. *NVIDIA Optimizing CUDA.* NVIDIA Developer Technology, 2008.

[14] M. Heikkila, M. Pietikainen, and J. Heikkila. A texture-based method for detecting moving objects. In *British Machine Vision Conference*, 2004.

[15] G. Heusch, Y. Rodriguez, and S. Marcel. Local binary patterns as an image preprocessing for face authentication. In *FGR.*

[16] A. Mittal and V. Ramesh. An intensity-augmented ordinal measure for visual correspondence. In *CVPR.*

[17] M. Singh, V. Parameswaran, and V. Ramesh. Order consistent change detection via fast statistical significance testing. In *CVPR.*

[18] C. Stauffer and W. Grimson. Adaptive background mixture models for real-time tracking. In *CVPR*, pages II: 246–252, June 1999.

[19] V. Takala, T. Ahonen, and M. Pietikainen. Block-based methods for image retrieval using local binary patterns. In *SCIA.*

[20] M. Yao and J. M. Odobez. Multi-layer background subtraction based on color and texture. In *CVPR*, 2007.

[21] R. Zabih and J. Woodfill. Non-parametric local transforms for computing visual correspondence. In *ECCV.*

[8]Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps