Thesis

A Real Time Video Pipeline for Computer Vision Using Embedded GPUs

Submitted by

Rutuja Patil

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2016

Abstract

A Real Time Video Pipeline for Computer Vision Using Embedded GPUs

This thesis presents case study confirming the feasibility of real time Computer Vision applications on embedded GPUs. Applications that depend on video processing, such as security surveillance, can benefit from applying optimizations common in scientific computing. This thesis demonstrates the benefit of applying such optimizations to real time Computer Vision applications on embedded GPUs. The primary contribution of this thesis is an optimized implementation of ViBe targeting NVIDIA's Jetson TK1.

ViBe is a commonly used background subtraction algorithm. Optimizing a background subtraction algorithm accelerates the task of reducing the field of view to only interesting patches of the frames of the video. Placing portable hardware close to capturing devices in the surveillance system reduces bandwidth requirements and cost. The goals of the optimizations proposed for this algorithm are to 1) reduce memory traffic 2) overlap CPU and GPU usage 3) reduce kernel overhead.

The optimized implementation of ViBe achieves a frame rate of almost 55 FPS beating the real time goal standard of 30 FPS for real time video. This is a small portion of the real-time window leaving processing time for additional algorithms like object recognition.

Acknowledgements

I feel relieved and proud to have completed my thesis titled 'A Real Time Video Pipeline For Computer Vision Using Embedded GPUs.' I would like to thank God for giving me the strength to endure till the end. I would like to dedicate this thesis to my family and friends for their constant love and motivation.

This feat would not have been possible without the guidance of Dr. Catherine Olschanowsky. She truly deserves to be credited for being so patient and motivating all through the journey. I would also like to thank Dr. Ross Beveridge for agreeing to be my advisor and giving me all the timely help when needed. Dr. Bruce Draper deserves special mention for volunteering to mentor me inspite of not being on my committee. I would like to thank Dr. Ross Beveridge and Dr. Bruce Draper for allowing me to be a part of the Vision Group. Lastly, I would like to thank my committee members Dr. Mahmood Azimi and Dr. Stephen Guzik for their insightful contribution to this work.

Table of Contents

Abstract ii

Acknowledgements iii

List of Figures vi

Chapter 1. Introduction 1

1.1. Thesis Goals 2

1.2. Contributions 2

Chapter 2. Background 4

2.1. ViBe Overview 4

2.2. Jetson TK1 9

Chapter 3. A Minimal Image Processing Kernel Optimized For Unified Memory GPUs 11

3.1. RGB to Grayscale Conversion 12

3.2. Scheduling 13

3.3. Experimental Results 15

3.4. Conclusion 19

Chapter 4. ViBe: Optimized for the Jetson TK1 20

4.1. Verification Procedure 21

4.2. Baseline Comparisons 28

4.3. ViBe:CUDA Implementation 29

Chapter 5. Related Work 38

5.1. Background Subtraction Algorithms 38

5.2. Performance Comparison 39

Chapter 6. Conclusion and Future Work 43

6.1. Conclusion 43

6.2. Future Work 43

Bibliography 45

Appendix A. Verification of Naive Parallel 47

Appendix B. Verification of Naive Parallel Overlap 48

Appendix C. Verification of Naive Parallel Blur 49

Appendix D. Verification of Fused Parallel Blur Overlap 50

List of Figures

1.1 The full video frame is reduced to a set of boxes bounding only the interesting portions. In this case the foreground comprises moving vehicles. 1

1.2 The setup To optimize the first step of the video pipeline 2

2.1 The 2D space represents the area of a single frame of video. At each location a history of 20 pixel values is stored. This collection of 20 values is the model used by ViBe 5

2.2 Euclidean space model for the incoming pixel. The R, G, B axis represent the coordinates for this space. 6

2.3 The Jetson TK1 and its peripherals [elinux.org JetsonTK1] 9

3.1 a) For Loop for 2D iteration space on CPU b) Visual representation of 2D iteration space c) 2D iteration space on GPU 13

3.2 2D iteration space divided into blocks on a GPU. 14

3.3 Iteration space for 2D blocking for image size 720x1280. 15

3.4 Block configuration sweep comparisons for blocked float RGBtoGRAY. The conclusion from this graph is that performance improves with increase in block size. 17

3.5 Block configuration sweep comparisons for blocked integer RGBtoGRAY. The conclusion from this graph is that performance improves with increase in block size. 17

3.6 Block configuration sweep comparisons for blocked float RGBtoGRAY zoomed on the interesting regions cropped from Figure 3.4. This graph highlights that the shape of the block does not affect the performance of the benchmark. 18

3.7 Block configuration sweep comparisons for blocked integer RGBtoGRAY zoomed on the interesting regions cropped from Figure 3.5. This graph highlights that the shape of the block does not affect the performance of the benchmark. 18

4.1 Foreground counter map. 21

4.2 Histogram of differences. 22

4.3 Test frame 350 of a video dataset. 23

4.4 Foreground counter map for first run of original implementation 25

4.5 Foreground counter map for second run of original implementation 26

4.6 Difference map for original ViBe 26

4.7 Foreground counter map for one run of our implementation of ViBe 26

4.8 Difference map for our ViBe Vs Original 27

4.9 a)Histograms of differences for original ViBe b) Histogram of differences for original Vs our serial implementation. 28

4.10 Serial flow of ViBe on the CPU 29

4.11 Naive parallel flow of ViBe on the CPU and GPU 30

4.12 Overlap between functions of ViBe on the CPU and the GPU 32

4.13 Transfer of median blur of ViBe to the GPU and the flow of algorithm between GPU and CPU. 34

4.14 Kernel fusion of segment and update functions of ViBe on the GPU 36

CHAPTER 1

Introduction

This thesis presents optimizations for real time Computer Vision applications on embedded GPUs. Real time video processing is a key requirement for developments such as surveillance monitoring. Real time video processing, especially in high definition, requires highly optimized and efficient code. This thesis optimizes the first stage of a Computer Vision pipeline. This work demonstrates the impact of kernel fusion on a GPU when applied to a commonly used algorithm called ViBe [3].

ViBe detects motion by segmenting the foreground from the background for every pixel of each frame of a video. ViBe is widely used and has been cited by 646 academic publications.[1] Background subtraction is an important first step that limits the field of view and focuses attention to only the interesting regions in a frame. The interesting regions can be further analysed to perform more complex tasks like motion recognition, object detection, and motion tracking. One example of a real world video analysis pipeline is security and surveillance monitoring. Figure 1.1 illustrates the flow of data through such a pipeline.

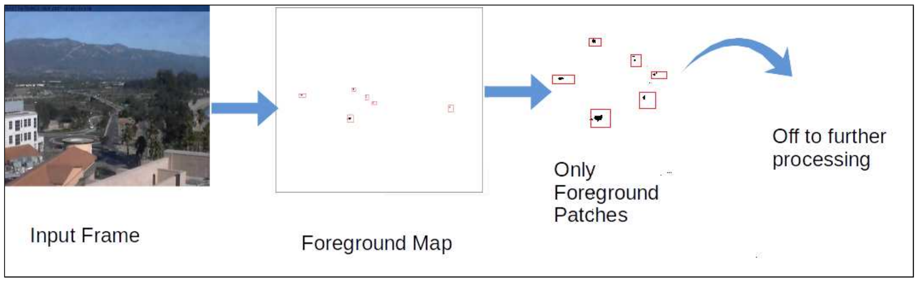


Figure 1.1. The full video frame is reduced to a set of boxes bounding only the interesting portions. In this case the foreground comprises moving vehicles.

Performing the background subtraction step at or near the camera reduces the bandwidth demands on the network supporting a surveillance system. Typically, multiple cameras

[1]According to Google Scholar dated 07/13/2016

transmit data to a central monitoring and processing center of a surveillance system. The video transmission over the network has bandwidth limitations. Therefore it is advantageous to put a small inexpensive processing device close to the capturing device. Pairing a small device with a camera can reduce network infrastructure cost and improve performance. We target NVIDIA's developed Jetson TK1, a low cost, ARM CPU and GPU integrated memory board, see Figure 1.2.

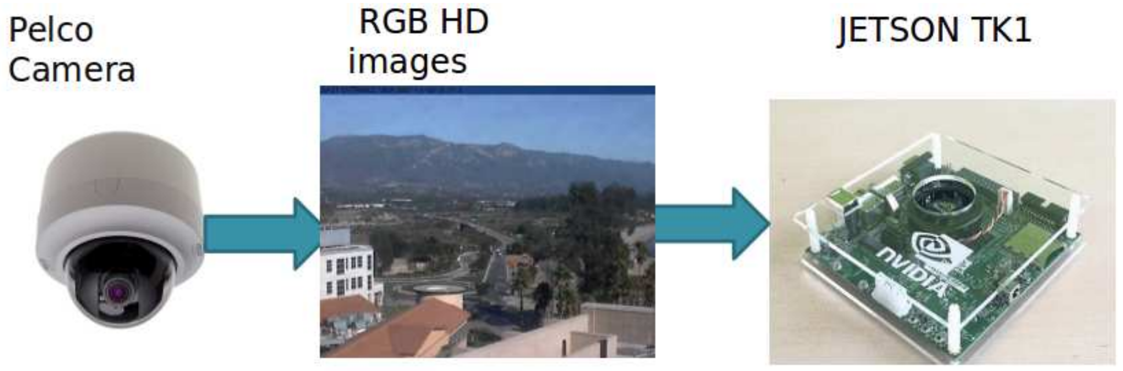


FIGURE 1.2. The setup To optimize the first step of the video pipeline

1.1. Thesis Goals

The main goal of this work is to demonstrate the possibility of real time video processing on small inexpensive hardware in order to build a starting step of a Computer Vision pipeline. A common background subtraction algorithm is optimized as a case study. The optimization techniques applied are typically utilized within the field of scientific computing[11][6].

1.2. Contributions

A small image processing benchmark provides initial evidence that real-time processing, in high definition, is possible. This minimal benchmark was used to explore alternative data decomposition approaches and study their impact on performance. A simplified serial version of ViBe was used as a starting point for serial GPU-capable implementations. Each implementation presents a different combination of optimizations. The goals of the optimizations are: to, 1) reduce memory traffic 2) overlap CPU and GPU usage 3) reduce kernel overhead. In summary, the contributions of this thesis are as follows:

(1) A minimal real time image processing benchmark.

(2) An exploration of data decomposition plans.

(3) A GPU implementation of ViBe.

(4) An exploration of the optimization space for this algorithm.

The rest of this thesis is organized as follows. Chapter 2 gives a background description of the ViBe algorithm and also presents the technical specifications of the Jetson TK1. Chapter 3 explains the performance of a simple proof-of-concept benchmark algorithm executed on the Jetson TK1. Chapter 4 lists the optimization experiments performed on ViBe and the results obtained. Chapter 5 provides an overview of the related work done in the field of optimization of ViBE and other related background subtraction algorithms. Chapter 6 states the conclusions and proposes the scope of future work in this domain.

CHAPTER 2

Background

This chapter describes the ViBe background subtraction algorithm and the targeted hardware. It intents to describe in detail each stage of the algorithm in order to understand the data structures used for this algorithm. This will help to understand the nature of the optimizations and the sections of the algorithm targeted for optimization in the future chapters. Also the details about the target hardware highlights the special features useful in order to achieve our problem statement.

2.1. ViBe Overview

ViBe is a background subtraction algorithm developed and patented by M.Van Droogenbroeck and O. Barnich in 2009 [3]. ViBe segments each pixel of each frame of a video into background and foreground. The implementation of ViBe has changed significantly since the original publication. We obtained the most recent source code directly from the authors in 2016.[1] The following description reflects the ViBe algorithm as implemented.

Background Subtraction algorithms depend on a background model used to decide the mapping of foreground and background pixels in the frames of the video. This algorithm consist of three stages that operate on the underlying model 1) Model Initialization 2) Segmentation of Frame 3) Model Update. The following describes the underlying model followed by segmentation, update and finally describes the model initialization strategy.

[1]This code is provided after performing bug-fixes on the code released in 2014.

2.1.1. The Model

The model is a history of values for each pixel from previous frames of a video. A pixel is the intensity value at a particular location of the frame of a video. The definition of a pixel varies according to the colour space. For the sake of this thesis we work in the RGB colour space. Hence each pixel is a three valued data structure of red, green and blue intensity values. Figure 2.1 is the illustration of the model data structure.

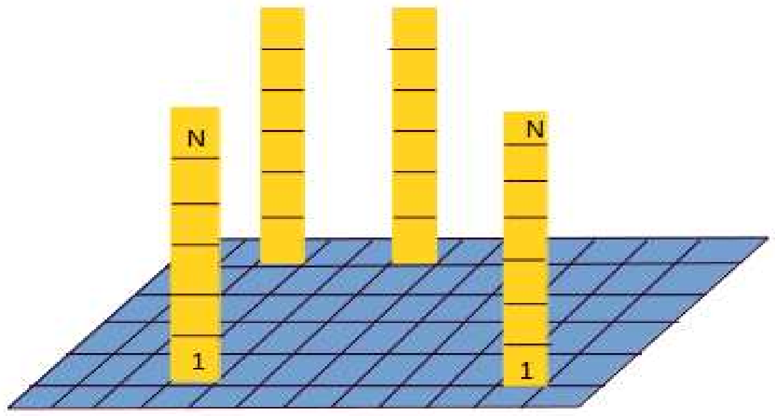


Figure 2.1. The 2D space represents the area of a single frame of video. At each location a history of 20 pixel values is stored. This collection of 20 values is the model used by ViBe

Referring to Figure 2.1, we can imagine the model to be a 3D data structure. The 2D rectangular box is the image space and the square grids on it define the pixel data structure of red, green and blue values. Each of these pixel value data structures has a collection of N pixel values represented by the rectangular box in the third dimension. The value of N for this work is 20. Twenty pixel values are stored for each location in a frame.

2.1.2. Segment Frames

The model is used as a guide to segment each pixel of a frame in a video into foreground or background. Segmentation is the part of the algorithm that makes this decision. The decision is based on the distance between the current pixel value and those in the history. This distance calculation is based upon the euclidean space. Consider a specific pixel to be placed at the center of a sphere of radius R in Euclidean space. The model values associated with the current pixel are plotted in this space. Figure 2.2 illustrates the Euclidean space model for the current pixel.

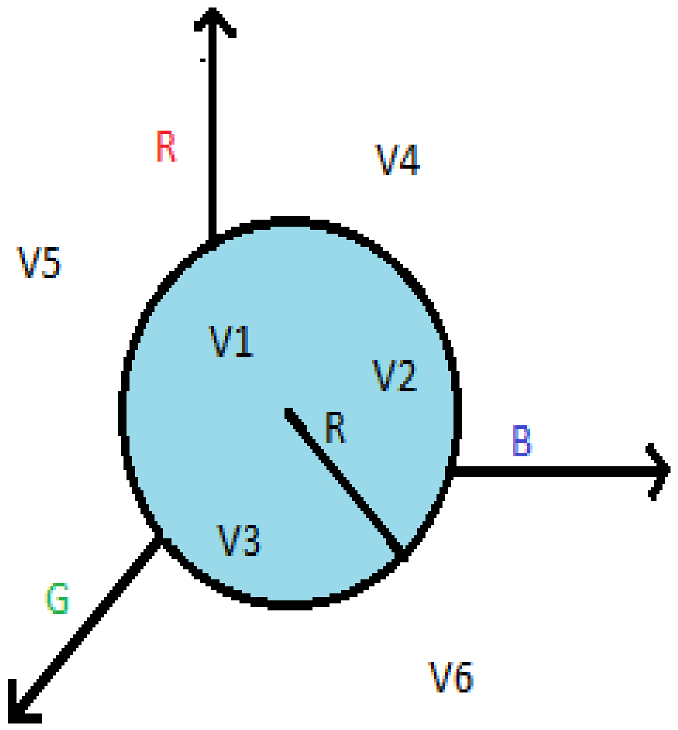


Figure 2.2. Euclidean space model for the incoming pixel. The R, G, B axis represent the coordinates for this space.

If a model value lies within the sphere of radius R, then it is counted. If the count exceeds a certain threshold, then we segment the center pixel as background, otherwise we call it foreground. The value of the count threshold is 2 in our case. Therefore, if the number of history values within a distance R from the center pixel is two or more then the center pixel is labeled as background.

2.1.3. Model Update

The model for every pixel has to evolve according to the changing content of the video. The update function is responsible for this gradual evolution of the model and takes place directly after segmentation. Model updates are only applied to the model of pixels in the incoming frame that have been classified as *background* by segmentation. Thus the update scheme of ViBe is conservative allowing the model to be populated with only background pixel intensity values.

Background pixels are randomly selected for update with a sampling rate. We have used a sampling rate of 1/16 for this work. This is the rate used by the current ViBe implementation. When a background pixel is selected for update one of the 20 history values is replaced with the current pixel value. Additionally, a history value from a randomly selected neighbour is also updated. The update scheme is such that it can effectively handle the elimination of ghost regions from the frame.

The ghost region can be explained in three scenarios. In the first case it is a region that has recently been abandoned by a static object. In this case it should be labelled as foreground by the algorithm. In the second scenario it is a foreground region that has come to rest. The algorithm should slowly be able to stop calling it foreground and label it is as background. The last scenario is that of a background region occluded by a moving object.

In this case the algorithm should be able to label that region as foreground for a short period of time and switch back to its original label once the moving object has passed. A car that stops is an example of a ghost region.

2.1.4. Model Initialization

The model initialization step is the first step of the algorithm. It happens only for the first frame. As mentioned earlier, the collection of 20 pixel values in a model associated with each pixel location in the image are typically drawn from previous frames. The first frame does not have a previous frame to draw from. The goal of model initialization is to provide an initial population for the model data structure.

For each incoming pixel of the first frame of a video, a random distribution of 20 values is selected from within a range of +/- 10 intensity values around it. For example a pixel with intensity values (245,245,10) gives us a colour in the colour palette of cyan. If this is one of the incoming pixels for the first frame, the associated model would be populated with values randomly selected from the range(235,235,0) to (255,255,20).

2.1.5. Median Blur

Median Blur is another function added by the creators of original ViBe. This is a post processing operation performed on the output map for each frame.

Median blur is a smoothing operation on an image. In this filtering operation each pixel is replaced by the median value from its 8 connected neighbourhood. This operation is useful to remove noise in the image.

2.2. Jetson TK1

NVIDIA's Jetson TK1 is a small inexpensive, power efficient board with an ARM CPU and companion GPU. It was launched by NVIDIA in March 2014. It is 5" X 5" in dimension and can hence be easily mounted on devices. It's average power dissipation is 5 Watts. The cost of this device is around $200.

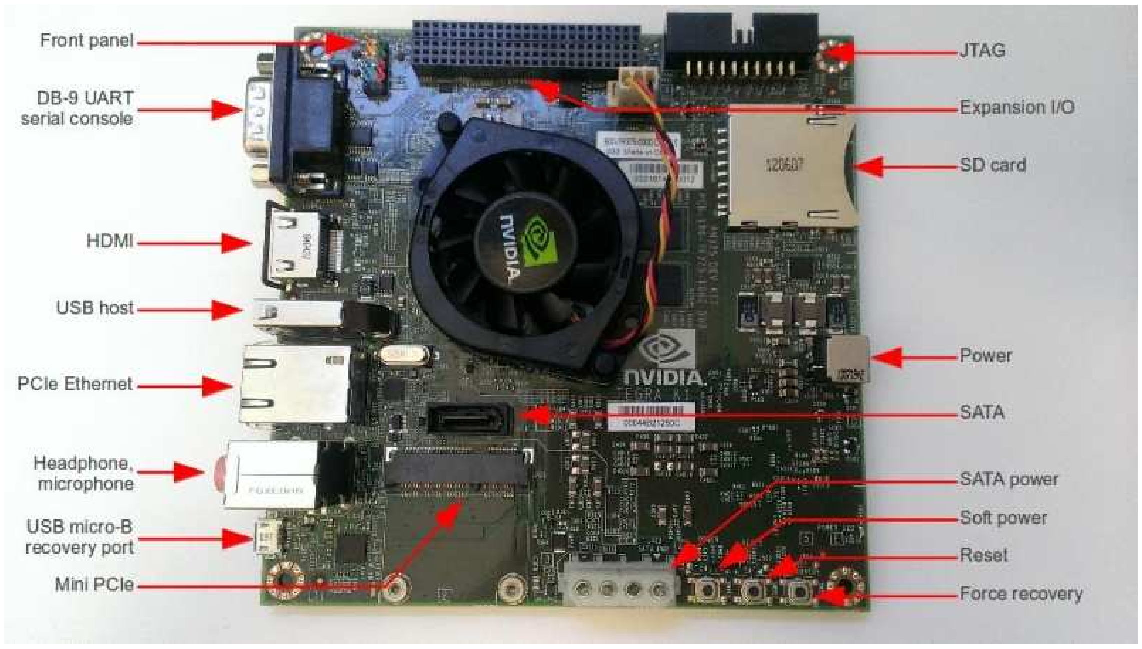


FIGURE 2.3. The Jetson TK1 and its peripherals [elinux.org JetsonTK1]

2.2.1. Technical Specifications

The Jetson can operate as a Linux platform and is pre installed with LinuxTegra4 OS which is the Ubuntu 14.04 distribution with some pre-configured drivers. The Jetson has a CPU + GPU + ISP (In System Programming) in a single chip. It has a 2.32 GHz ARM Cortex-A15 quad core processor, an NVIDIA Kepler GPU with 192 SM3.2 CUDA cores, 2 GB of DRAM, DDR3L 993 MHz and flash memory of 16 GB.

The other hardware peripherals on the board include a HDMI port, USB 2.0 port, USB 3.0 port, a RS232 serial port, a SATA data port and a Realtek GigE LAN. Figure 2.3, shows the picture of the Jetson TK1 board.

Table 2.1 highlights a few GPU specifications of the Jetson TK1

Table 2.1. GPU specifications of the Jetson TK1

GPU Clock Rate	852 Mhz
Memory Clock Rate	924 Mhz
L1 cache	32 KB
L2 cache size	128 KB
Total amount of shared memory per block	48 KB
Maximum number of threads per block	1024
Integrated GPU sharing host memory	Yes
Memory bandwidth	170 GB/s

2.2.2. Unified Memory

The CPU and the GPU memory is shared in the Jetson TK1. Hence a memory transfer over the Jetson involves shuffling physical memory addresses over the address space. This almost eliminates the memory transfer time from the CPU to the GPU and vice versa. This further supports our objective of achieving real time performance by reducing memory transfer times.

CHAPTER 3

A Minimal Image Processing Kernel Optimized For Unified Memory GPUs

The high-level goal of this work is real time processing of a HD video using inexpensive, parallel hardware. The ViBe algorithm involves mathematical operations and various read and write operations to memory complicating performance analysis. In order to easily reason about the potential performance bottlenecks a simpler benchmark was implemented and its performance evaluated using the target hardware. The evaluation focused on data movement, the time required to transfer data from CPU to GPU memory, into the registers for operation and back again. This was done to verify that the transfer of data was fast enough to support the data flow rates required by streaming HD video. It allowed us to measure the time left in a real time window to accommodate more complex operations.

The benchmark converts an HD image from RGB to gray scale. Two versions of the benchmark were implemented; float and integer versions. A framerate of nearly 63 FPS was achieved for the best performing variant between the two mentioned above with the help of hardware scheduling on the Jetson TK1. A single frame(720x1280) took 15.91 ms nearly 48% of the real time window. The time window for a single frame to perform in real time following the 30 FPS standard is around 33 ms. Nearly 17 ms remain for additional processing.

3.1. RGB to Grayscale Conversion

The RGB to Grayscale conversion algorithm takes a single HD video frame and converts the RGB triple at each pixel location to a single gray scale pixel value. This is a basic image processing function and libraries such as OpenCV [1] include such operations.[1]

The floating point version read three channel values for a pixel, performed a weighted average on the three values, and returned the rounded off result as a gray scale pixel value. This version requires three floating point multiplications and two floating point additions. The calculation for the floating point version of the algorithm can be seen as follows.

$$gray = 0.299R + 0.587G + 0.114B$$

The integer version builds a table of integer constants for each shade of R,G and B. The RGB triple looks up corresponding channel values from the table, thus retrieving three values. These three values are added and then right shifted by 14. This conversion gives approximately the same result as the floating point version.

The integer version is nearly 2 times faster on the ARM processor of the Jetson TK1 (serial) hardware. However we did not observe this magnitude of difference between the float and the integer on GPU hardware. The operation is a memory bound operation and it is limited by the movement of data in and out of GPU cache structure. Due to the low arithmetic intensity, the operations themselves are of little consequence to the performance. Hence there is only a small difference between the two versions when run on a GPU (approx. 8%).

[1]The implementation can be done in floating point arithmetic or integer arithmetic. We implemented both.

3.2. Scheduling

Programming for a CPU and GPU differs in the way code is written and executed on the two platforms. A 2D loop is programmed as shown in Figure 3.1(a) on the CPU. This iteration space can be visually represented in Figure 3.1(b). The iteration space on a GPU is a mapping of iterations to threads. The mapping is calculated as seen in Figure 3.1(c). Using this mapping, every thread calculates the gray scale value at exactly one pixel location.

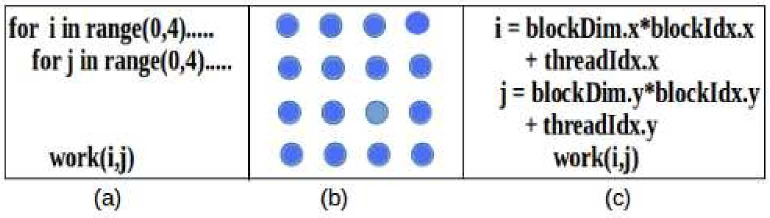


FIGURE 3.1. a) For Loop for 2D iteration space on CPU b) Visual representation of 2D iteration space c) 2D iteration space on GPU

The full iteration space is divided into blocks and blocks are mapped to Streaming Multiprocessors. This concept is illustrated in Figure 3.2. The 2D iteration space(4x4) is broken into blocks (2,2). In other words, creating 2 threads in the x direction and 2 threads in the y direction. The indices of the thread are calculated using built-in CUDA API's and can be seen as follows:

$$i = blockDim.y * blockIdx.y + threadIdx.y$$

$$j = blockDim.x * blockIdx.x + threadIdx.x$$

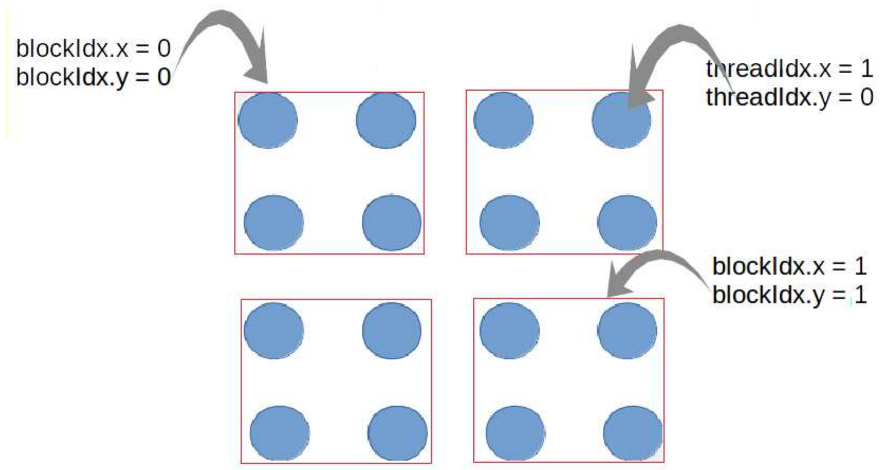


FIGURE 3.2. 2D iteration space divided into blocks on a GPU.

The 2D Block Scheduling uses a one to one iteration to thread mapping on an image of size 720x1280. The image is divided into a grid of valid sized blocks that is a block size that totally covers the underlying iteration space of the image without over or under fitting. Each block is a set of iterations. Each iteration has a unique identity (i,j) wherein i and j are the global indexes of every pixel in the image. Figure 3.3 illustrates that each iteration space corresponds to a single pixel value RGB.

In this schedule each thread reads the R,G,B values of the corresponding pixel and performing operations on them. Each iteration writes back a single data value to the output image.

For example, in Figure 3.3 the image is divided into blocks of size (5,4). Hence each block in the grid is a set of 20 unique mappings of threads to a global pixel index. The ideal number of blocks depends on the hardware under consideration. In order to get an ideal block size for the benchmark application we performed a block size parameter sweep for the float and integer versions of the benchmark.

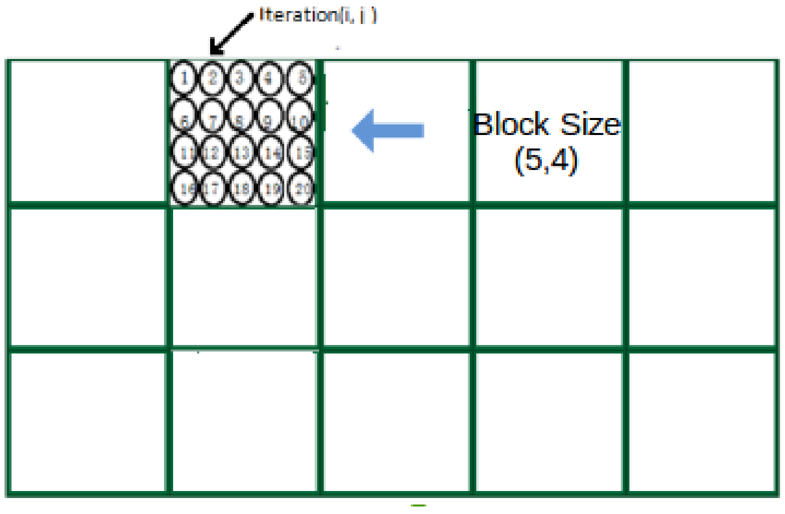


FIGURE 3.3. Iteration space for 2D blocking for image size 720x1280.

3.3. Experimental Results

The execution time for each variant was measured for a single frame. The Table 3.1 illustrates the result:

TABLE 3.1. Comparison of execution timings of different variants of the benchmark demonstrating the performance on CPU and GPU. The table highlights that this is a memory bound operation on the GPU and hence this affects the performance comparison between the integer and float version on the GPU. Also this table demonstrates the results of best block sizes of hardware scheduling on the target hardware.

Variant	Total Processing time(ms)
float serial on a CPU	96.85
integer serial on a CPU	50.15
blocked parallel float on GPU (best block size(128,1))	17.25
blocked parallel integer on GPU (best block size(32,4))	15.91

The table results demonstrate that for serial hardware the integer algorithm for this benchmark is nearly twice as fast as the float algorithm. However, the 2D Blocking on parallel hardware, is 8.42% faster for the integer than the float variant of the benchmark. As we can observe in the table, the 2D blocked integer variant of the benchmark gives an optimum performance of 15.91 ms for a block size(32,4) as compared to 17.25 ms for the 2D Naive blocked float variant of the benchmark.

Figure 3.4 and 3.5 represent graphically the performance of the float and integer benchmark block parameter sweep. These graphs show the variation in execution timings for different block sizes for the two variants. The x-axis reads the total block size or the total number of threads assigned for running that configuration. The legend tells us the size of the block in the x direction. Therefore 128X means 128 is the size of the block in the x direction that is the most contiguous direction.

We can observe that in the graphs Figure 3.4 and Figure 3.5 the performance becomes nearly flat for block sizes greater than 64. We can argue that for smaller block sizes the number of blocks launched for the entire application is more as compared to the larger block sizes. There is an overhead associated with launching more blocks on the hardware as the blocks have to wait in queue for processing due to the limited number of block schedulers on the hardware. Thus smaller block sizes introduce more overhead for the same task.

Figure 3.6 and Figure 3.7 are zoomed in graphical representations of the Figure 3.4 and Figure 3.5 respectively. From these graphs we can observe that the shape of the blocks does not help to effectively improve the performance of the application. For example in both the graphs the different configurations of block sizes in the x direction do not give a big performance win. Say for example we take a configuration (32,8), (64,4) or (128,2) almost have the same performance timings. Hence we can conclude that different shapes of the block are not affecting the performance of the application. That means memory transfers are playing a major role than computational efficiency for the two versions of the benchmark.

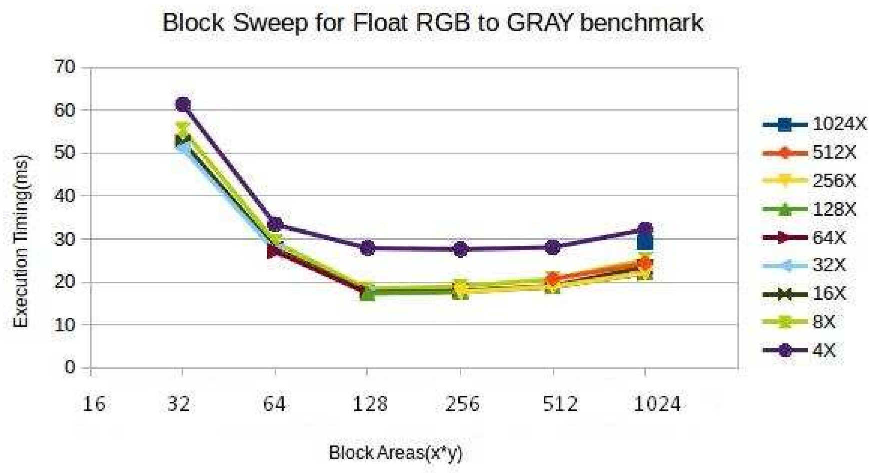


FIGURE 3.4. Block configuration sweep comparisons for blocked float RG-BtoGRAY. The conclusion from this graph is that performance improves with increase in block size.

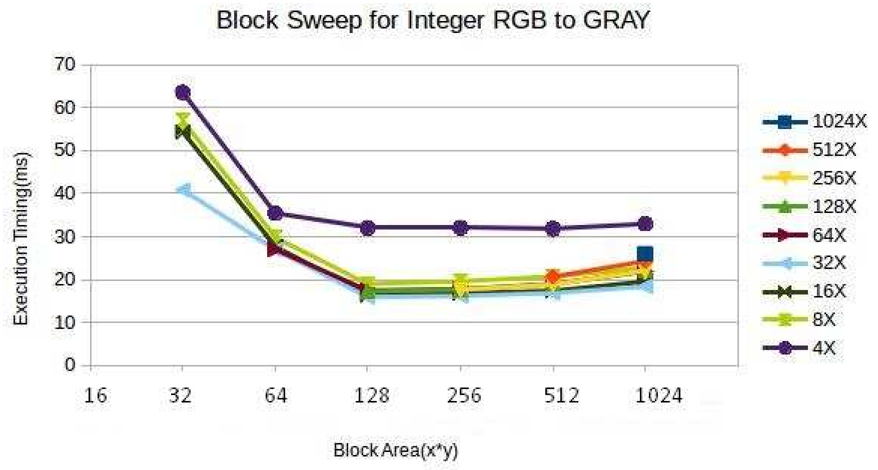


FIGURE 3.5. Block configuration sweep comparisons for blocked integer RG-BtoGRAY. The conclusion from this graph is that performance improves with increase in block size.

3.3.1. FRAME RATE RESULTS

The 2D Naive Blocked integer variant achieves a frame rate nearly two times the real-time performance standard of 30 FPS. It has a measured execution time of 15.91 ms for a single frame. Therefore, the number of frames that can be processed in a second is (1/0.01591) =

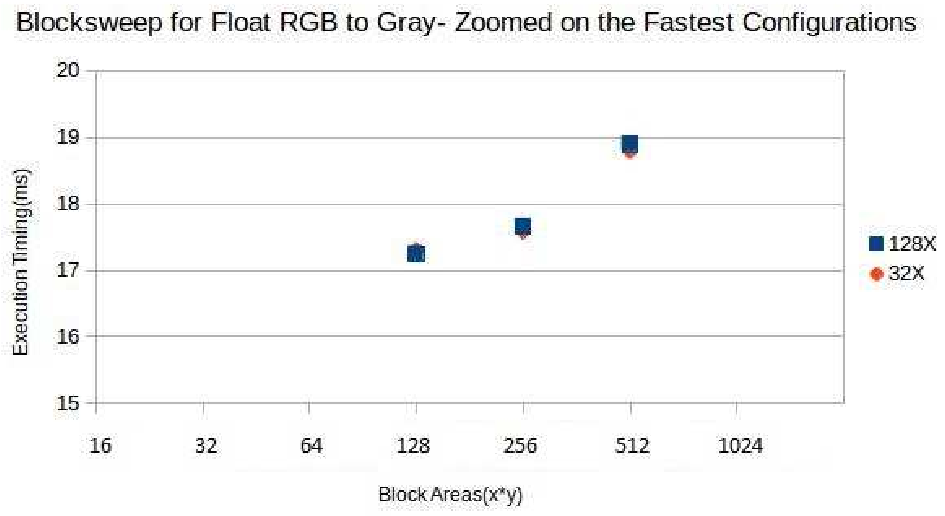


FIGURE 3.6. Block configuration sweep comparisons for blocked float RGBtoGRAY zoomed on the interesting regions cropped from Figure 3.4. This graph highlights that the shape of the block does not affect the performance of the benchmark.

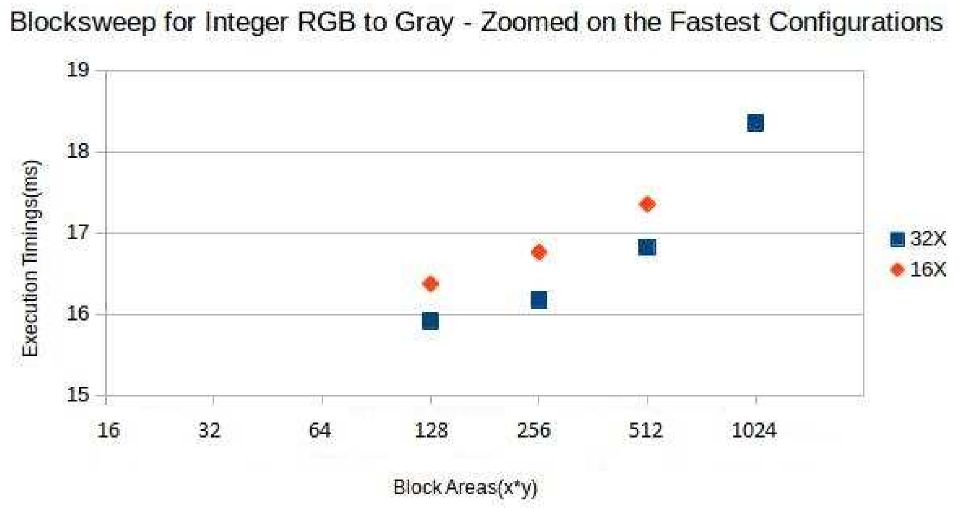


FIGURE 3.7. Block configuration sweep comparisons for blocked integer RGBtoGRAY zoomed on the interesting regions cropped from Figure 3.5. This graph highlights that the shape of the block does not affect the performance of the benchmark.

62.85. Thus the frame rate of our application is nearly 63 FPS. Thus we have surpassed the 30 FPS threshold for real time performance.

3.4. Conclusion

The benchmark was developed to measure the bottle necks in a very simple algorithm implemented on Jetson TK1. The benchmark chosen is a memory bound operation. We observed that memory transfers are free from the host side to GPU side on a Jetson. The memory traffic between main memory and registers host is a major bottleneck for our application.

The results for the float and integer 2D blocked variants reveals that though serial versions differ in performance the parallel versions give nearly the same performance. Therefore we can conclude that we require to give more importance to optimizing interactions with the GPU cache structure in order to optimize complex operations on this hardware.

The experiments also reveal that block size is an important programming aspect of scheduling on the Jetson and we can achieve appreciable improvement in performance by selecting the right block size. The best block size for this benchmark was recorded as(32,4).

Thus the performance achieved by this benchmark demonstrates the potential for real time performance in more complex algorithms.

CHAPTER 4

ViBe: Optimized for the Jetson TK1

The following chapter describes the optimizations performed on the ViBe algorithm targeting the Jetson TK1. ViBe, a patented algorithm, was first released in 2009 [3]. The latest version of ViBe was released in 2014 [13] and the one supplied to us by the authors is the current optimized version. It is the functional and evaluation baseline for our work.

We implemented our own simplified non-optimized version of the original ViBe. The author's code is highly optimized for a CPU. The programming is quite complex, as is expected of highly optimized code. Therefore, a slower more simple implementation provides a better starting point. Our version of ViBe was developed to test performance optimizations on the GPU. As we started with a new code base it was important to verify the functional similarity with the original ViBe.

ViBe is a non-deterministic algorithm as it randomly allocates a new background model for segmentation on each run. Running the original algorithm twice gives subtly different results. There can be pixels that are classified as foreground for a frame in one run and background in the other depending on the background model of the pixel. Hence, we adopted a statistical strategy to verify functionality that does not demand an exact result per pixel per frame. Our strategy is to compare the probability of a pixel being foreground or background, determined through multiple runs of each implementation. Through this statistical analysis we were able to demonstrate the functional equivalence between the original ViBe and our serial implementation. This baseline serial implementation on a single ARM processor of the Jetson TK1 was further optimized that achieved a frame rate of 55.33 FPS.

4.1. Verification Procedure

This section describes the procedure used to verify the functional similarity between the original ViBe implementation and our serial implementation of ViBe. This procedure uses a statistical approach to verify the claim of functional similarity between the above two implementations of ViBe. The task of optimization can produce desired results. However it is important to verify if the nature of the algorithm is not altered in the process of optimization. This verification procedure was important to ensure that we did not change the fundamental functionality of original ViBe during the course of our own implementations.

4.1.1. Generation of Statistical Map

The nature of the verification required us to generate two foreground counter maps, for the original implementation and our serial implementation. The foreground counter map is an illustration of the probability of the pixels being foreground. Pixels always labelled background are white, always labelled foreground are black, and the others are gray scale. The pixel locations whose answers vary are further examined using a histogram. Figure 4.1 and 4.2 are an illustration of the foreground counter map and the histogram.

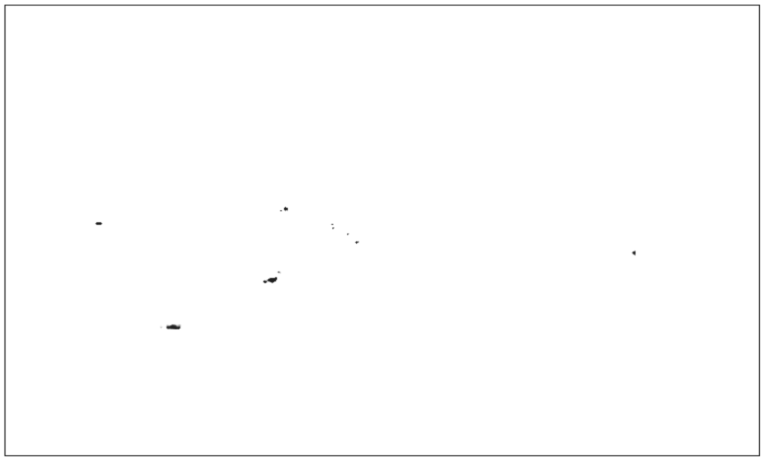


Figure 4.1. Foreground counter map.

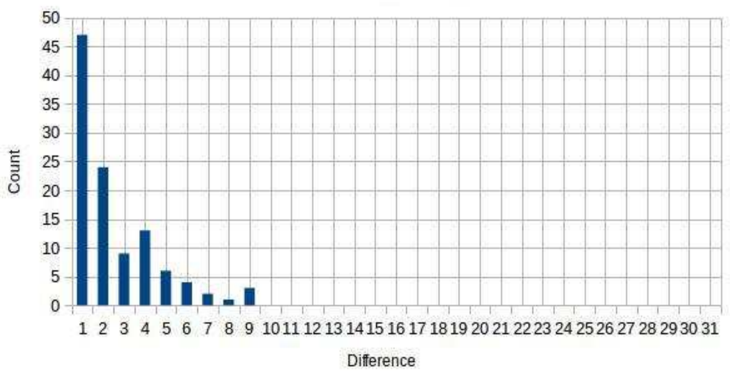


FIGURE 4.2. Histogram of differences.

The following section describes in detail the procedure used to generate the foreground counter maps and the histogram from it. The input to the algorithm is a video of traffic and the output is a video of segmented frames. In our verification we focus our attention only on a single frame. A run is defined by 32 executions of the same implementation. As our result, we obtain the segmented output of the chosen frame. Therefore, we obtain a collection of 32 segmentation maps for the chosen frame. The frame we choose for evaluation is frame 350. Testing and observation shows the video frame 350 as one of the more challenging frames of the video. The scene of the frame involves cars stopping and moving again. Thus we choose for verification the frame having the most chances of giving variation in results. Figure 4.3 displays frame 350 of the video chosen for testing and evaluation.

The first step is to run the reference implementation and our implementation each 32 times. The goal is to compare the variation between two runs of their implementation with the variation between a run of theirs and ours. Hence we run their implementation twice (each 32 times) to collect a folder of 32 frames each. This gives us two folders of 32 segmented maps for their implementation and one for our implementation. We generate the foreground counter maps from this folder of 32 segmented frames. The foreground counter map is a

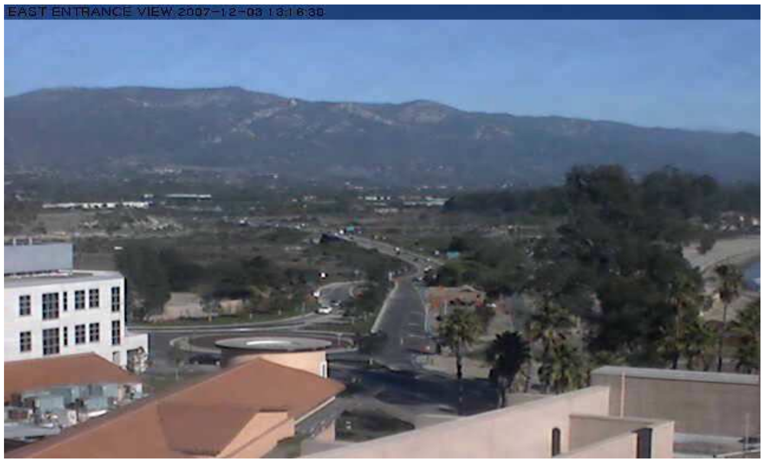


FIGURE 4.3. Test frame 350 of a video dataset.

counter to track the number of times a particular pixel would be segmented as a foreground in a run. Hence we run a foreground pixel counting algorithm on each of the 32 frames of the chosen folder. We get a counter map the same size of the frame, where each pixel of the counter is a count of the number of times that pixel got segmented as foreground. We obtain three such foreground counter maps, two trials for the reference and one for our implementation.

We compute a pixel by pixel absolute difference between the counter maps to obtain a difference map. Each pixel in the difference map is indicative of the difference in result between the original and our implementation of ViBe. It is indicative of the number of times a pixel got wrongly classified as foreground or the number of times it was supposed to be classified as foreground but it did not appear to be.

We generate a histogram using the difference map. The x axis of the histogram are the possible difference values and the y axis is the count of the differences for each possible difference value. For example if a difference 6 occurs 100 times in the difference map, the bin 6 on the x axis of the histogram would measure 100 on the y axis. The histogram formed from this procedure would then be used for evaluation for correctness. The analytic comparison of the histogram is explained in the next section.

4.1.2. Analysis of Histogram

The maximum possible difference value in the histogram would be 32 for those pixels that are segmented as foreground by the reference algorithm and background by our implementation and vice versa. A large value of this bin would mean high false positive or false negative rate. The smallest possible difference would be zero meaning complete agreement. We would expect the height of this bin to be the highest. We obtain two histograms one with the difference map generated by the difference between two runs of the original implementation and the second difference map of the differences between our implementation with the original implementation. In the comparison of the two histograms, the difference in counts of the bin do not matter,but in fact the shape of the histogram would be decisive. If the shape of the histograms matches then we can say that our algorithm is functionally equivalent to their algorithm.

4.1.3. Analysis of the Difference Map

The difference map produced by subtracting the foreground counter maps is an image that gives us the regions in the frame where there was difference in classification. As mentioned earlier the maximum value in the difference map is 32 and the lowest value is zero. To obtain a visual image showing the differences we scale the difference map where maximum value scales to 0 and the least value is 255. By comparing the difference map of their implementation with their implementation and the difference map of their implementation with our implementation we can see the regions where the differences are evident. Depending on the variation of the highlighted regions in the difference maps we can say that the two implementations are functionally close or not.

4.1.4. Experimental Results of Serial Verification

This section provides data that demonstrates the functional similarity between their implementation of ViBe and ours. The results are for a video sequence of 685 frames of size 480x640 of a traffic scene with continuously moving cars. Out of 307200 pixels in a frame, our implementation agrees on 307042 pixels with the original implementation of ViBe. Thus we get nearly 99.94% correct classification like the original. In the scope of this algorithm the meaning of a foreground pixel is a moving object in the scene.

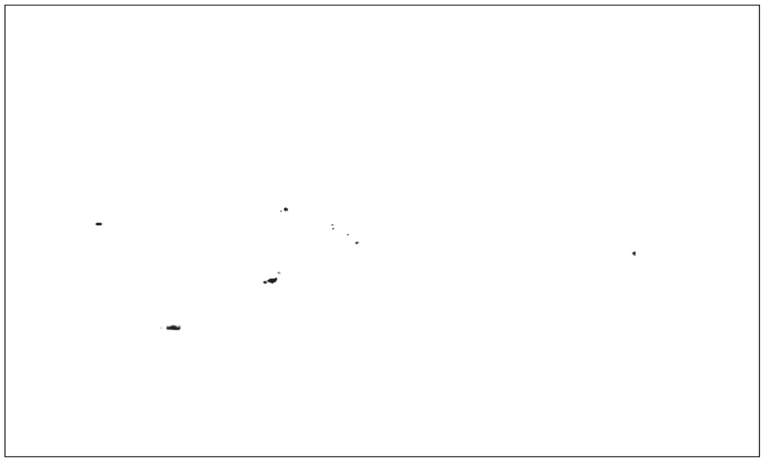


Figure 4.4. Foreground counter map for first run of original implementation

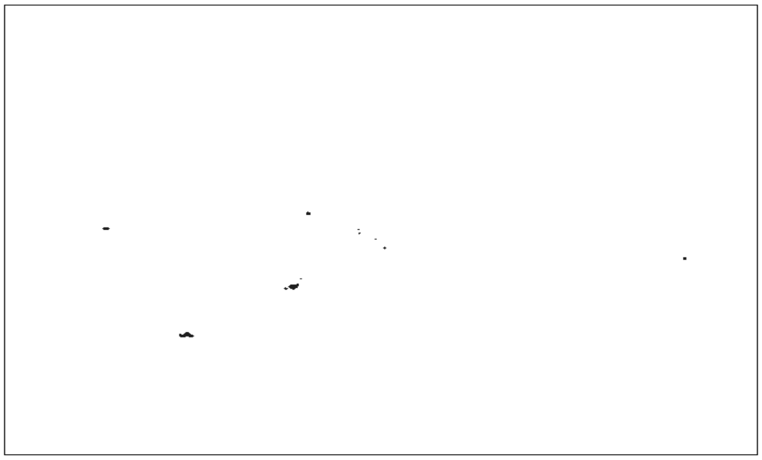


Figure 4.5. Foreground counter map for second run of original implementation

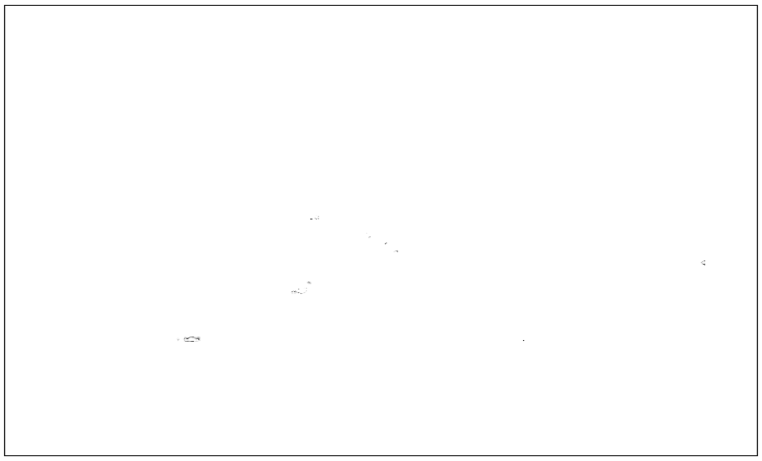


Figure 4.6. Difference map for original ViBe

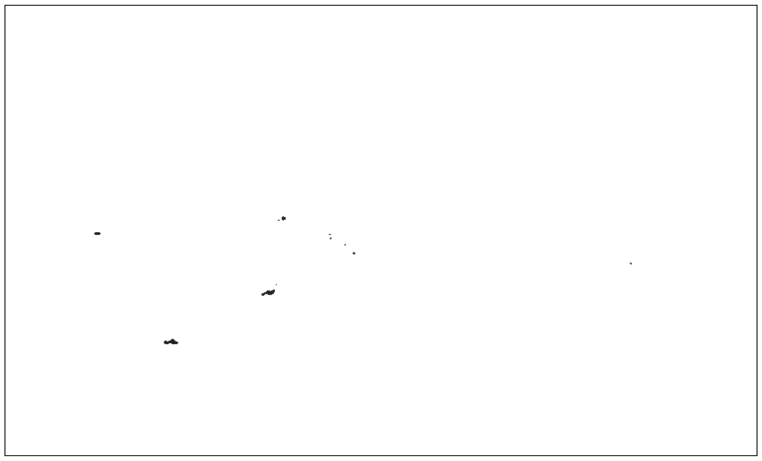


Figure 4.7. Foreground counter map for one run of our implementation of ViBe

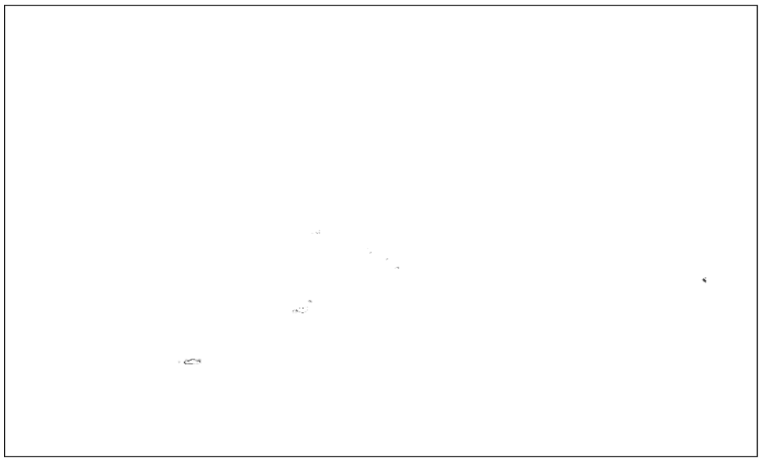
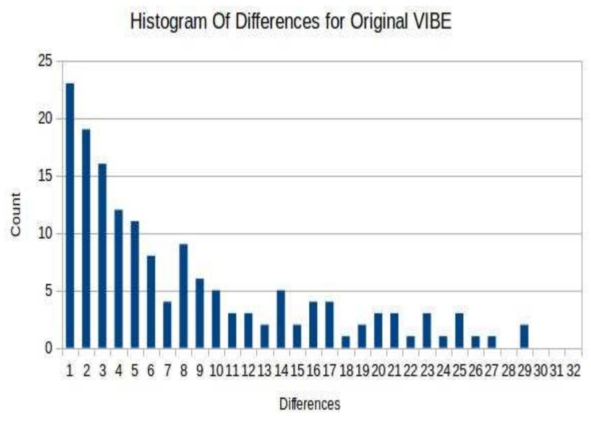


FIGURE 4.8. Difference map for our ViBe Vs Original

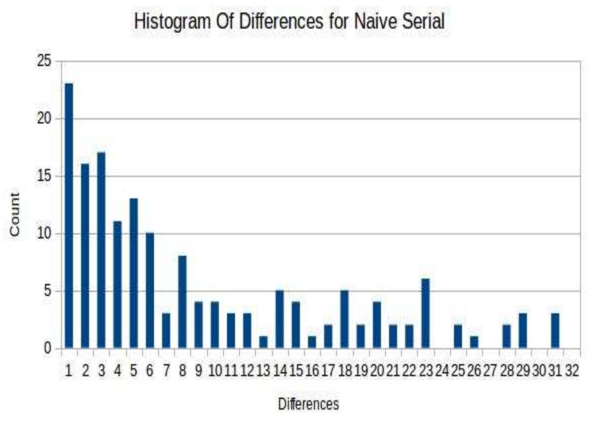
Figures 4.4 and 4.5 are the foreground counter maps for two runs of the original algorithm. By careful observation they are quite similar but as expected, not an exact match. The difference map in Figure 4.6 further provides more proof to the above statement. The very light shades of gray barely visible in Figure 4.6 indicate that there is very slight disagreement between the two underlying implementations being compared. The Figures 4.4 and 4.7 are the foreground counter maps for the original and our implementation of ViBe. From the comparison of the two we can also conclude that they are very similar but not exactly identical. Their difference map in Figure 4.8 further highlights these differences.

The differences in the difference maps mostly vary similarly. Hence we can observe that the difference highlighted in the difference maps 4.6 and 4.8 are mostly around the same region and have almost similar intensity. This can be used as proof to our support our claim of functional equivalence. This can be further evaluated using histograms produced from these difference maps.

The histograms in Figure 4.9(a) and 4.9(b) capture only the differences between the original implementation and our serial implementation. This is roughly 150 pixels out of 307,200 whose classification varies. From the histograms we can say that the two histogram



(a) Histogram for comparison between two runs of Original VIBE.



(b) Histogram for comparison between our implementation and Original VIBE.

FIGURE 4.9. a)Histograms of differences for original ViBe b) Histogram of differences for original Vs our serial implementation.

shapes look the same. If we notice carefully the height of all the bins is not similar, however the shape of the entire histogram is qualitatively the same. The first bin of the histogram of their implementation comparison implies that 25 pixels in a frame have the 1/32 probability of being differently classified. Similarly this comparison for our histogram is 23 pixels with a 1/32 probability. Roughly out of the 150 pixels whose classification varies by the two algorithms around 90 are below the difference count of 6. Thus the false negative and false positive rate of both the algorithms is also comparatively small. From this data we can say that our algorithm performs statistically similar to the original algorithm.

4.2. Baseline Comparisons

The original ViBe is our performance evaluation baseline for real time processing. The frame rate achieved by it is approximately 37 FPS on the ARM processor of the Jetson TK1. Figure 4.10 describes the flow of operations of ViBe on the CPU. Table 4.1 shows

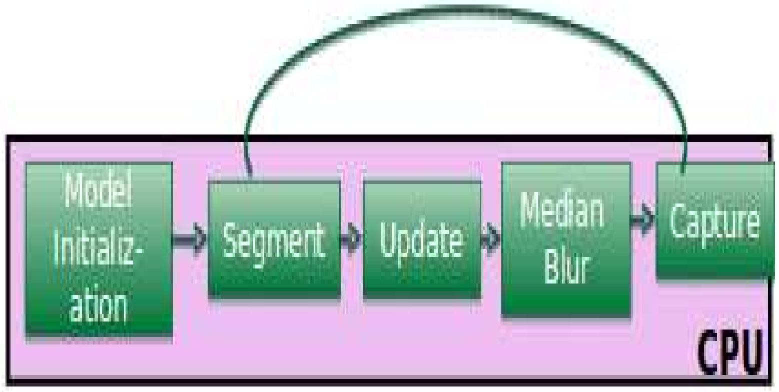


Figure 4.10. Serial flow of ViBe on the CPU

the time taken per function by our serial implementation of ViBe and the original serial implementation of ViBe.

Table 4.1. Table of comparisons between Original and our Serial Implementation of ViBe.

	Initialization	Segment	Update	Median	Capture	Total
Our serial VIBE	1.17	7.58	20.82	7.41	1.95	38.98
Original serial ViBe	1.34	6.90	0.85	7.38	2.05	18.56

Our goal is not to do serial optimizations, but to port the code to CUDA. Thus the next section describes the stages of translation of the serial code to a parallel implementation.

4.3. ViBe:CUDA Implementation

Four parallel versions of ViBe were implemented. The naive serial implementation performed poorly with a frame rate of 17 FPS. After applying the optimizations the CUDA implementation gives a frame rate of 55 FPS, nearly 3 times faster than our serial implementation and 1.5 times faster than the original ViBe implementation.

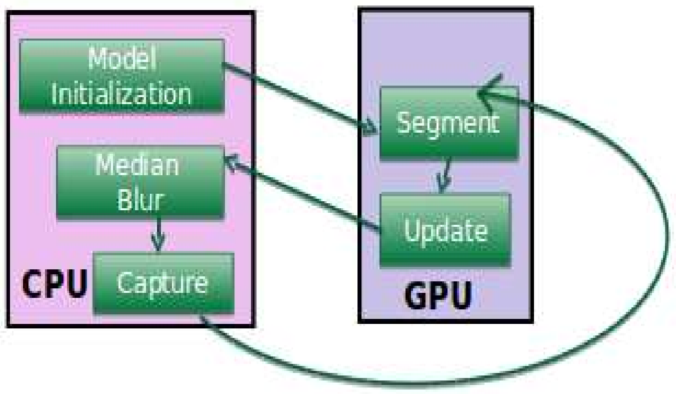


FIGURE 4.11. Naive parallel flow of ViBe on the CPU and GPU

4.3.1. Naive Parallel Implementation

The results in the previous section shows us that there is a major difference in the amount of time taken by their implementation and our implementation for segment and update functions. Hence it was important to import these functions to the kernel and check their performance. The naive parallel implementation of ViBe ports the segment and update kernels to the GPU. Figure 4.11 pictorially depicts the functioning of ViBe on the CPU and the GPU.

A grid of blocks was launched equal to the size of the frame and each thread in the block of threads was responsible for operations on the corresponding pixel of the frame. There was no change in the algorithm . This is similar to the 2D Naive Block mapping used in the benchmark. The basic idea was to get a datum of performance evaluation for the naive parallel and perform optimizations further based on this result.

To get the best performing timing on the GPU, a parameter sweep for different legitimate block sizes was done on the naive parallel version. The different sections of the algorithm have been timed individually by 6 timers and a total timer summing and giving the total execution time. Table 4.2 presents the timing results for the best performing block size.

TABLE 4.2. Timing results for Naive Parallel Implementation of ViBe.

Initialization (secs)	1.29
Segment Frames (secs)	9.11
Update Frame (secs)	19.17
Memory Transfers (secs)	2.86
Median Blur (secs)	21.83
Capture (secs)	3.82
Total Time (secs)	63.08

The results have been achieved for the block size (256,4). From the table above we can observe that importing the two kernels to the GPU, improved the timing results for the update function but not for the segment function as compared against our serial ViBe. Moreover we can observe that the median blur operation on the host side is taking significantly more time in our parallel implementation.

Median Blur is implemented as an OpenCV function on the CPU. The memory gets moved to L1 cache and this makes the operation much faster. But in our parallel implementation the GPU copies back the map for median blur operation from GPU global memory to CPU memory. This means that the map does not reside in cache. Hence this increases the time for median blur in our implementation.

The segment and the update do not show any significant improvement in performance because in CUDA there is an overhead associated with launching multiple kernels. This is because with each new kernel there is a need to move data from GPU global memory to the registers. This memory transfers adds latency to the application. Also this suggests that there is synchronization between the functions on the GPU and the CPU that might be causing the overhead.

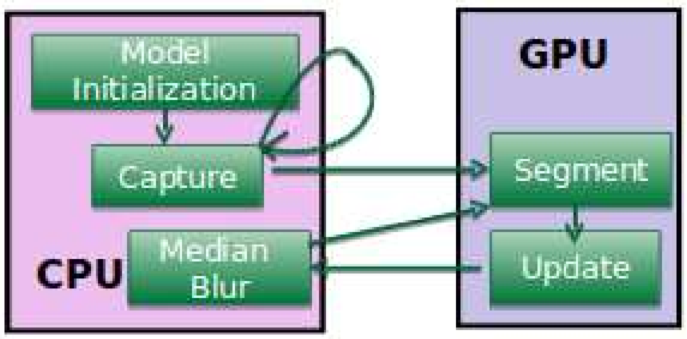


Figure 4.12. Overlap between functions of ViBe on the CPU and the GPU

The functional verification of this optimization was conducted in a similar fashion as for the serial code. The evidence for functional verification of the naive parallel implementation has been given in Appendix A.

4.3.2. Naive Parallel Overlap

The next optimization we present is the removal of synchronization between the CPU and GPU functions. The division of timing results show that the capture operation takes some amount of the total execution time. The capture is used to read the frames of the incoming video. This is done using the VideoCapture class of the OpenCV library in the serial version. If the frame is not captured faster than the processing of it, then this can add latency to the algorithm. Hence it is very important to perform the capture and ViBe simultaneously so that the latency can be reduced. This function cannot be imported to the GPU like the segment and the update functions. Hence the only way to optimize this task is by performing the capture on the CPU simultaneously while the segmentation of frames happens on the GPU. Figure 4.12 represents the overlap between the functions on the CPU and the GPU.

Once the frame captured from the CPU is transferred to the GPU, the capture instance on the CPU can start reading the next frame. Hence by the time the GPU finishes processing, the next frame is ready for segmentation. This is done by making use of the asynchronous kernel calls of the GPU. Once the kernel is called the control immediately transfers to the next line in the CPU. Hence by calling capture frame instance immediately after the segment kernel we can perform both the tasks simultaneously. The outcome of this optimization can be seen in Table 4.3. The evidence of functional verification of the naive parallel overlap implementation has been given in Appendix B.

Table 4.3. Comparison of parallel optimization results for ViBe.

	Naive Parallel	Naive Parallel Overlap
Initialization(s)	1.29	1.33
Segment Frames(s)	9.11	7.10
Update Frame(s)	19.17	12.24
Memory Transfers(s)	2.86	2.09
Median Blur(s)	21.83	19.57
Capture(s)	3.82	0.01
Total Time(s)	63.08	45.73

The above results are for the block size (512,2). The overlap between the GPU and CPU functions has lead us to slightly faster timing results than the Naive Parallel Implementation. The results show that the update kernel got faster by nearly 6 seconds, median blur timing results reduced by 2 seconds and the capture timing recorded is negligible.

4.3.3. Naive Parallel Blur Implementation

The original ViBe performed a median blur filtering on the segmentation map of every frame. Median blur is a smoothing filter that selects a median value for a pixel from a neighbourhood of 3x3 pixels and replaces the center value with the median value. This filter is implemented

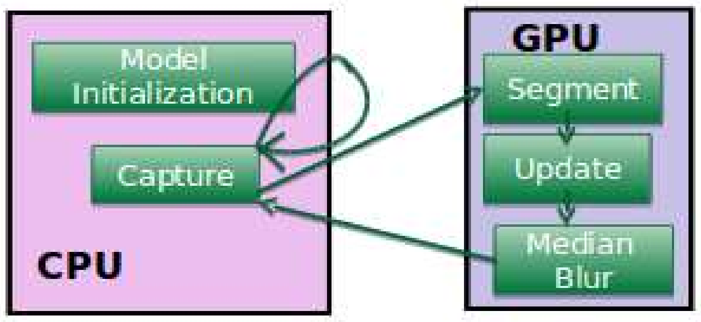


Figure 4.13. Transfer of median blur of ViBe to the GPU and the flow of algorithm between GPU and CPU.

as an OpenCV function in the original ViBe. Though this implementation is faster on a serial machine, it is much slower on the host side of a GPU. This suggests the need to develop a kernel version of the median blur on the GPU. Figure 4.13 represents the transfer of the median blur function on the GPU and the flow of the algorithm.

The median blur kernel implemented on the GPU is functionally similar to the OpenCV implementation. This kernel makes use of shared memory for performing the smoothing operation. The shared memory is the local memory for a block of threads. As shared memory is faster than global memory it is a good option to transfer the segmentation map to shared memory and perform read operations from it. The write, however happens only to the global memory. The use of shared memory significantly accelerates the process.

Table 4.4 shows the timing results collected on a block size of (256,1). The inclusion of the median blur kernel onto the GPU greatly improved the performance timings of our parallel implementation as compared to the previous parallel implementations. Adding the blur to the GPU has got us as close as 4 seconds to the original implementation. Another observation is that this optimization has also reduced the timing results for the segment and the update kernels. Now we can say that the latency added to the application has

TABLE 4.4. Comparison of Parallel Optimization results for ViBe.

	Naive Parallel	Naive Parallel Overlap	Naive Parallel Blur
Initialization(s)	1.29	1.33	1.28
Segment Frames(s)	9.11	7.10	5.98
Update Frame(s)	19.17	12.24	6.63
Memory Transfers(s)	2.86	2.09	3.06
Median Blur(s)	21.83	19.57	1.47
Capture(s)	3.82	0.01	2.19
Total Time(s)	63.08	45.73	22.28

been reduced and the segment and update functions can now be optimized further. The evidence for functional verification of the Naive Parallel Blur implementation has been given in Appendix C.

4.3.4. Fused Parallel Blur Overlap

The Fused Parallel Blur Overlap is the implementation of ViBe that fuses the segment and update kernels. Fusion of kernel calls is to perform computations of both the kernels in a single kernel call. Memory operations are slower and the memory bandwidth on a GPU is limited. The idea to perform kernel fusion is to share data among kernels with a common data space without needing to copy or transfer memory for the separate kernels. This improves the memory locality of the application and hence reduces memory operations. The segment and the update kernels share the model, image and segmentation map data. The operations on every pixel are independent and hence each thread can independently operate on a pixel. This makes the kernel fusion possible for the segment and update functions. Figure 4.14 pictorially represents the kernel fusion optimization on the GPU.

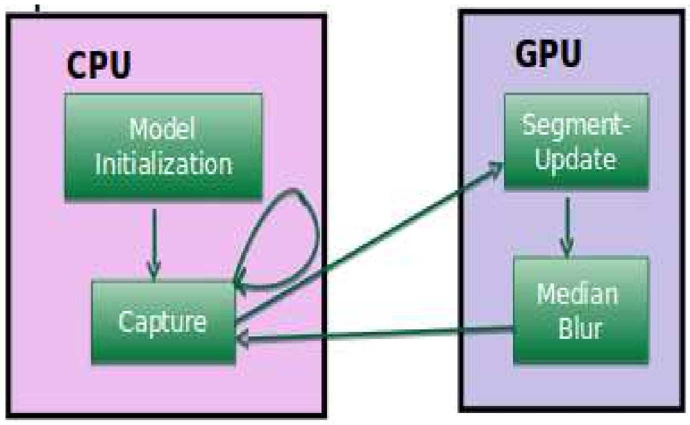


FIGURE 4.14. Kernel fusion of segment and update functions of ViBe on the GPU

TABLE 4.5. Comparison of Parallel Optimization results for ViBe.

	Naive Parallel	Naive Parallel Overlap	Naive Parallel Blur	Fused Parallel Blur Overlap
Initialization(s)	1.29	1.33	1.28	1.29
Segment Frames(s)	9.11	7.10	5.98	6.02
Update Frame(s)	19.17	12.24	6.63	0
Memory Transfers(s)	2.86	2.09	3.06	1.46
Median Blur(s)	21.83	19.57	1.47	0.13
Capture(s)	3.82	0.01	2.19	0.18
Total Time(s)	63.08	45.73	22.28	12.38

Table 4.5 provides timing results for the block size (256,1). The results indicate that the kernel fusion optimization did improve the results for the two kernels by nearly 6 seconds. The result for the median blur kernel also record negligible timings. Thus we have optimized the pipeline of functions for background segmentation by fusing operations. But with this chain of optimizations we have achieved our goal to out perform the original ViBe by nearly 6 seconds. The proof of functional verification of the Fused Parallel Blur Overlap implementation has been given in Appendix D.

4.3.5. Relative and Real Time Performance

The CUDA implementation records 12.38 s as the best result. This result beats the execution timing of 18.56 s recorded for the original ViBe. The result is for a video of 685 frames with resolution 640X480. Therefore the number of frames that can be processed and transferred in a single second is 685/12.38 = 55.33. Thus the frame rate achieved by our parallel implementation of ViBe is 55.33 FPS.

CHAPTER 5

Related Work

This chapter presents other background subtraction algorithms and how they relate to ViBe. It also describes the other attempts at optimizing ViBe for real time execution. Our work is unique in that it uses optimizations developed for scientific workloads to port a vision application to low cost, portable hardware.

5.1. Background Subtraction Algorithms

This section lists the variations of the ViBe algorithm and other background subtraction algorithms chosen for optimization by other authors.

The 2.4.5 release of OpenCV included an initial GPU implementation of ViBe. However due to the patented nature of this algorithm, the module for ViBe did not make it to any future releases. Tomas Kryjak et al.[9] implements a modified version of ViBe with displacement estimation for moving cameras on a FPGA. This algorithm works in a different colour space namely the CIE workspace as compared to the RGB colour space we use for computations. It tries to correct the segmentation errors caused by a moving camera and therefore saves only a portion of the background model for each frame discarding the rest. Even though the algorithm is modified for moving cameras it functions for static cameras as well according to the original ViBe implementation [4].

The PBAS-Pixel Based Adaptive Segmenter [8] is another modified version of ViBe that uses the ViBe methodology to segment frames. However its model initialization differs from the original Vibe scheme. It depends on a history of N previous samples. PBAS uses the same update model mechanism like ViBe with the only difference that the update depends on two values R(x_i) and T(x_i), the distance threshold and time threshold, that change adaptively for every pixel.

Besides ViBe and its modifications other background subtraction algorithms have been implemented on GPUs. One such algorithm is the background subtraction using a single Gaussian distribution [5]. It uses conditional probability of the single Gaussian to decide if a pixel is foreground or background. Although this method differs in methodology, the decision for every pixel is independent and hence like our algorithm it has two kernels; one for initialization of kernel model and one for processing. Another background method proposed by Raj Gupta et al. [7] uses patch wise matching of feature descriptors instead of pixel intensities for classification between foreground and background pixels.

Background segmentation is mainly divided into parametric and non-parametric methods. The extended GMM Gaussian Mixture Model [12] is a parametric approach to background segmentation where background segmentation is decided based on a distribution of M Gaussians. This approach differs from our approach as ViBe is a non parametric method that depends on the statistical properties of the history of N values for segmentation.

The MOG approach, which is the baseline approach for the above algorithm implementation, has also been implemented on accelerating devices. However it suffers from the drawback that it uses a lot of power. Therefore Shoaib Azmat et al. [2] successfully solves this problem by implementing a low power version of the MOG on the GPU.

5.2. Performance Comparison

The state of the art optimized background subtraction algorithms have been implemented on different architectures like GPUs and FPGAs. To the best of our knowledge only a single attempt publishes work that explores the optimization space available to ViBe on GPUs.

Furthermore, our work is unique in that it targets a low cost, portable, power efficient hardware platform .

Dan Liu [10] presents a fast processing version of ViBe on a GPU using CUDA. The algorithm performs experiments across three image sizes and achieves a speedup nearly between 4.2-6.7 compared to serial implementation. Their optimizations focus on the memory layout of the data in the global memory and ensure coalesced memory reads. They perform their experiments on a NVIDIA K20 GPU which is bigger than the board we are targeting. It does not fit in the image processing pipeline that we wish to target. This work is complimentary to ours and a combination of these techniques should be subject of future work.

Tomasz Kryjak and Mark Gorgon [9] uses a VC707 Xilinx FPGA board connected to a camera through HDMI as a setup for experimentation. The camera takes a 700 x 576 video stream at 50 FPS as input. The setup of their experimentation is similar to our idea of placing the camera close to the processing device. Moreover the power dissipation of this implementation is as low as 4 W, matching the performance expectation specifications of power consumption from our algorithm. The performance of this algorithm on the FPGA is 60 GFLOPS/s. However there is some advantage to using a more easily programmed GPU in the work flow.

PBAS (Pixel-Based Adaptive Segmenter) [6], another modified version of ViBe, has also been implemented on the VC707 FPGA and gives a total power consumption of 6.38 W. The article performs a comparative study of ViBe and PBAS across different architectures. From the analysis it is evident that PBAS only improves the accuracy of the task of background segmentation but the performance of PBAS scales poorly as compared to ViBe on a FPGA. This is because the size of the background model in PBAS is bigger than ViBe. Hence the transfer of extra parameters through the external memory slows down the approach. Therefore the PBAS does not scale well on an FPGA platform as compared to other background segmentation algorithms like Gaussian mixture models.

The background subtraction using a single Gaussian distribution [5] performs the processing of multiple videos in a single run on a GPU. All the data is stored in the GPU global memory. R.Boghdady et al. state their results as the number of videos processed in real time. GPU resources are kept busy by launching enough threads in a block from the available stack of frames. Besides memory latency is reduced by using asynchronous memory transfers. GPU suffers from a problem of low bandwidth for memory transfers. This paper takes care of this problem by using pinned memory on the host side. Besides they also use memory coalescing as a method to increase throughput. They have used a low-end GPU just somewhat similar to the one used in our experiments a GeForce GT 730 GPU with 192 cores/MP that has 2 MPs. They managed to process 9 video sequences at real time as compared to 5 video sequences on a CPU.

The background subtraction order based approach [7] uses the NVIDIA Tesla C1060 with 240 cores and it achieves a 25X speedup for 960x720 image resolution. For most of the algorithms it is observed that the speedup decreases with higher resolution images, but this paper presents improvement with increase in image resolution. They attribute this to more parallelism obtained by keeping more processors busy with increasing image resolution. The optimizations used are asynchronous memory transfer and optimized memory usage. This approach also primarily focuses on optimization of instruction usage by reducing the number of diverging branches in the algorithm.

The parametric background subtraction algorithms like GMM [12] require additional storage of parameters. Hence these algorithms pose a greater challenge as compared to non-parametric algorithms to give good performance scaling. This method uses pinned memory to increase the bandwidth of the GPU. Besides this method also uses memory coalescing for increasing the throughput. It also prefers to use Array of Structure approach as compared to Structure of Array to store pixel components. The method achieves real time performance for varying image sizes from lower resolution to HD images. These experiments were performed on a Core 2 Quad Q9400 CPU and GeForce 9600 GT GPU. This has slightly more GPU resources than the processor we are using.

The target hardware used by the MOG [2] based background subtraction algorithm is the NVIDIA ION that has a very low power utilization averaging 5W for many computer vision applications. This exactly matches the power specifications of the JETSON. The paper manages to achieve 100 fps for a VGA video. We also work on a VGA standard video and our results achieved scale half in comparison. The main goal of this algorithm is to reduce computational and memory cost by aligning memory access patterns. Additionally they also experiment with reducing the register usage per thread in a SM to increase the occupancy of the SM.

CHAPTER 6

Conclusion and Future Work

6.1. Conclusion

A real time system is one in which the data can be transferred and processed in a time frame window of 30 frames in a second. Our optimized ViBe algorithm achieves a frame rate of 55.33 frames per second for a pixel resolution of 640x480. This is within the window with more time to perform more complex operations and develop a more sophisticated vision pipeline. Background subtraction [1] is the initial pre-processing step of a vision pipeline. We can develop a more complex system by adding additional steps for motion detection or object recognition to the above pipeline. Thus we conclude that we have laid the foundation for an optimized computer vision pipeline.

Another important aspect of this thesis was to develop a power efficient pipeline. The JETSON TK1 is a low power processor. The maximum power wattage of any computer vision application tested on it is around 5 W. Hence by developing our pipeline on this hardware we have achieved efficiency by restricting power usage to 5 W.

6.2. Future Work

This thesis uses kernel fusion as a technique to reduce the number of individual function calls to build a single kernel. However, in our implementation the segment and the update kernels were independent of each other. Hence fusing them was easy. However, if we consider the median blur post processing step, it has a dependency on the neighbourhood pixels. Thus the median blur cannot be fused into segment and update without any further efforts to smartly schedule it on the GPU. Overlapped tiling is a technique used to resolve these dependencies and then fuse them in the segment and fuse kernel. This we anticipate will lead to free computation of median blur operation and the time required to process it individually as a separate kernel will completely vanish thus giving us the operation for free. This will further accelerate our baseline vision pipeline and make it more flexible for advanced use.

In the same way we can fuse other functions into this pipeline depending on the system we are trying to build. For example, for a motion detection pipeline we can fuse functions like tracking to the already developed baseline vision pipeline.

Bibliography

[1] Opencv. Released in 1999. 12

[2] Shoaib Azmat, Linda Wills, and Scott Wills. Accelerating adaptive background modeling on low-power integrated gpus. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 568–573. IEEE, 2012. 39, 42

[3] Olivier Barnich and Marc Van Droogenbroeck. Vibe: a powerful random technique to estimate the background in video sequences. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 945–948. IEEE, 2009. 1, 4, 20

[4] Olivier Barnich and Marc Van Droogenbroeck. Vibe: A universal background subtraction algorithm for video sequences. *Image Processing, IEEE Transactions on*, 20(6): 1709–1724, 2011. 38

[5] R. Boghdady, C. Salama, and A. Wahba. Gpu-accelerated real-time video background subtraction. In *Computer Engineering Systems (ICCES), 2015 Tenth International Conference on*, pages 34–39, Dec 2015. doi: 10.1109/ICCES.2015.7393013. 39, 41

[6] Bartlomiej Bulat, Tomasz Kryjak, and Marek Gorgon. Implementation of advanced foreground segmentation algorithms gmm, vibe and pbas in fpga and gpu–a comparison. In *Computer Vision and Graphics*, pages 124–131. Springer, 2014. 2, 40

[7] Raj Gupta, Reddy M Sailaja, Swagatika Panda, Sushant Sharma, and Anurag Mittal. Foreground-background separation on gpu using order based approaches. In *Proceedings of the Seventh Indian Conference on Computer Vision, Graphics and Image Processing*, pages 448–455. ACM, 2010. 39, 41

[8] Martin Hofmann, Philipp Tiefenbacher, and Gerhard Rigoll. Background segmentation with feedback: The pixel-based adaptive segmenter. In *Computer Vision and Pattern*

Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on, pages 38–43. IEEE, 2012. 38

[9] Tomasz Kryjak and Marek Gorgon. Real-time implementation of the vibe foreground object segmentation algorithm. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pages 591–596. IEEE, 2013. 38, 40

[10] Dan Liu. Gpu accelerated background subtraction. In *2015 IEEE 16th International Conference on Communication Technology (ICCT)*, pages 372–375, Oct 2015. doi: 10.1109/ICCT.2015.7399860. 40

[11] Catherine Olschanowsky, Michelle Mills Strout, Stephen Guzik, John Loffeld, and Jeffrey Hittinger. A study on balancing parallelism, data locality, and recomputation in existing pde solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 793–804. IEEE Press, 2014. 2

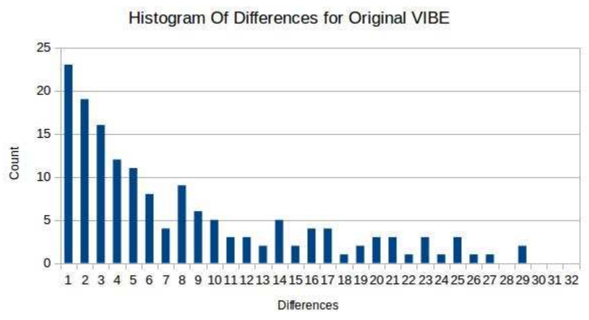
[12] Vu Pham, Phong Vo, Vu Thanh Hung, et al. Gpu implementation of extended gaussian mixture model for background subtraction. In *Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 2010 IEEE RIVF International Conference on*, pages 1–4. IEEE, 2010. 39, 42

[13] Marc Van Droogenbroeck and Olivier Barnich. Vibe: A disruptive method for background subtraction. *Background Modeling and Foreground Detection for Video Surveillance*, pages 7–1, 2014. 20

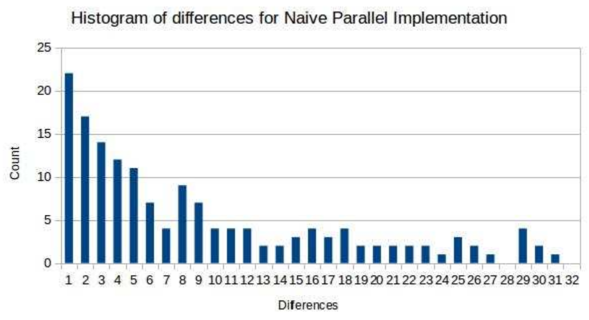
APPENDIX A

Verification of Naive Parallel

The Naive Parallel implementation of VIBE is the porting of the two main functions in the algorithm onto the GPU. As the algorithm is still the same it should not produce any functional difference from the original ViBe implementation. The following histogram shows the comparison of the result of its variation from original ViBe.



(a) Histogram of differences of Original ViBe



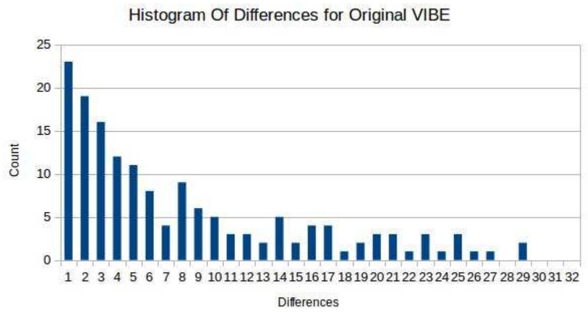
(b) Histogram of differences of Naive Parallel ViBe Implementation

From the comparison of the shape of the above histogram to the serial histogram we can observe that it is qualitatively the same. The number of correct classification of pixels by both the algorithms is exactly the same that is 307043 out of 307200 pixels for both the implementations . Hence we can conclude that the naive parallel implementation of ViBe is functionally equivalent to the original serial implementation of ViBe.

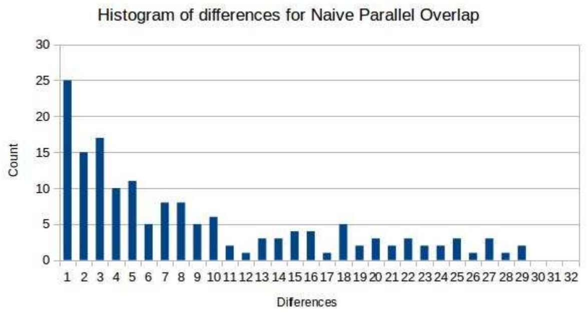
APPENDIX B

Verification of Naive Parallel Overlap

This optimization performs the desynchronization of the CPU functions and the GPU kernel. Hence this version of the parallel optimization should not cause any change to the functionality of the algorithm.



(a) Histogram of differences of Original ViBe



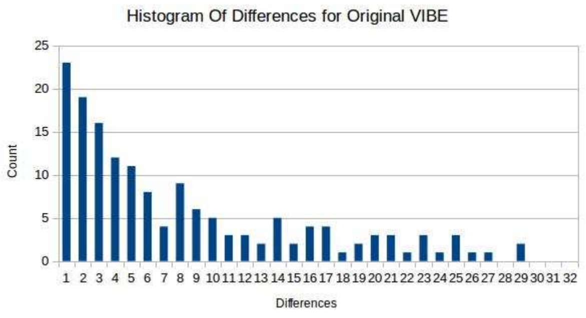
(b) Histogram of differences of Naive Parallel Overlap ViBe Implementation

The matching shape of the histogram to the histogram of original ViBe proves that the above optimization does not alter the functionality of the algorithm. The number of correct classification of pixels by both the algorithms is exactly the same that is 307043 out of 307200 pixels for both the implementations. Thus we are still qualitatively performing equivalently.

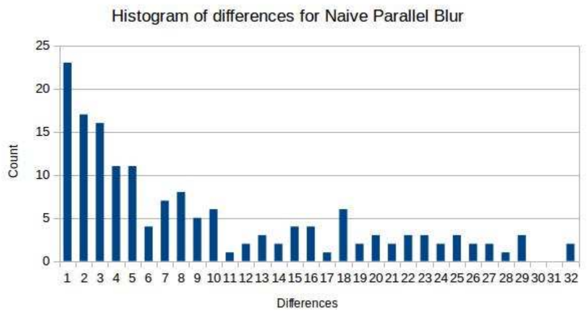
APPENDIX C

Verification of Naive Parallel Blur

The Naive Parallel Blur implementation moved the OpenCV library function of median blur to the GPU. As the original VIBE is using a library function it is a challenge to replicate the same functionality on the GPU. Hence through this verification we can look out for differences if there arise any.



(a) Histogram of differences of Original ViBe



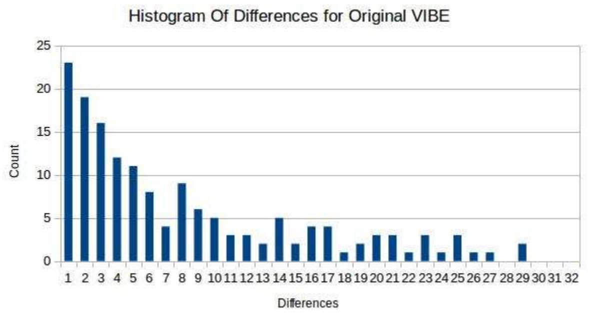
(b) Histogram of differences of Naive Parallel Blur ViBe Implementation

From the above result we can observe that the shape of the histogram is the same. Hence we can say that we have not introduced many wrongly classified pixels in the algorithm. If our implementation of median blur was wrong it would have introduced many wrongly classified pixels and thus changed the shape of the histogram. Hence we can say that the GPU implementation of median blur must be equivalent to the original due to the result seen above. Moreover a test driver for testing median blur individually was also developed. The result of the test driver gave 100% matching between the original and GPU implementation of blur. The above histogram gives 307041 number of correct classifications. All this proves that we have not changed the functionality of our application.

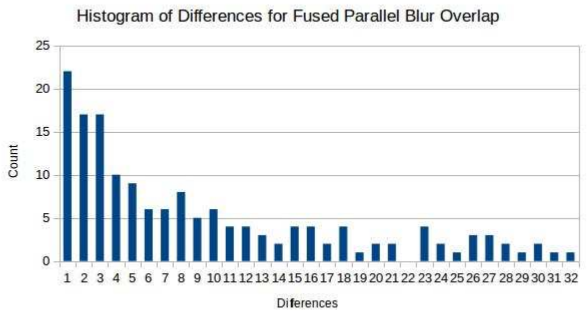
APPENDIX D

Verification of Fused Parallel Blur Overlap

The Fused Parallel Blur Overlap implementation was the combination of the two different kernels of ViBe into a single kernel. The segment and update functions in VIBE are independent that is they do not write to the same data structures. The segment function writes to the segmentation map while the update function writes to the model of the corresponding pixel and its neighbour. Hence there should not be any race conditions in which both the sections of the single kernel are writing to the same memory location. Therefore as per expectations this should not change the functionality of the code.



(a) Histogram of differences of Original ViBe



(b) Histogram of differences of Naive Parallel Blur ViBe Implementation

The shape of the histogram has not changed from the comparison to the serial implementation. This version of optimization classifies 307042 pixels correctly. Hence we can say that we have not changed the functionality due to this version of the parallel implementation.