REAL TIME FAST MOTION TRACKING WITH MULTIPLE CAMERAS USING NVIDIA JETSON TK1

By:
Muhammad Shahid Noman Siddiqui
UET-14S-MCE-CASE-06

Supervisor
Dr. Shoab A. Khan

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING
CENTRE FOR ADVANCED STUDIES IN ENGINEERING
UNIVERSITY OF ENGINEERING AND TECHNOLOGY TAXILA
SPRING 2016

DECLARATION

The substance of this thesis is the original work of the author and due references and acknowledgements have been made, where necessary, to the work of others. No part of this thesis has been already accepted for any degree, and it is not being currently submitted in the candidature of any degree.

_____

MUHAMMAD SHAHID NOMAN SIDDIQUI

UET-SP14-MCE-CASE-06

Countersigned:

_____

DR. SHOAB A. KHAN

Thesis Supervisor

DEDICATION

To my mother for her unconditional love!

ABSTRACT

Motion Tracking is an interesting dynamic task of computer vision which constitutes the basic building block for human computer interaction, autonomous vehicles and video surveillance problems. Due to advancements in video processing and increased expected levels of quality demands higher frame rates and larger frame sizes. This transpired into computationally intensive computer vision algorithms. The inherent structure of computer vision problems incapacitates CPUs from handling them efficiently in real time. GPUs, on the other hand, gained ground where CPUs were incapable since GPUs are handy in sorting out parallel problems efficiently by deploying millions of lighter threads. There is a natural mapping of such tasks to GPU's architecture.

Most of the graphics processors available are installed in either personal computers or clusters and can't be tailored for deployment as a standalone embedded system. The aim of this project is to achieve speed up for embedded GPU platform. A basic tracking algorithm was selected for mapping over NVIDIA's JETSON TK1 platform. Multiple cameras were utilized for simultaneous multi object tracking in real time. Results are compared between CPU, CPU+GPU and JETSON implementation of same algorithm.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
FPGA	Field Programmable Gate Array
SOC	System On Chip
FPS	Frames Per Second
GPGPU	General Purpose Computing Using GPU
GEM	Graphics Environment For Multimedia
HD	High Definition
DCT	Discrete Cosine Transform

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Motion Tracking is a fundamental problem in the domain of computer vision. It is a process of locating single or multiple objects over time using a camera. It has found its applications in the domains of security and surveillance, autonomous cars, traffic control, and medical imaging just to name a few. Motion tracking is challenging because of its computationally intensive nature, video distortion because of camera noise, environmental illumination changes, higher frame rates in videos and greater frame resolutions. Moreover, tracking multiple objects in real time with a multi camera system simultaneously is even more computationally intensive and requires very fast processing speeds. A single threaded processor can process each pixel of each frame sequentially and hence would not be able to produce the desired results. GPUs on the other hand are optimized for throughput instead of latency and can process such inherently parallel tasks very efficiently. GPUs have hundreds and thousands of smaller cores possessing millions of threads capable of processing data in parallel. GPU threads would therefore map efficiently to all the pixels of a frame simultaneously and hence are capable of producing huge performance gains as compared to a CPU.

1.1 MOTIVATION

Image and video processing tasks are inherently parallel in nature. Traditionally, CPUs were deployed for such tasks where a single or a multi core processor with single or multiple threads would be processing image pixels sequentially or would provide a very little parallelism. With the advent of NVIDIA's Compute Unified Device Architecture (CUDA), software developers had now the leverage of explicitly handling millions of threads simultaneously. This architecture is of particular importance to imaging tasks where each thread can have their own set of data to process. All the threads can do the processing in an entirely parallel fashion thus providing massive speed ups as compared to CPUs. GPUs are therefore the best platform for real time computer vision tasks due to their ease of availability, low cost and higher performance.

1.2 PROBLEM STATEMENT

Video surveillance systems have been wide spread across strategic organizations and large and even smaller business centers. There are some extremely sensitive areas in production lines where zero motion is required and everything must be static. Moreover, there can be more than one sensitive area which needs constant monitoring at the same time. A powerful system is required which must be capable of capturing even the slightest of the motion on multiple cameras simultaneously. Moreover, the system must be low cost and low powered for its feasible realization into such surveillance systems.

1.3 OBJECTIVE

The objective is to parallelize a simpler but high performance algorithm which can map well to the GPU resources. An efficient implementation of the algorithms must be done on NVIDIA's JETSON TK1 Development platform. Moreover, the system must be able to track multiple moving objects on multiple cameras in real time. A comparison between CPU and GPU tracking performance is also required in terms of FPS, single versus multiple camera based tracking, different video resolutions and different CPU+GPU systems.

1.4 REPORT ORGANIZATION

In addition to this chapter, the report consists of 5 more chapters. The second chapter is for the literature review and discusses related work. It also addresses GPGPU and CUDA programming model. Third chapter is dedicated to complete system design and discusses sequential and parallel algorithm along with system level design. Fourth chapter discusses the complete software development cycle. Fifth chapter explains the experimental setup and observations. The final chapter is for conclusion and future work.

CHAPTER 2

LITERATURE REVIEW

This chapter considers the literature review related to 'GPU Accelerated Computer Vision', 'NVIDIA CUDA Architecture' and 'GPU Software Model'.

2.1 GPU ACCELERATED COMPUTER VISION

With the advent of NVIDIA's CUDA architecture, there is been a great shift towards realizing the computationally intensive Image/Video processing problems onto GPU based systems. GPUs provide a very low cost platform for visions tasks when compared to FPGA based systems. The GPU software model is also relatively easier than FPGA or other high performance devices in terms of implementation. Several vision related tasks have been implemented on GPUs and a significant performance gain has been achieved.

Zhiyi Yang, 'Parallel Image Processing using CUDA', analyzes the distinct features of CUDA GPU, summarizes the general program mode of CUDA. Furthermore, implement several classical image processing algorithms by CUDA, such as histogram equalization, removing clouds, edge detection and DCT encode and decode etc. Excluding memory transfer time, histogram computation gets more than 40X speedup, removing clouds can get an about 79x speedup, DCT can gain around 8X and edge detection more than 200X [1].

Ir. Rudi GIOT, 'Image Processing Algorithm Optimization with CUDA for Pure Data', accelerated the already existing image processing modules in Pure Data (the GEM library, Graphics Environment for Multimedia) by dispatching a part of the process to the GPU and achieved an accelerating factor of 2734 [2].

Lawrence Chan, 'Parallelizing H.264 Motion Estimation Algorithm using CUDA', demonstrated the viability of a hierarchical (pyramid) motion estimation algorithm in CUDA. This solution addresses the MVp problem while still taking advantage of the CUDA framework [3].

Peter Kuchnio, ‘GPU-Accelerated Foveation for Video Frame Rate Tracking’, accelerated motion tracking in video using a combination of foveation and CUDA technology. To illustrate the technique, the implementation of an optical flow algorithm and its application to motion-segmented video for the real time visual position-servo of a robotic manipulator was provided. Mapping of the foveated motion segmentation algorithm to a 240 processor GPU was illustrated and the performance of the algorithm was characterized with examples of both synthetic and real data. The non-foveated segmentation algorithm shown a significant performance increase over a single-threaded CPU application and the foveated-based segmentation is found to give an additional performance gain of up to 27X over non foveated optical flow [4].

Jing Huang, ‘GPU-Accelerated Computation for Robust Motion Tracking Using the CUDA Framework’, discussed the implementation of a graphics hardware acceleration of the Vector Coherence Mapping vision processing algorithm. They demonstrate how flexibly and readily vision processing algorithms can be mapped onto massively parallelized GPU architecture and achieved performance gain of more than 40-fold of speedup over state-of-art CPU implementation of VCM algorithm [5].

Sidi Ahmed Mahmoudi, ‘Multi-GPU based Event Detection and Localization using High Definition Videos’, proposed an effective exploitation of single and multiple GPUs, in order to achieve real-time detection and localization of abnormal events, using HD and Full HD videos. The proposed approach detects portions of video that corresponds to sudden changes of motion variations of movements. The use of multiple GPUs enabled a real time treatment of high definition videos with a global speedup ranging from 5 to 35, by comparison with CPU implementations [6].

Min Su, ‘Constructing a Mobility and Acceleration Computing Platform with NVIDIA JETSON TK1’, constructed a mobility and acceleration computing platform with NVIDIA JETSON TK1. Besides, two tools, ClustalWtk and MCCtk are designed based on NVIDIA JETSON TK1. These tools both can achieve 3 and 4 times speedup ratios on single NVIDIA JETSON TK1 by comparing with their CPU versions on Intel XEON E5-2650 CPU and ARM Cortex-A15 CPU, respectively. Moreover, the cost-performance ratio by NVIDIA JETSON TK1 is higher than that by NVIDIA Tesla K20M [7].

Yash Ukidave, 'Performance of the NVIDIA JETSON TK1 in HPC', explored the use cases of the JETSON TK1 board as an interface device for cloud computing, and also as a scalable device for energy efficient HPC. They evaluated the performance of the unified memory structure of the TK1 and also the power-performance ratio, as well as energy use, when executing co-scheduled applications [8].

Current high-end graphics processing units (GPUs), which contain up to thousand cores per-chip, are widely used in the high performance computing community. However, in the past, the cost and power consumption of constructing a high performance platform with graphics cards, such as Tesla and Fermi series, are high. Moreover, these graphics cards all installed in personal computers or servers, and then the immediate and mobility requirements can't be provided by this platform. NVIDIA JETSON TK1 (TEGRA K1) is a full-featured platform for embedded applications and it contains 192 CUDA Cores (KEPLER GPU). Due to its low cost, low power consumption and high applicability, NVIDIA JETSON TK1 has become a new research direction.

2.1.1 MOTION TRACKING

Most of the motion algorithms previously tailored onto GPUs are compared against powerful CPUs. Significant speed ups have already been achieved, but the GPUs used were installed in desktop based systems. In this work however, a new low power, low cost and embedded platform has been used for fast motion tracking. Not only, a significant speed up has been achieved over CPUs, but also better tracking results.

Further, background subtraction algorithm previously used was very prone to environmental changes like day and night effects, light illumination problems and camera noise. Even a slightest change in the camera position would completely destroy the algorithm. The algorithm has been modified in a novel way, which adapts to changing back ground. The tracking algorithm adjusts even if the camera is moved to a different background in real time. Eventually, a comprehensive performance analysis has been performed not only between the CPU and GPU, but also among a low powered embedded device against powerful CPU+GPU hybrid platforms.

Results show a significant higher performance to cost ratio for embedded platform as compared to desktop based systems.

2.2 GPU COMPUTING AND GPGPU

The Graphics Processing Unit (GPU) is a specialized processor on a graphics card for highly parallel and computationally intensive tasks. Primarily, it is designed for transforming, rendering and accelerating graphics. It has billions of transistors, much more than the Central Processing Unit (CPU), specializing in floating point arithmetic. The GPU has evolved into a highly parallel, multithreaded processor with exceptional computational power. Since 1999, GPU has been a dominant technology in advance gaming and 3D graphics application.

The difference between a CPU and a GPU is that a CPU is a serial processor while the GPU is a stream processor. Each instruction is fetched and executed by the CPU one at a time. While a stream processor executes a kernel on a set of input data stream simultaneously. The input elements are passed into the kernel and processed independently. This allows the program to be executed in a parallel fashion. Because of their highly parallel architecture, GPUs are outperforming CPUs by some amazing factors on floating point calculations. The reason behind the performance difference lies in the design philosophies between the two types of processors. The CPU is optimized for latency and makes use of complicated control structure to manage the execution of many. The large cache memories used to reduce access latency and slow memory bandwidth also contributes to the performance gap.

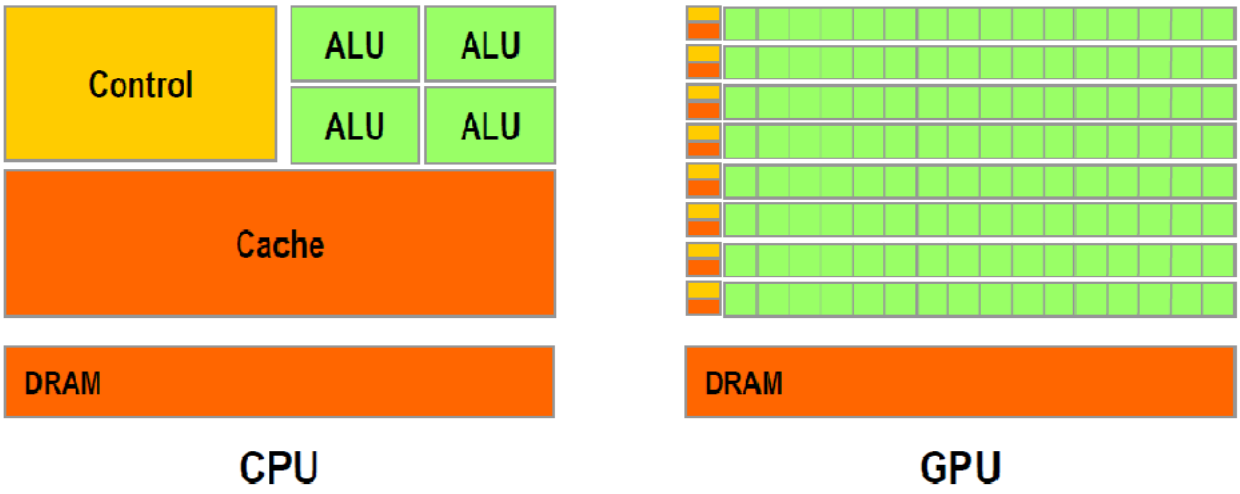

Figure 1 CPU Vs GPU

The design philosophy for GPUs on the other hand has got its motivation by the rapidly growing video game industry which requires the ability to perform massive floating-point calculations in advanced video games. The objective is to optimize the execution of millions of threads, reduce control logic, and introduce small memory caches so that more chip area can be used for floating-point.

A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously.

2.2.1 GPU SOFTWARE MODEL

The CUDA processing flow is very simple, all the sequential tasks are done by a CPU, and only the computationally intensive portions are off loaded to GPU. A generalized CUDA processing flow is as follows:

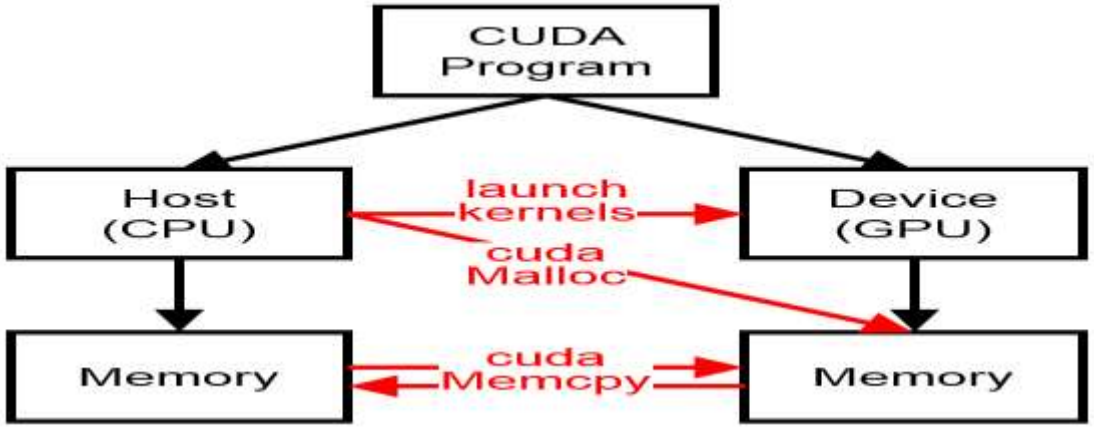

Figure 2 CUDA Program Flow

CPU takes care of the managing all the memory allocation and data transfer on GPU. CPU is also responsible for launching the GPU kernel. Here is an example of CUDA processing flow:

- Copy data from host (CPU) memory to device (GPU) memory
- CPU instructs the process to GPU
- GPU executes in parallel using each core

- Copy back the results from device memory to host memory

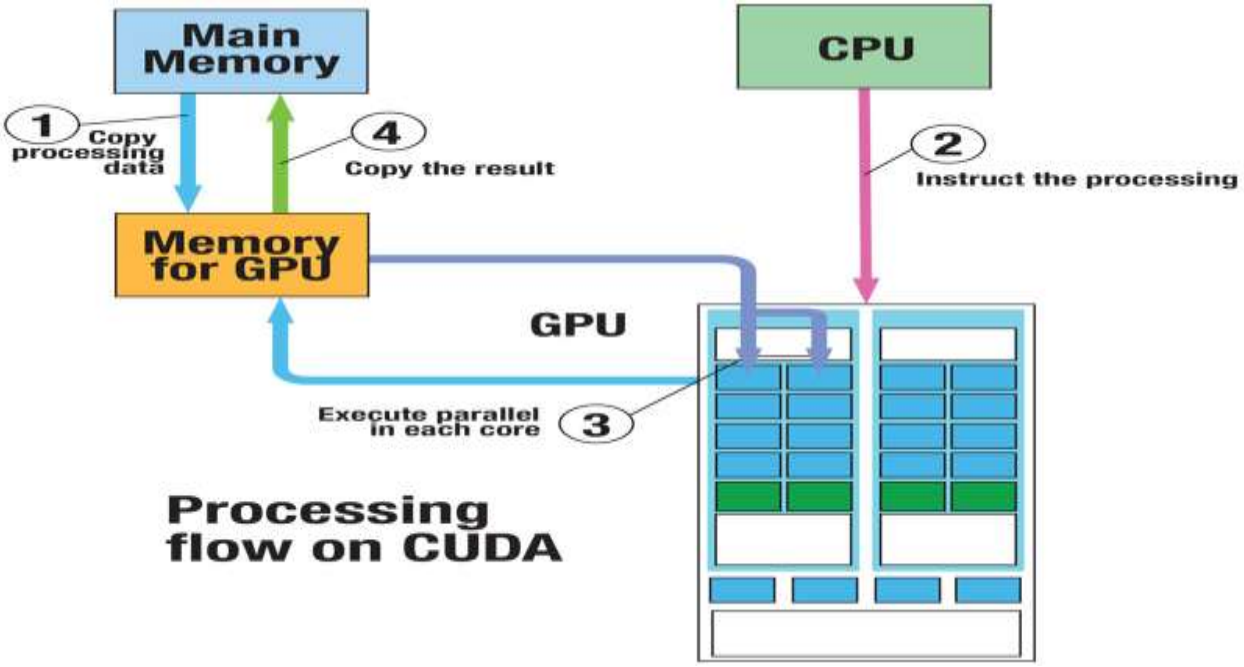

Figure 3 CUDA Processing

The communication between the CPU and GPU takes place via PCIe bus. The bandwidth of the PCIe bus is very important metric in performance evaluation as it can create a bottleneck during the data transfer phase.

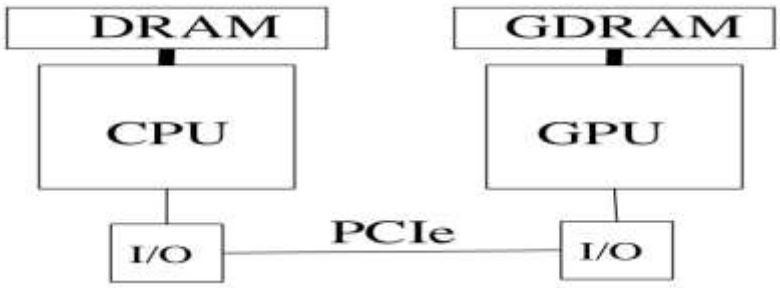

Figure 4 PCIe Communication

2.2.2 GPU BLOCKS AND THREADS

As said earlier, GPUs have millions of light weight threads which can be launched simultaneously and process data in parallel. Threads can be launched in different configurations (x,y,x) depending on the type of the data. For tasks related to 2D image processing or video frame processing threads can also be launched in a 2D configuration(x,y). In newer GPUs, a block of threads can have a maximum of 1024 threads launched simultaneously in different configurations like, (32,32), (16,16,4) and (8,8,8). The thread number must not increase the maximum thread limit. Moreover, up to (1024, 1024, 64) blocks can be launched on JETSON TK1. A conceptual of grid, blocks and threads is shown below:

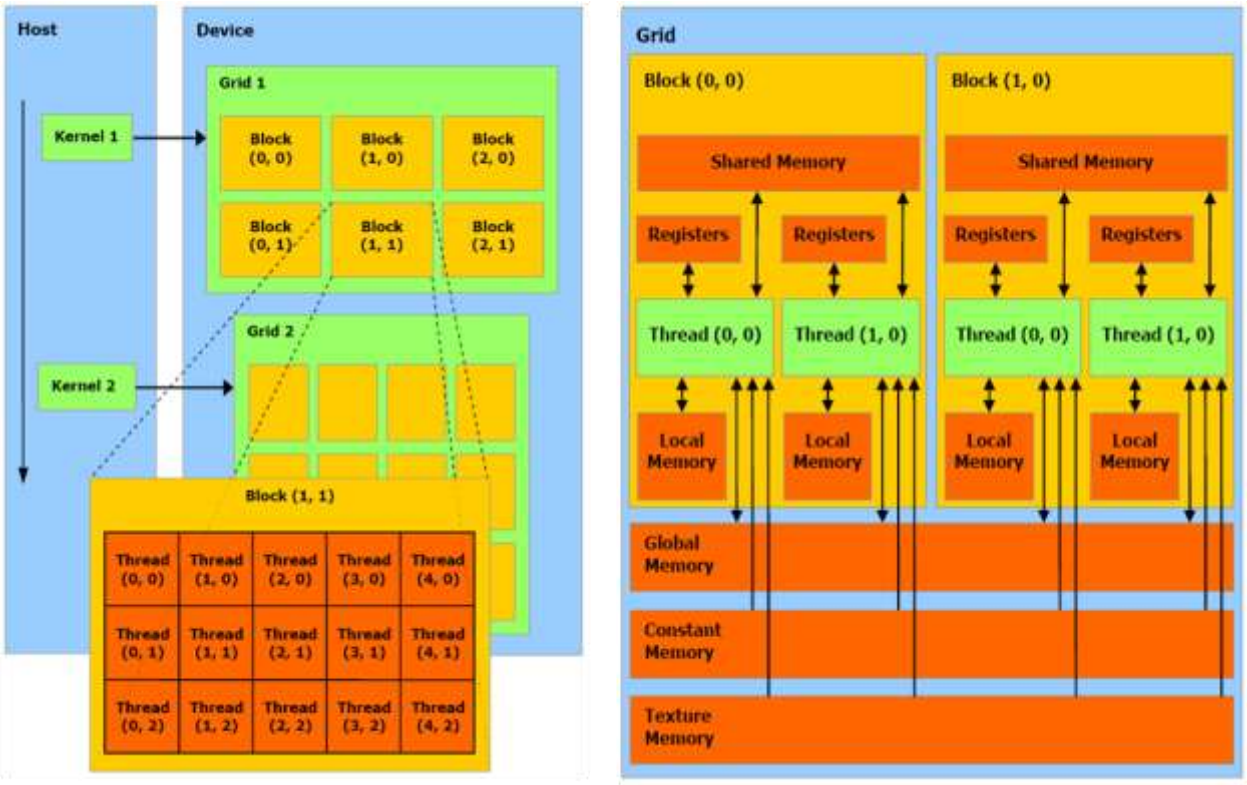

Figure 5 CUDA Blocks and Threads

CHAPTER 3

SYSTEM DESIGN

Considering the algorithm has to be eventually implemented onto GPU, a simpler but more robust approach has been taken to track the motion. This section describes the motion tracking algorithm that has been used in this work. A sequential approach shall be described first, and the algorithm shall be parallelized for GPU afterwards. After that, system level design is described.

3.1 ALGORITHM DESIGN

One approach would be to measure the background first and then use the averaged background frame subtraction to track the moving objects. But this method has the disadvantage that the slightest change in the camera position would completely destroy the algorithm. A small modification has been made, which is to take current back ground frame each time and do the frame subtraction.

3.1.1 SEQUENTIAL ALGORITHM

While Video Available do:

Capture Video Input

Capture Frame 1

Capture Frame 2

For each pixel of frames do:

Subtracted Frame = Frame 2 – Frame 1

For each pixel of Subtracted Frame do:

BGR to Gray Conversion

For each pixel of Gray Frame do:

Gray to Binary Conversion

For each pixel of Binary Frame do:

Erode 3 X 3

For i= 1 to 4 do:

For each pixel of Eroded Frame do:

Dilate 3 X 3

Find Contours to Dilated Frame

Draw Rectangles with Contours

Repeat

Let us discuss what each algorithm step would do.

1. Subtracting two consecutive frames would capture any immediate movements taking place at higher speeds. This approach allows us to change the position of camera at any time and start tracking on newer scenes. This step also increases the sensitivity and introduces noise because of illumination changes in the environment. The subtraction takes place on BGR frames and subtracts each blue, green and red component of the corresponding frames.
2. The second step is simple; iterates over the subtracted frame and convert it to grayscale for further processing.
3. In the third step, we threshold the grayscale frame and convert it to a binary frame. This allows us to apply morphological image processing techniques on the image.
4. Erosion filter is applied to remove the noise. Erosion can be applied multiple times depending upon the sizes of the unwanted signals.
5. Dilation operation is needed to expand the size of the moving objects; it also helps connecting the distorted patches of the object. This would eventually create a blob of the moving object.
6. After that contours are drawn around the blob.
7. In the final step, we draw rectangle shapes around the drawn contours.

Here is a flow chart of the algorithm:

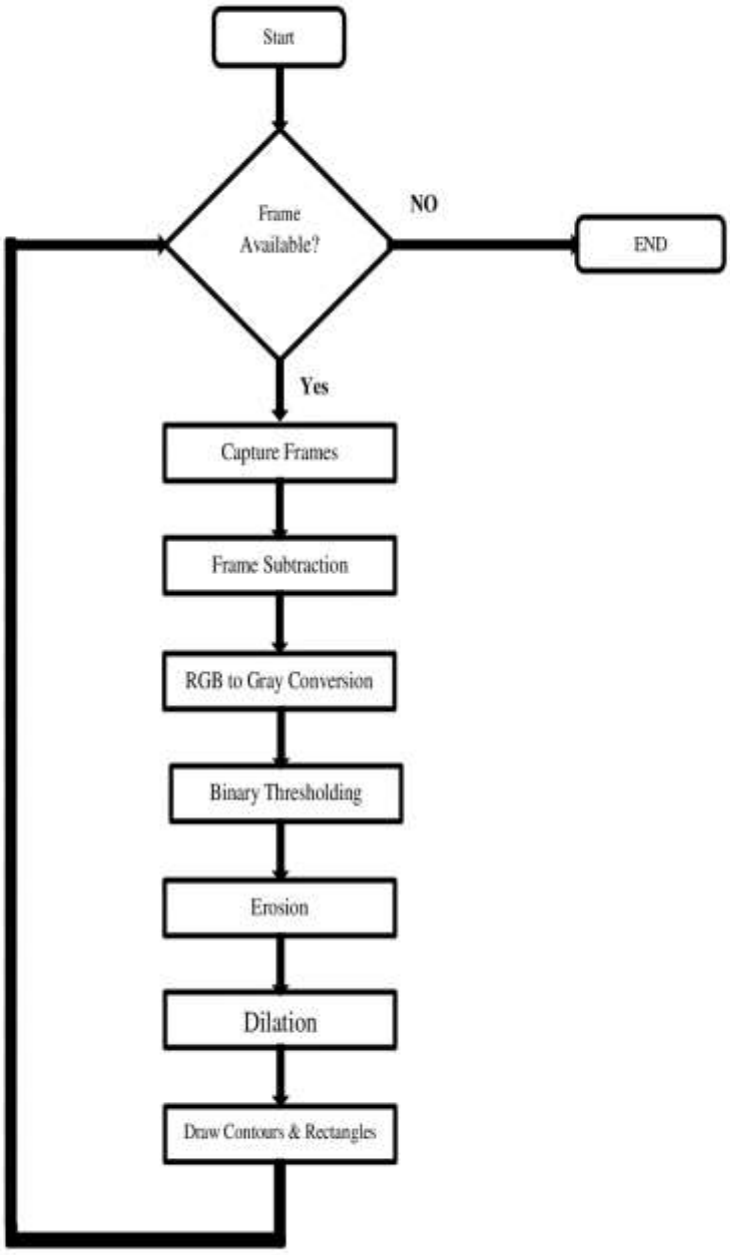

Figure 6 Sequential Algorithm

3.1.2 PARALLEL ALGORITHM

The purely sequential algorithm has been parallelized in a very simple fashion. The algorithm is a hybrid of sequential and parallel tasks. The computationally intensive portion of the algorithm has been parallelized to be offloaded onto the GPU. The simpler tasks have been left sequential.

While Video Available do:

Capture Video Input

Capture Frame 1

Capture Frame 2

Allocate GPU Memory and Send Frames for Processing

For each pixel of frames do in parallel:

Subtracted Frame = Frame 2 – Frame 1

For each pixel of Subtracted Frame do in parallel:

BGR to Gray Conversion

For each pixel of Gray Frame do in parallel:

Gray to Binary Conversion

For each pixel of Binary Frame do in parallel:

Erode 3 X 3

For i= 1 to 4 do:

For each pixel of Eroded Frame do in parallel:

Dilate 3 X 3

Send processed frame back to CPU

Find Contours to Dilated Frame

Draw Rectangles with Contours

Repeat

The algorithm itself is purely the same for the CPU and the GPU, but the looping structures in the sequential algorithm have been completely parallelized. The GPU would process each pixel of the frame in parallel. The GPU might be slower in terms of processing one pixel when compared to CPU, but when such large number of pixels is considered being processed in parallel, GPU has a fairly high throughput. Here is a flow chart of the modified algorithm:

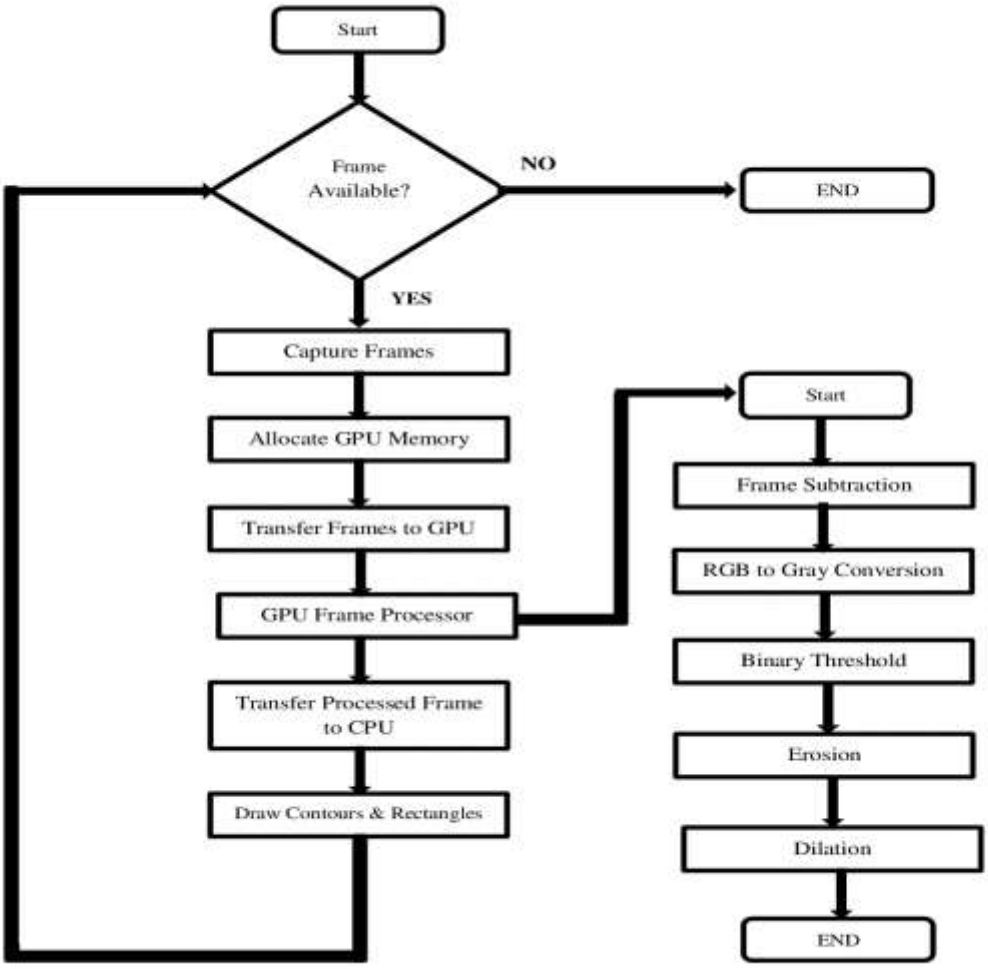

Figure 7 Parallel Algorithm

3.2 SYSTEM LEVEL DESIGN

CPU based system level design is fairly simple and is described first, after that GPU based design is discussed.

3.2.1 CPU SYSTEM LEVEL DESIGN

CPU based system have the following major components:

- Camera Input Capture Module
- Frame Processor Module
- Contour Finder and Rectangle Drawer Module
- Tracked Motion Display Module

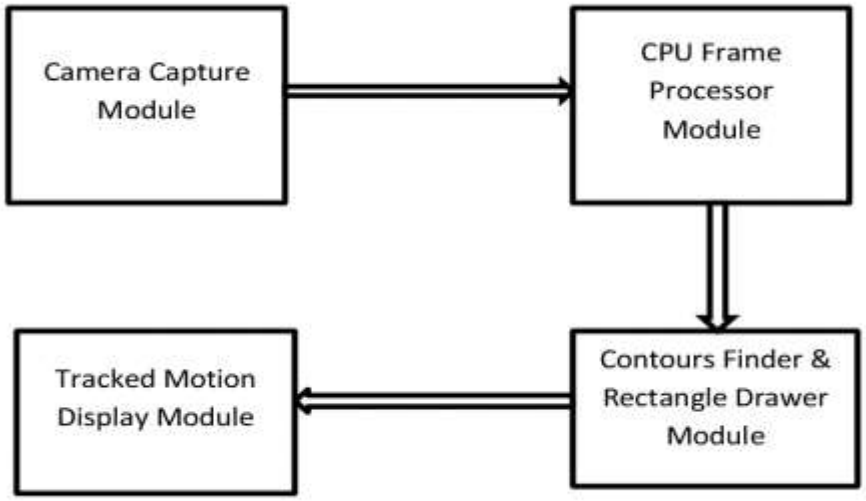

Figure 8 CPU based System

Camera Input Capture Module is responsible for capturing videos from single or multiple cameras. It fetches the video frame by frame and passes each frame to Frame Processor for further processing.

Frame Processor Module runs the actual algorithm comprising of frame subtraction, RGB to Gray conversion, binary thresholding, erosion and dilation operations. It then passes the processed frame to the next module.

Contour Finder and Rectangle Drawer Module takes the processed frame and finds contours around the moving objects, finally draws rectangles around them.

Tracked Motion Display module displays the final output of the tracked motion.

3.2.2 GPU SYSTEM LEVE DESIGN

The GPU based system design comprises of both the CPU and GPU based modules and is a heterogeneous system. Some of the modules are same as in CPU based system.

- Camera Input Capture Module
- GPU Resource Manager Module
- GPU Frame Processor
- Contour Finder and Rectangle Drawer Module
- Tracked Motion Display Module

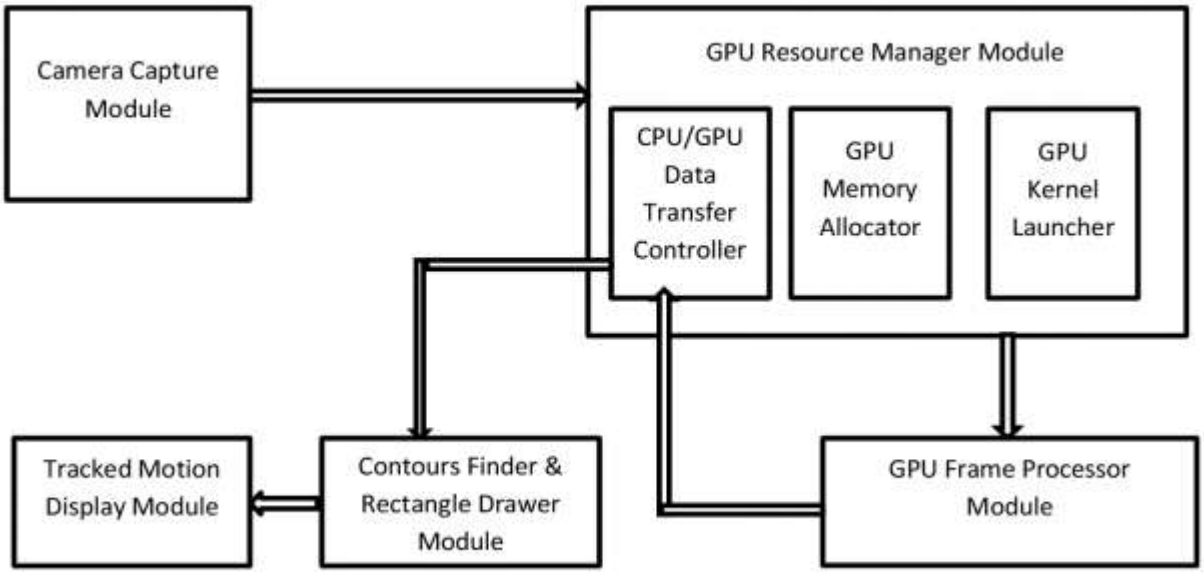

Figure 9 GPU based System

Camera Input Capture Module is the same and the CPU is responsible for capturing video frames. It then sends captured frames to GPU Resource Manager.

GPU Resource Manager is the CPU based host for managing all the GPU related activities. This module further comprises of three modules:

1. GPU Memory allocator allocates the memory onto GPU.
2. CPU/GPU Data Transfer Controller is responsible for transferring data from host (CPU) to the (Device) and device to host.
3. GPU Kernel Launcher launches the actual function that has to run on GPU. In this case, it is the GPU Frame Processor.

GPU Frame Processor is the actual function that runs entirely onto GPU and is responsible for all the speed gains. It runs the same algorithm as CPU but does that in an entirely parallel fashion. It then sends the processed frame to GPU Resource Manager which sends it back to the CPU.

Contour Finder and Rectangle drawer runs on CPU and does the same job.

Tracked Motion Display Module is the same as was in CPU's case.

CHAPTER 4

SOFTWARE DEVELOPMENT

This chapter starts with the introduction of the development environment and then discusses complete software development cycle for the motion tracking problem. The JETSON TK1 development platform is discussed first. Then, CPU based sequential algorithm implementation using the open source OpenCV library is described. After that, CUDA C APIs are discussed with example programs. Finally, we inTEGRAte OpenCV with CUDA C to accelerate the motion tracking algorithm onto GPU.

4.1 DEVELOPMENT ENVIRONMENT

In April 2014, NVIDIA launched JETSON TK1, a fully featured development platform which had all the features as in a Raspberry PI board but also some PC oriented features i.e. SATA, mini-PCIe and a cooling fan. JETSON TK1 comprises of TEGRA SOC with the same architecture and features as any desktop GPU but draws a very low power like a mobile chip. Therefore, TEGRA can use the same CUDA code as a desktop GPU and can produce similar performance gains. A few key features of the TEGRA K1 SOC are:

- A Quad Core ARM Cortex A15 CPU.
- A 192 CUDA Core KEPLER GPU.
- Dual ISP Core that delivers 1.2 Giga Pixels per second of processing power.
- Advance Display Engine that is capable of simultaneously driving both local and external display.
- Built on the TSMC 28nm HPM process to deliver excellent performance.

The KEPLER GPU in TEGRA K1 is built on the same high performance, energy efficient architecture that is found in GEFORCE, QUADRO, and TESLA GPUs for graphics and computing. TEGRA K1 is the only mobile processor that supports CUDA 6 for computing and full desktop OpenGL 4.4 for graphics.

The JETSON TK1 Development Kit runs Linux for TEGRA (L4T), a modified Ubuntu 14.04 Linux distribution provided by NVIDIA. The software provided by NVIDIA includes the Board Support Package (BSP) and the software stack that includes CUDA 6 Toolkit, OpenGL 4.4 drivers and the NVIDIA VisionWorks Toolkit.

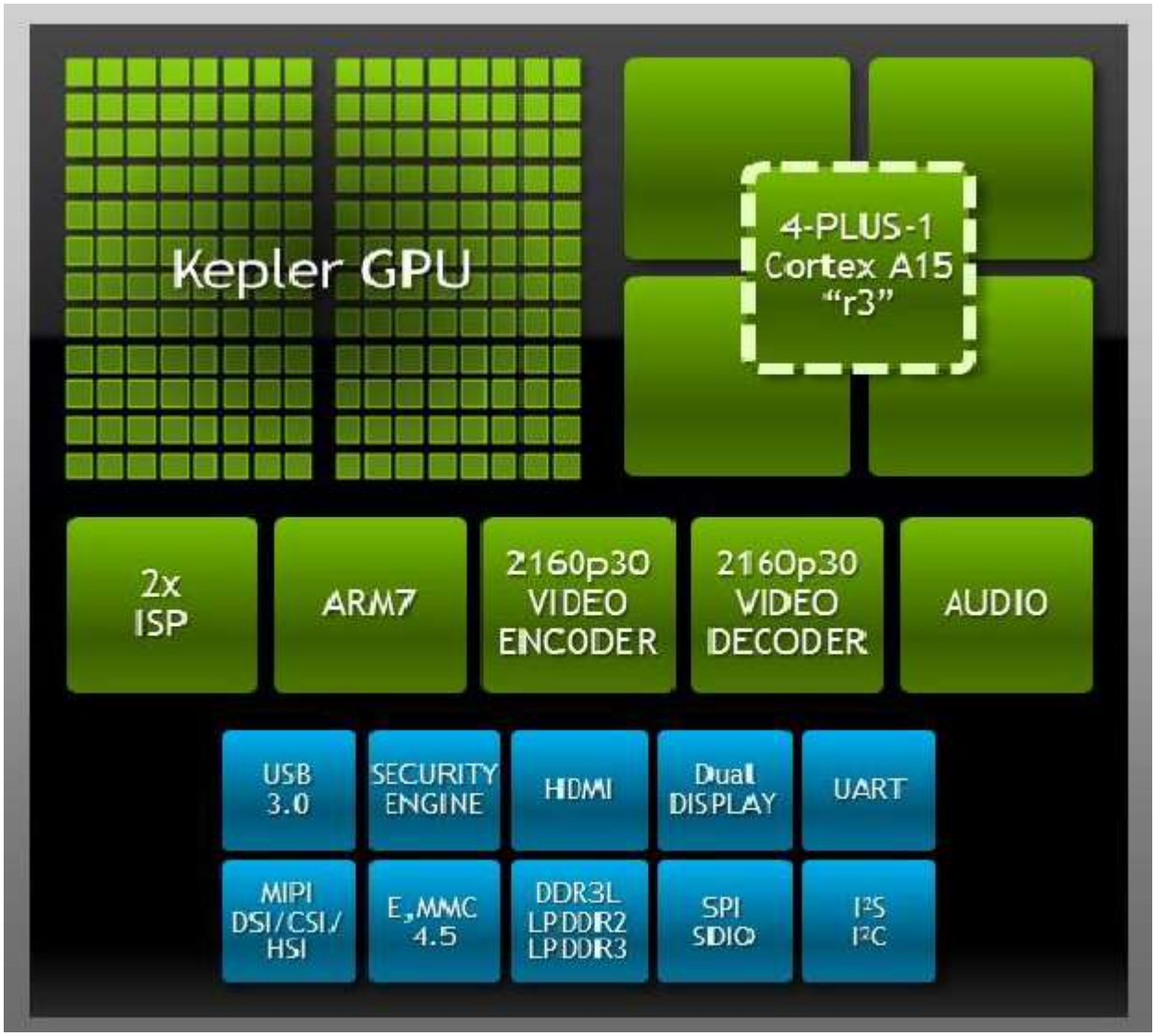

Figure 10 TEGRA K1

The architecture of the KEPLER GPU in TEGRA K1 is identical to the KEPLER GPU architecture used in high-end systems, but includes optimizations for mobile system usage to conserve power and deliver industry-leading mobile GPU performance. While the highest-end KEPLER GPUs in desktop, workstation, and supercomputers include up to 2880 single-precision

floating point CUDA cores and consume a few hundred watts of power, the JETSON TK1 platform with TEGRA K1 includes 192 CUDA cores and consumes significantly lower power.

Note that the TEGRA K1 KEPLER GPU has more cores than many entry-level to mainstream desktop GPUs of just a few years ago. Delivering higher power efficiency and much lower platform power consumption, the JETSON TK1 platform is ideally suited for applications in the embedded space that require exceptional power efficiency, low thermal dissipation and significantly higher performance than current FPGA and x86-based embedded solutions.

When TEGRA TK1 is compared to latest mobile processors, it delivers almost 2.5X the peak performance of competing mobile processors. When limited to match the power consumption of competing mobile processors, TEGRA K1 delivers almost 50% higher performances per Watt.

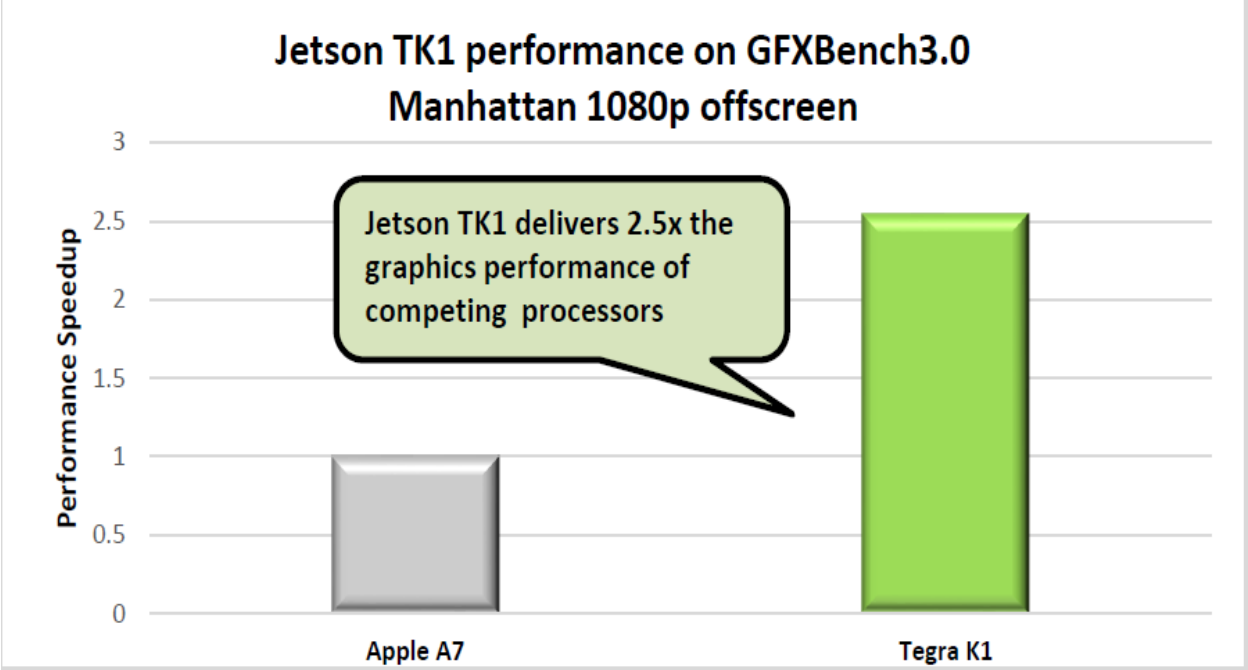

Figure 11 TK1's 2.5X performance

The NVIDIA JETSON TK1 Development kit is the world's first mobile supercomputer for embedded systems and opens the door for embedded system designs to harness the power of GPU-accelerated computing. JETSON TK1 will enable a new generation of applications for computer vision, robotics, medical imaging, automotive, and many other areas.

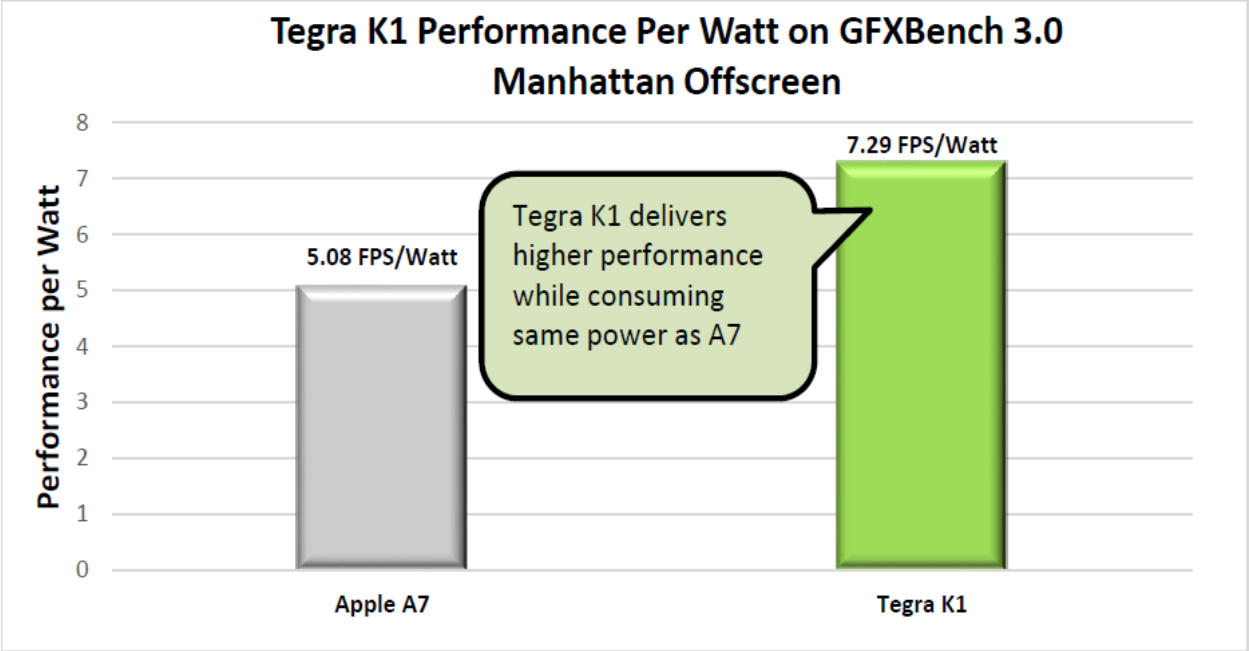

Figure 12 TEGRA K1 Performance Per Watt

Powered by the revolutionary 192-core NVIDIA TEGRA K1 mobile processor, the JETSON platform delivers over 300 GFLOPS of performance that is almost three times more than any similar embedded platform. The fully programmable 192 CUDA cores in TEGRA K1 along with CUDA 6 Toolkit support makes programming on JETSON TK1 much easier than on FPGA, Custom ASIC, and DSP processors that are commonly used in current embedded systems. The CUDA programming model is used by over 100,000 developers at over 8,000 institutions worldwide. The JETSON TK1 Developer Kit comes with full support of the CUDA 6.0 developer tool suite, including debuggers and profilers, to allow development of powerful embedded applications.

JETSON TK1 fast tracks embedded computing into a future where machines interact and adapt to their environments in real time, and deliver whole new experiences in various fields such as robotics, augmented reality, computational photography, human-computer interface, and advanced driver assistance systems.

JETSON TK1 is a full featured platform containing all the features as in a Raspberry PI board but also PC oriented features.

Standard ports on Jetson TK1 Development board
• 1 Half mini-PCIE slot
• 1 Full-size SD/MMC connector
• 1 Full-size HDMI port
• 1 USB 2.0 port, micro AB
• 1 USB 3.0 port, A
• 1 RS232 serial port
• 1 ALC5639 Realtek Audio Codec with Mic in and Line out
• 1 RTL8111GS Realtek GigE LAN
• 1 SATA data port
• SPI 4MByte boot flash
Additional ports available via Expansion port
• DP/LVDS
• Touch SPI
• 1x4 + 1x1 CSI-2
• GPIOs, UART, HSIC and I2C

Figure 13 JETSON Peripherals and IO Ports

The JETSON TK1 Development platform is primarily designed to enable the development of GPU-accelerated embedded applications and is not optimized to deliver the low power consumption required on mobile devices such as tablets and smart phones.

Being a development platform, the JETSON TK1 has numerous hardware, IO, and peripheral interfaces that add to the total platform power consumption. The core software package is optimized to deliver the performance required for embedded applications such as computer vision, robotics, and image processing. For example, mobile platforms that run on battery power may use power-efficient LPDDR3 memory, smart panel displays, PMICs, and other low power components. The JETSON development platform is designed to run on AC power and therefore

uses standard DDR3L memory, Ethernet adapters, standard HDMI, and other platform components that are not optimized for low power.

Due to these design choices, the JETSON TK1 development platform should not be used evaluate the power efficiency of the TEGRA K1 mobile processor for mobile implementations and applications. Developers and OEMs who want to power profile applications for mobile are encouraged to contact our NVIDIA Developer Relations team to request mobile platform support. In addition, the Developer Relations team can provide guidance for power efficiency optimization for mobile and embedded applications. The following table breaks down the JETSON TK1 platform power consumption at idle state and at peak performance.

Platform Component	Idle State Platform Power (milli-Watts)	Platform power when AP+DRAM is constrained to Apple A7 power when running GFXBench 3.0[4] (milli-Watts)	Platform power when delivering peak GFXBench 3.0 Performance(milli-Watts)
AP+DRAM[5]	660	3660[6]	6980
Power consumed by Platform components such as Fan, HDMI, PCIE, SATA and others	2080	2090	2090
SoC Voltage regulator Efficiency loss	280	960	1790
Total Power at DC input of the board	3020	6710	10860
AC to DC conversion loss (15%)[7]	450	1180	1630
Power consumed at AC outlet[8]	3470	7890	12490

Figure 14 TK1 Power

JETSON TK1 comes pre installed with a modified version of Linux UBUNTU 14.04 called L4T (Linux for TEGRA) with hardware drivers already configured. It can be used as a fully featured development environment by installing CUDA Toolkit, OpenCV and other open source software. Cross development option is also there but JETSON TK1's Quad Core processor is pretty fast at compiling the code natively.

4.2 OPENCV IMPLEMENTATION

OpenCV is an open source computer vision library aimed at developing real time vision related applications. OpenCV is written in C++, its primary interface is in C++ and it can run on various platforms such as Linux, Windows and Mac operating systems. Currently, it contains over 2500 vision related functions readily available to be deployed in academic research or even commercial projects for free.

http://elinux.org/JETSON/Installing_OpenCV is a complete guide on installing the OpenCV library for JETSON TK1 board and also includes building and running some OpenCV samples.

4.2.1 BASIC IMAGE/VIDEO OPERATIONS

In this section, we describe the basic image and video read/display tasks using the OpenCV C++ functions.

4.2.1.1 DISPLAYING AN IMAGE

Here is a simple example code for reading and displaying an image using the OpenCV library functions.

```
#include<opencv2/highgui/highgui.hpp>

int main(int argc, char** argv)
{
      cv::Mat img = cv::imread(argv[1],-1);

      if(img.empty()) return -1;

      cv::namedWindow("Image Display",cv::WINDOW_AUTOSIZE);

      cv::imshow("Image Display",img);

      cv::waitKey(0);
```

```
    cv::destroyWindow("Image Display");
}
```

Here is an explanation of this code snippet:

- 'highgui' is a pre-built graphical user interface library and contains functions such as reading and writing image/video files.
- `cv::Mat img = cv::imread(argv[1],-1);` the Mat data type is the most frequently used data type in OpenCV and is used for holding images/video frames information.
- `cv::namedWindow("Image Display",cv::WINDOW_AUTOSIZE);` creates a window to display the read image.
- `cv::imshow("Image Display",img);` displays the image.
- `cv::waitKey(0);` waits for the user to hit any key and finish the display.
- `cv::destroyWindow("Image Display");` destroys the CV windo object.

4.2.1.2 READING CAMERA INPUTS

Reading and displaying a video file is no different than reading and displaying an image. The additional thing is that we creat a loop equal to the video file length and read and display each individual frame just like an image. Here is the code snippet:

```
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>

using namespace std;

int main(int argc, char** argv)
{

    cv::VideoCapture capA;
    cv::VideoCapture capB;
```

```
        capA.open(0);
        capB.open(1);
        cv:: Mat frameA;
        cv:: Mat frameB;

        while(1)
        {
              capA>>frameA;
              if(!frameA.data) break;

              capB>>frameB;
              if(!frameB.data) break;

              cv::imshow("CAM A",frameA);
              cv::imshow("CAM B",frameB);

              if(cv::waitKey(33)>=0) break;

        }

return 0;
}
```

- `cv::VideoCapture capA;` VideoCapture object is responsible for capturing and holding video frames. In this code, we have two VideoCapture objects for holding two video frames from two cameras simultaneously.
- `capA.open(0);` provides the opening of the video stream from camera device named 0.
- After that, we enter into a while loop which reads the video frame by frame and displays it as long as the input frame is available.

4.2.2 OPENCV MOTION TRACKING IMPLEMENTATION

Now we'll move to our actual sequential motion tracking algorithm implementation. From our discussion about the sequential tracking algorithm in the previous chapter, recall that we need the following major imaging operations:

1. Frame Subtraction
2. RGB to GRAY Conversion
3. Binary Thresholding
4. Erosion
5. Dilation
6. Contour Detection
7. Rectangle Drawing

This section would describe the complete implementation of the above mentioned tasks. We have divided our codes in three main parts:

1. Main Function
2. Frame Processor
3. Contour Finder and Rectangle Drawer

4.2.2.1 MAIN FUNCTION

The code for main function is given below. Main function primarily deals with the video acquisition, handling sub functions, measuring frame processing time and finally displaying the tracked motion. FrameProcessor and DrawContour are the subfunctions. The main function grabes each frame and sends it to the FrameProcessor function for runing the main algorithm. After that, FrameProcessor sends back the processed frame to the main function and the main function sends this processed frame to the DrawContour function which is responsible for finding the contours around the moving objects and drawing rectangles around them. Let us take a look at the code.

```
int main(int argc, char** argv)
```

```
{

	cv::VideoCapture capA;
	capA.open(1);

	clock_t Time_Start, Time_End, Time_Difference;
	double Time;

	Mat InputA;
	Mat InputB;

	Mat FrameFinalA;

	for(;;)
	{

	capA>>InputA;
	capA>>InputB;

	Time_Start=clock();

	FrameProcessor(InputA,InputB,FrameFinalA);

	Time_End=clock();
	Time_Difference=Time_End-Time_Start;
	Time=Time_Difference/(double)CLOCKS_PER_SEC ;
	printf ("CPU Frame Rate = %f FPS\n",1/Time);

	DrawContour(FrameFinalA,InputA);

	imshow( "CPU Tracking", InputA);
```

```
        if(waitKey(30)>=0) break;
    }

    return 0;
}
```

Please note that we have only a single VideoCapture object which grabs two consecutive frames. These frames are then sent to the FrameProcessor function which is responsible for running the actual motion tracking pre processing. Below are the screen shots of the original running video from a fixed camera over a highway capturing moving vehicles.

Figure 15 Highway Video 1

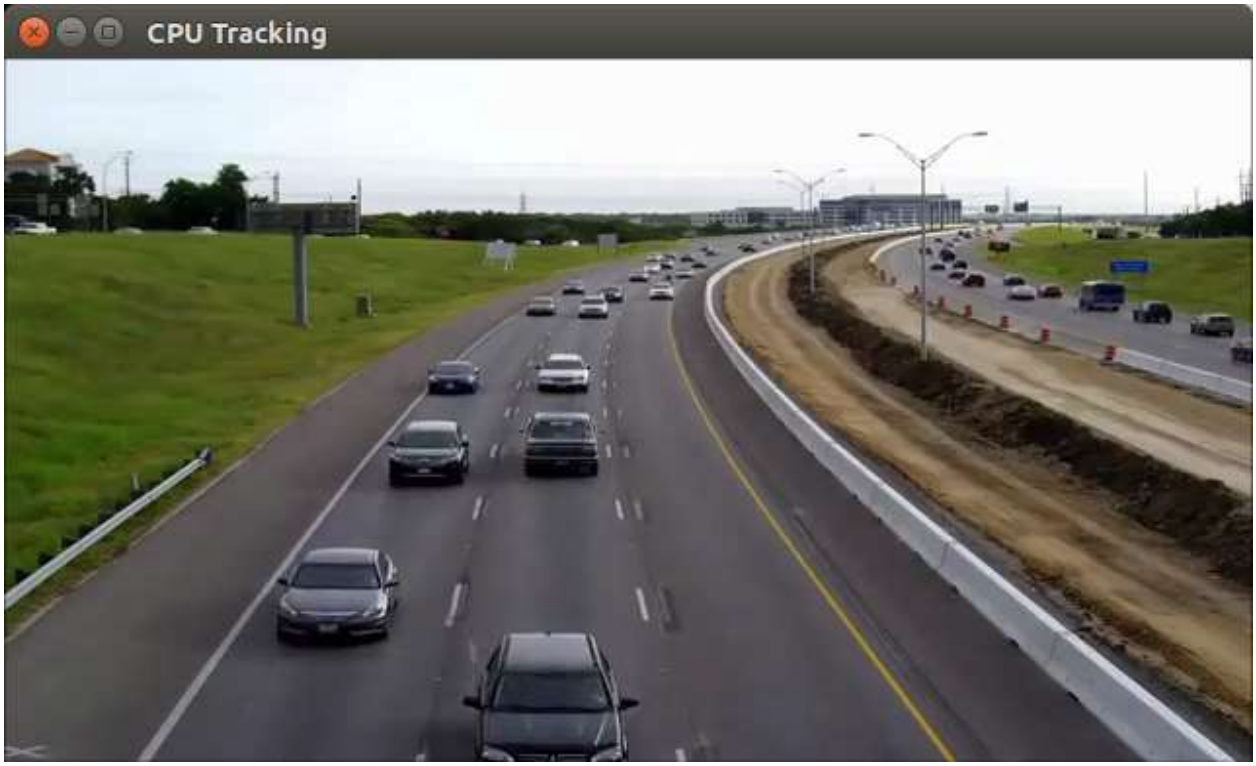

Figure 16 Highway Video 2

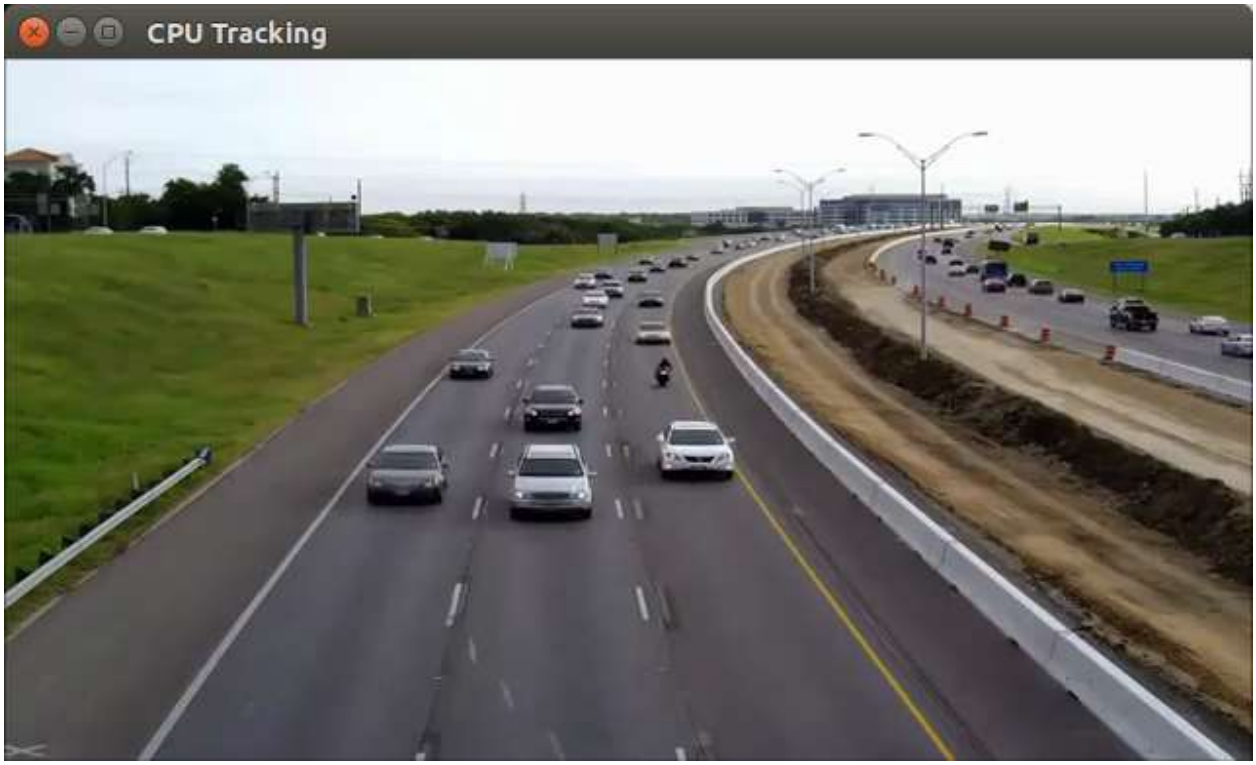

Figure 17 Highway Video 3

4.2.2.2 FRAME PROCESSOR

The frame processor module takes two consecutive frames as an input does the below mentioned operations and returns the processed frame:

- Frame subtraction of two consecutive frames.
- BGR to Gray conversion.
- Binary Thresholding
- Erosion
- Dilation

```
void FrameProcessor(const cv::Mat& InputA, const cv::Mat& InputB,
cv::Mat& FrameFinal)
{

      int erosion_size=0;
      int d_size=2;

      FrameFinal=InputA-InputB;

      cvtColor(FrameFinal,FrameFinal,CV_BGR2GRAY);

      threshold(FrameFinal,FrameFinal,100,255,THRESH_BINARY);

      Mat element =

getStructuringElement(MORPH_RECT,Size(2*erosion_size+1,2*erosion_size+
1),
      Point(erosion_size,erosion_size));
```

```
        erode(FrameFinal,FrameFinal,element);
        erode(FrameFinal,FrameFinal,element);
        erode(FrameFinal,FrameFinal,element);

        Mat element1 =
getStructuringElement(MORPH_RECT,Size(3*d_size+1,3*d_size+1),

        Point(d_size,d_size));

        dilate(FrameFinal,FrameFinal,element1);
        dilate(FrameFinal,FrameFinal,element1);
        dilate(FrameFinal,FrameFinal,element1);
        dilate(FrameFinal,FrameFinal,element1);
        dilate(FrameFinal,FrameFinal,element1);
        dilate(FrameFinal,FrameFinal,element1);
        dilate(FrameFinal,FrameFinal,element1);

}
```

FrameProcessor module is failry simple.

- It first initializes the kernel sizes for erosion and dilation operators.
- `FrameFinal=InputA-InputB;` subtracts the two consecutive frames. The screenshots for subtracted frames are given below:

Figure 18 Subtracted Frame 1

Figure 19 Subtracted Frame 2

- `cvtColor(FrameFinal,FrameFinal,CV_BGR2GRAY);` converts the the frame from BGR to Grayscale.

Figure 20 Gray Frame 1

Figure 21 Gray Frame 2

Figure 22 Gray Frame 3

Figure 23 Gray Frame 4

- `threshold(FrameFinal,FrameFinal,100,255,THRESH_BINARY);` converts the grayscale image to binary image so it may be applied with morphological operators.

Figure 24 Binary Frame 1

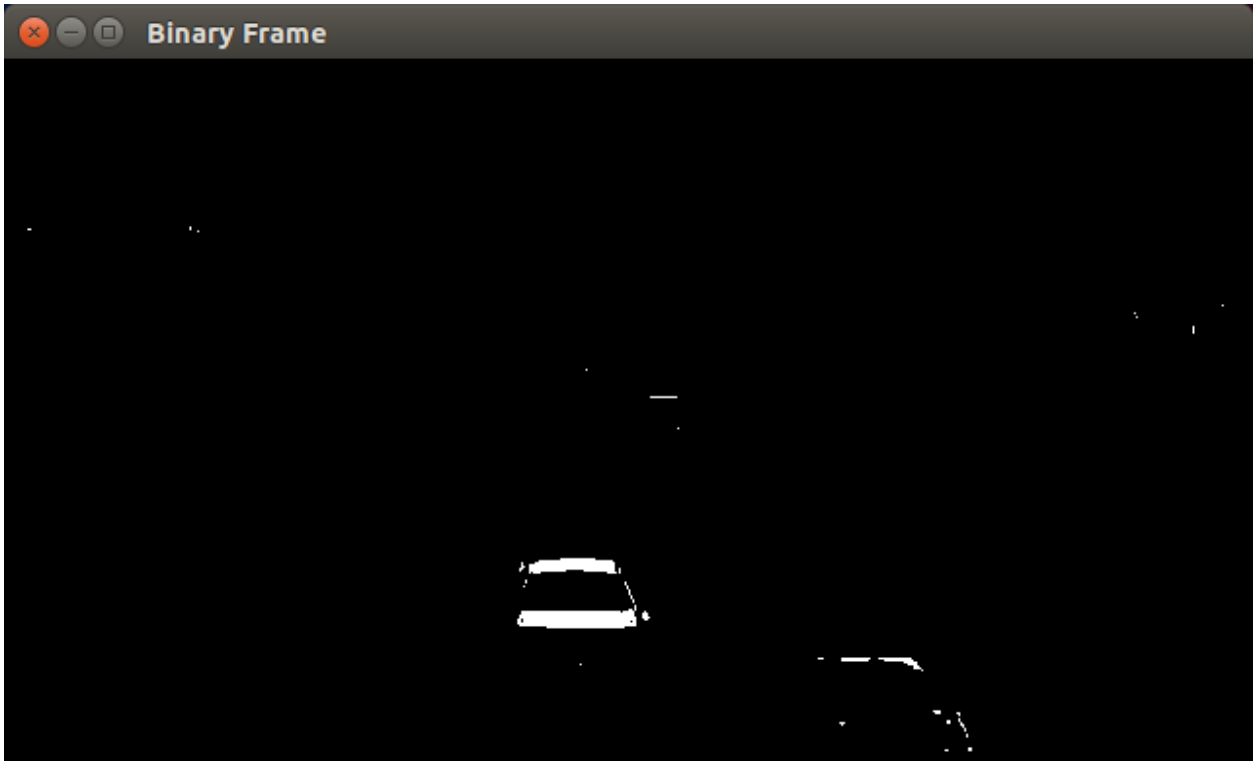

Figure 25 Binary Frame 2

Figure 26 Binary Frame 3

- `Mat element = getStructuringElement(MORPH_RECT,Size(2*erosion_size+1,2*erosion_size+1), Point(erosion_size,erosion_size));` defines the binary mask to be applied on to binary image. In this case, we have a rectangular shaped mask of size 2 X 2.
- `erode(FrameFinal,FrameFinal,element);` the erode function applies the erosion filter on the entire image.

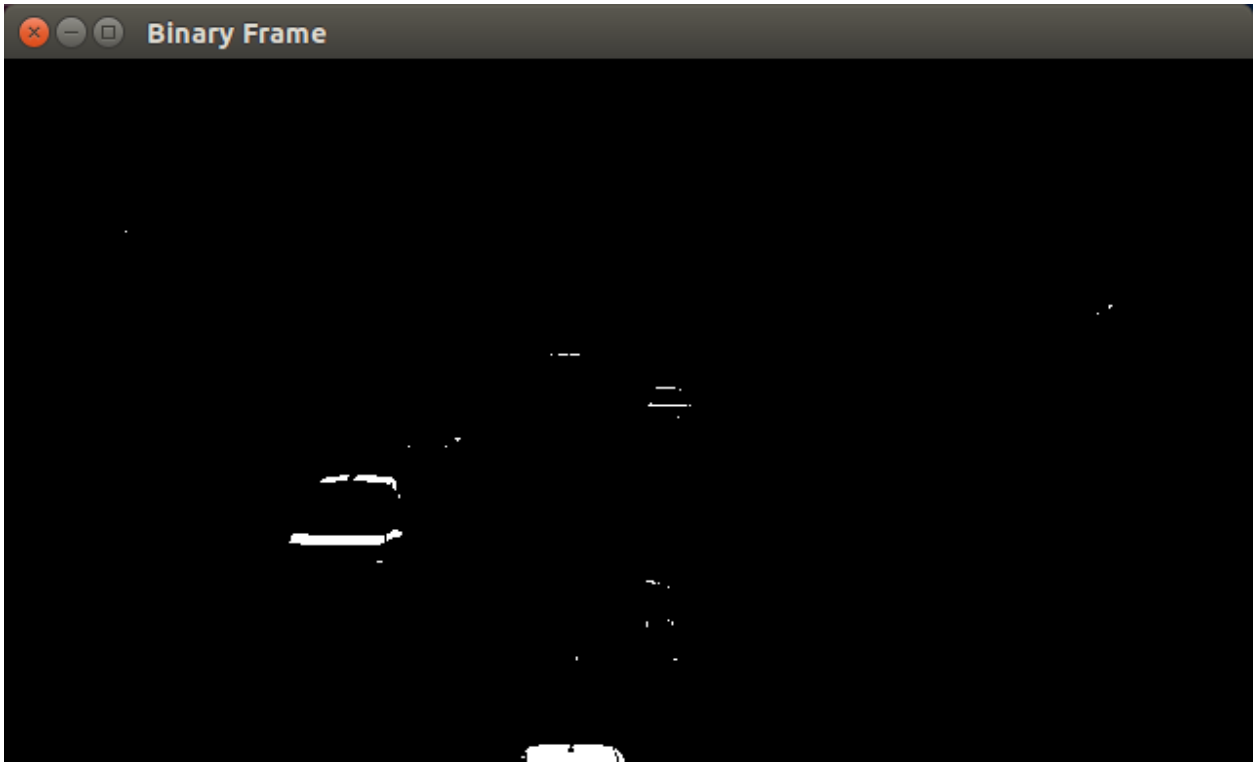

Figure 27 Eroded Frame 1

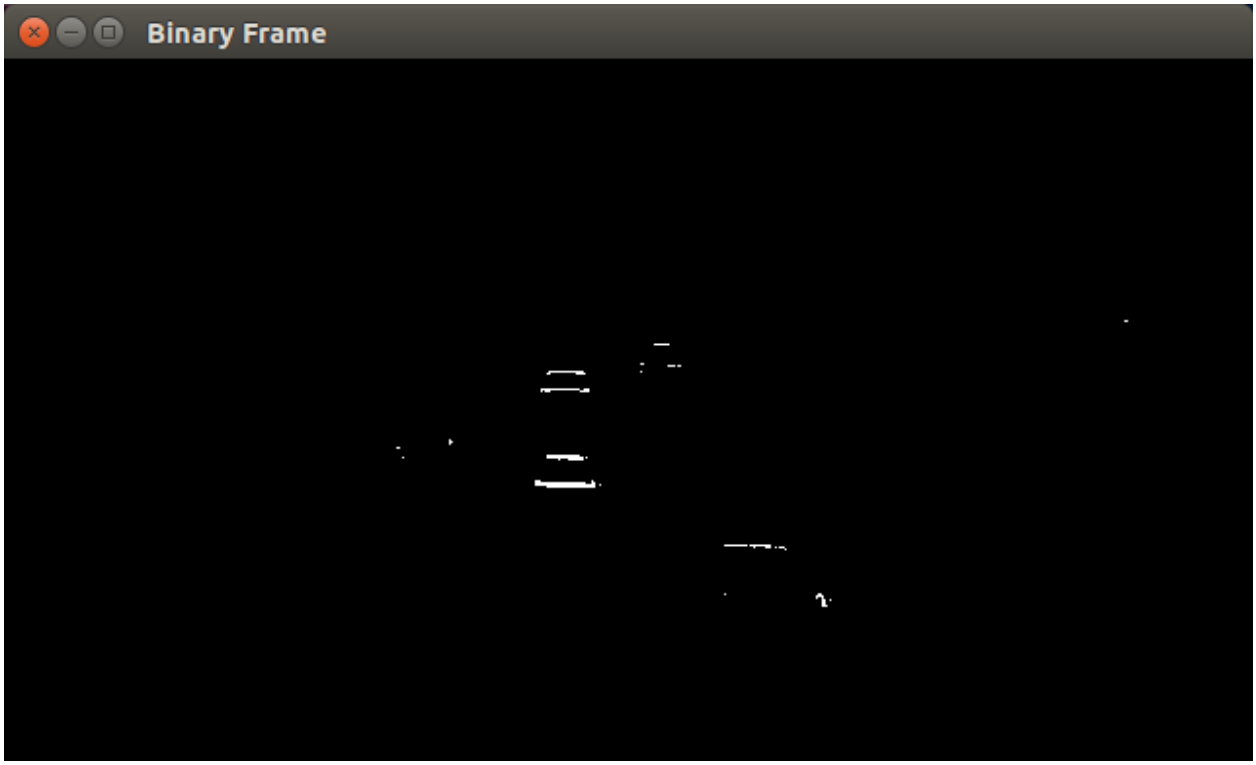

Figure 28 Eroded Frame 2

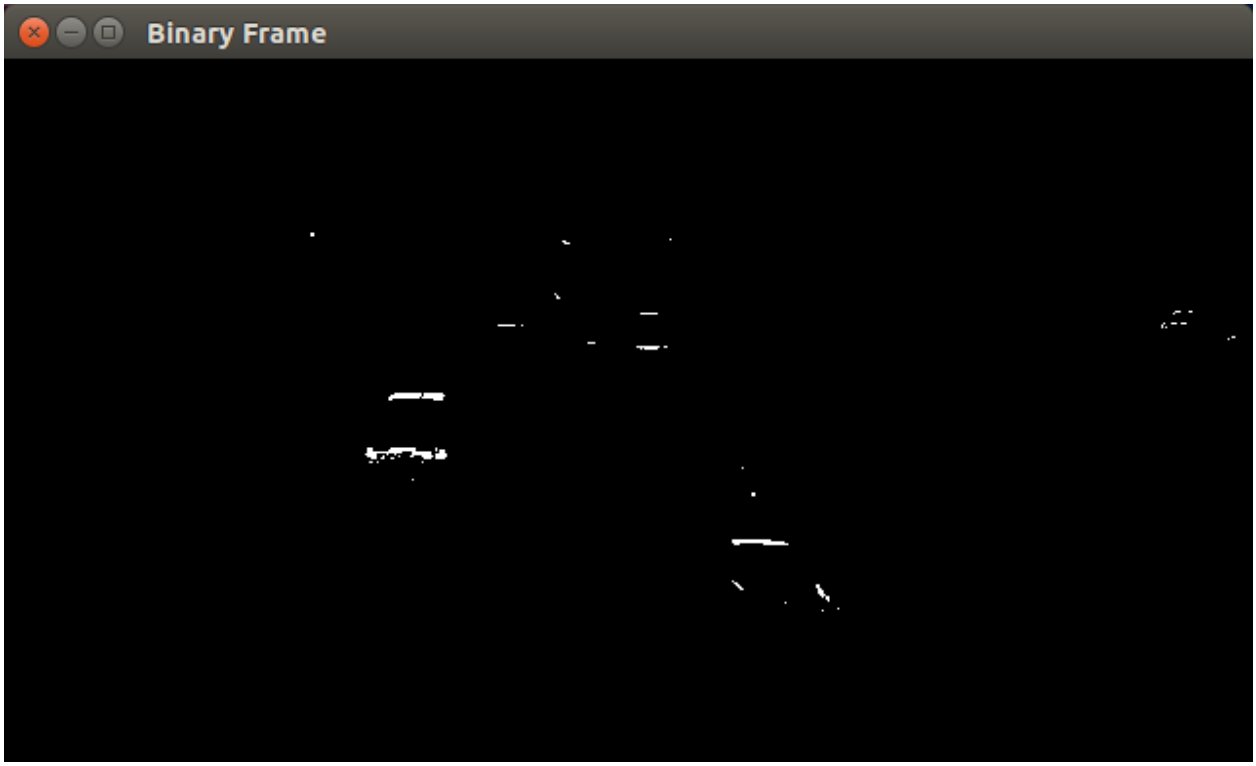

Figure 29 Eroded Frame 3

- `Mat element1 = getStructuringElement(MORPH_RECT,Size(3*d_size+1,3*d_size+1), Point(d_size,d_size));` defines the dilation mask to be applied. In this case, we have a rectangular mask of size 9 X 9.
- `dilate(FrameFinal,FrameFinal,element1);` dilate applies dilation mask to the entire frame.

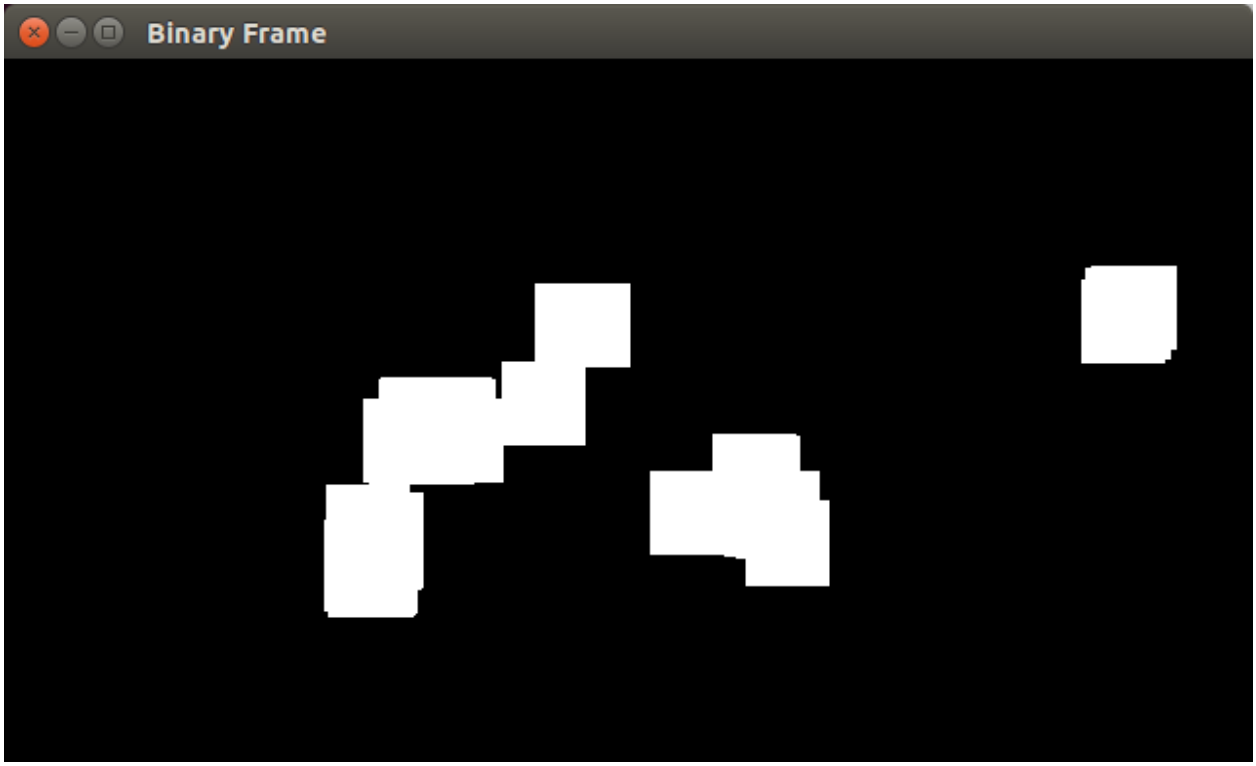

Figure 30 Dilated Frame 1

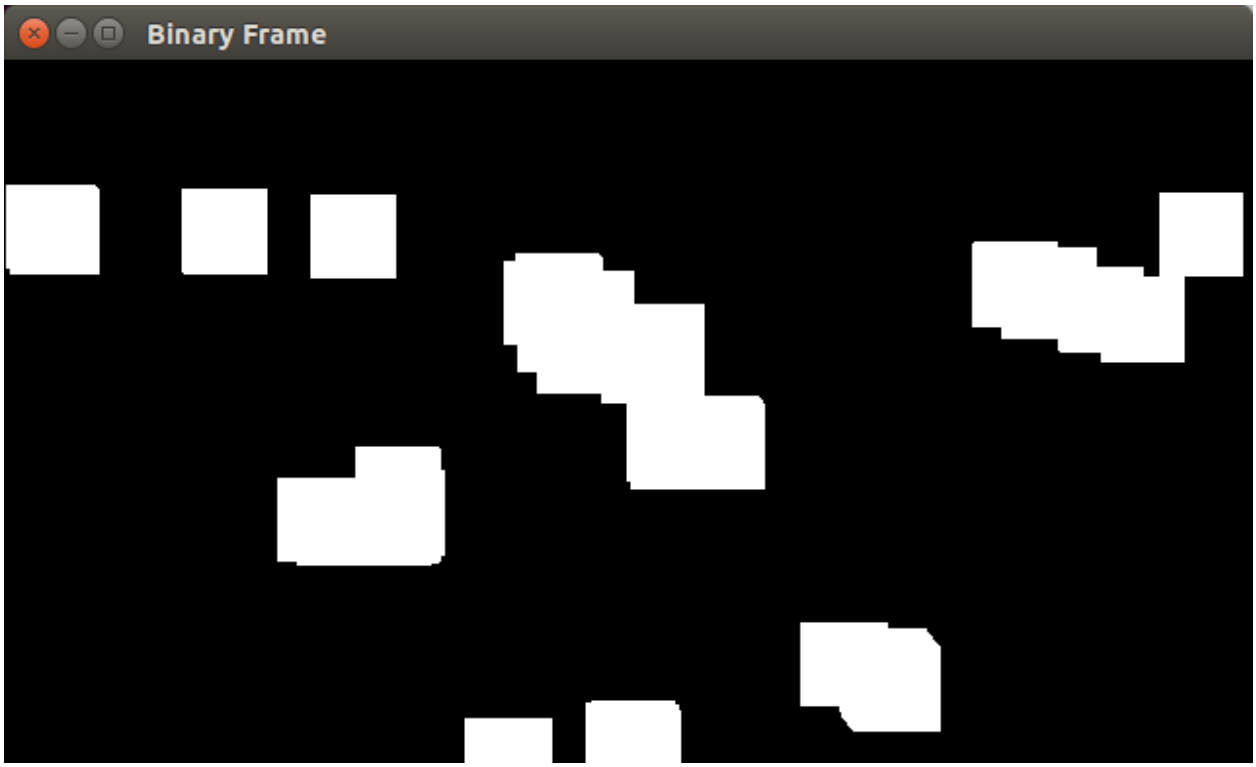

Figure 31 Dilated Frame 2

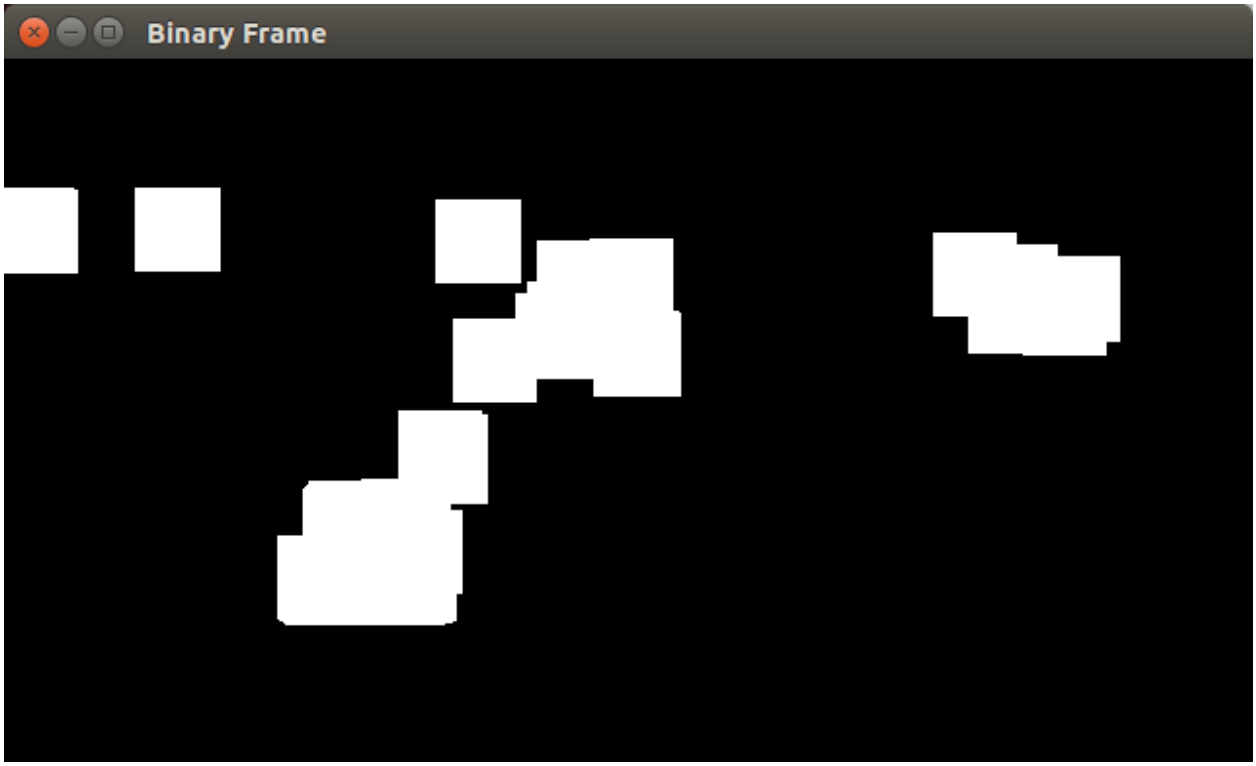

Figure 32 Dilated Frame 3

- After all these operations, the processed frame is sent back to the main function for further processing.

The main function then sends the processed frame to DrawContour function that uses OpenCV built in functions to find contours in the processed frame. After finding contours, the DrawContour function draws rectangles around the moving objects now visible as contours.

4.2.2.3 CONTOUR FINDER

The DrawContour function takes in the processed frame and performs the following operations:

- Find Contours
- Draw Rectangles

Here is the code for this function:

```
void DrawContour(const cv::Mat& FrameFinal, cv::Mat& Input)
{
          RNG rng(12345);
          vector<vector<Point> >contours;
          vector<Vec4i>hierarchy;


     findContours(FrameFinal,contours,hierarchy,CV_RETR_TREE,CV_CHAIN_
APPROX_SIMPLE,Point(0, 0));

          vector<vector<Point> > contours_poly(contours.size());
          vector<Rect>boundRect(contours.size());
          vector<Point2f>center(contours.size());
          vector<float>radius(contours.size());

          for(int i=0;i<contours.size();i++)
               {

     approxPolyDP(Mat(contours[i]),contours_poly[i],3,true);
                    boundRect[i]=boundingRect(Mat(contours_poly[i]));

     minEnclosingCircle((Mat)contours_poly[i],center[i],radius[i]);
               }


          for(int i=0;i<contours.size();i++)
               {
                    Scalar color = Scalar(0,0,255);

     rectangle(Input,boundRect[i].tl(),boundRect[i].br(),color,2,8,0);
               }
```

```
}
```

This function takes in the processed frame and draws the rectangles on the original input video frames. This function has been kept to run onto CPU even when the main frame processing has been done onto GPU. First it finds the contours:

Figure 33 Contoured Frame 1

Figure 34 Contoured Frame 2

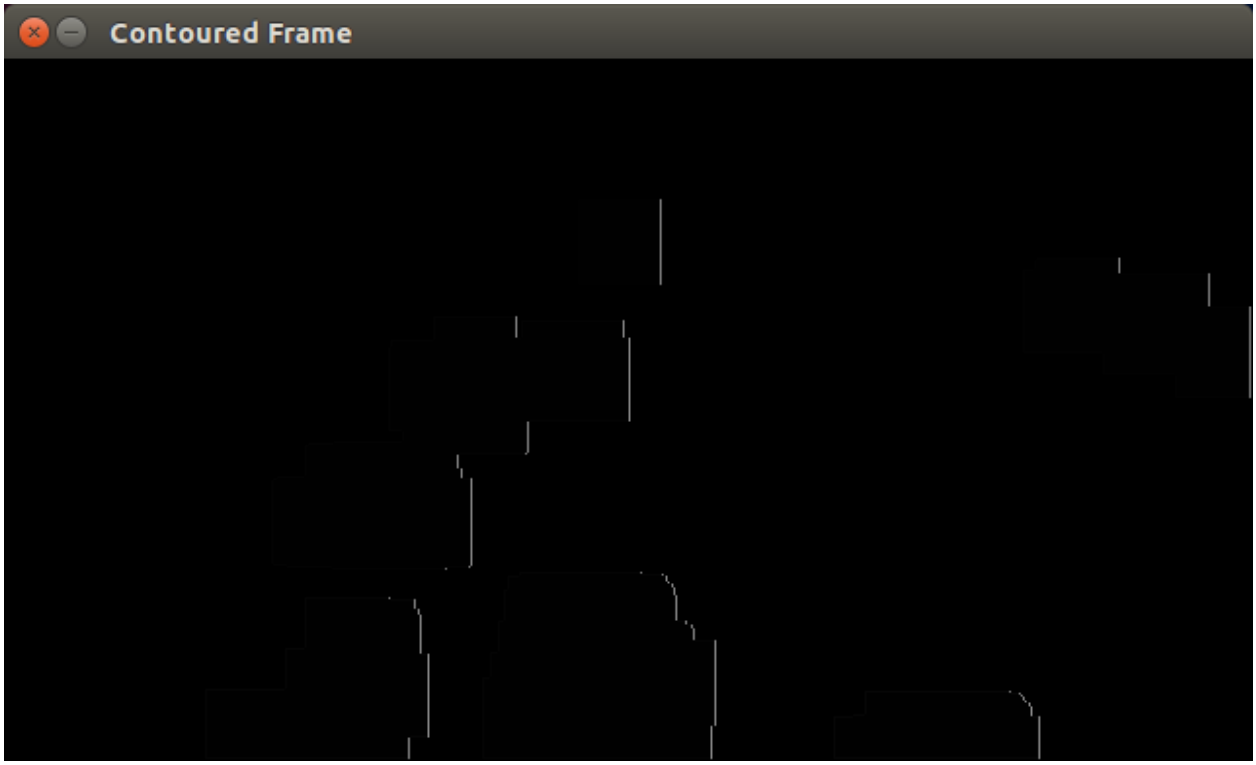

Figure 35 Contoured Frame 3

After that we use the OpenCV function to draw rectangles along contours. Below are the screen shots from the final tracked motion:

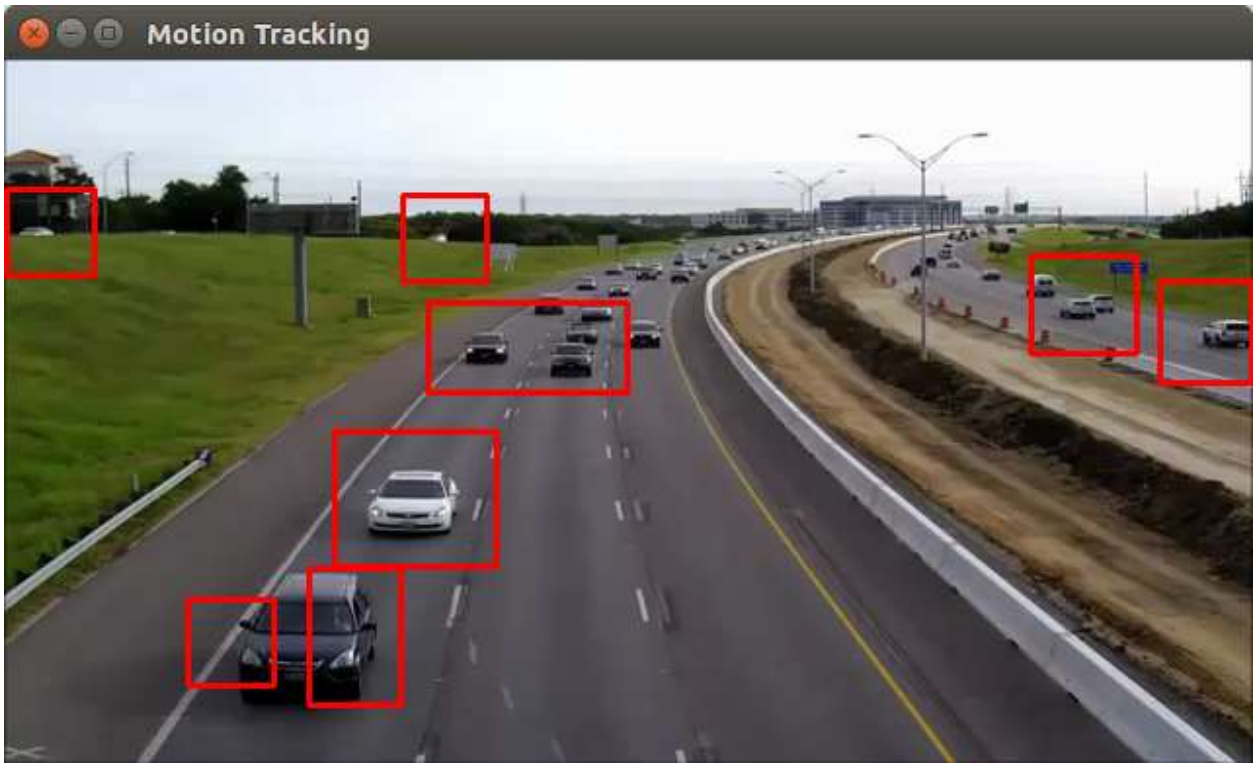

Figure 36 Tracking Frame 1

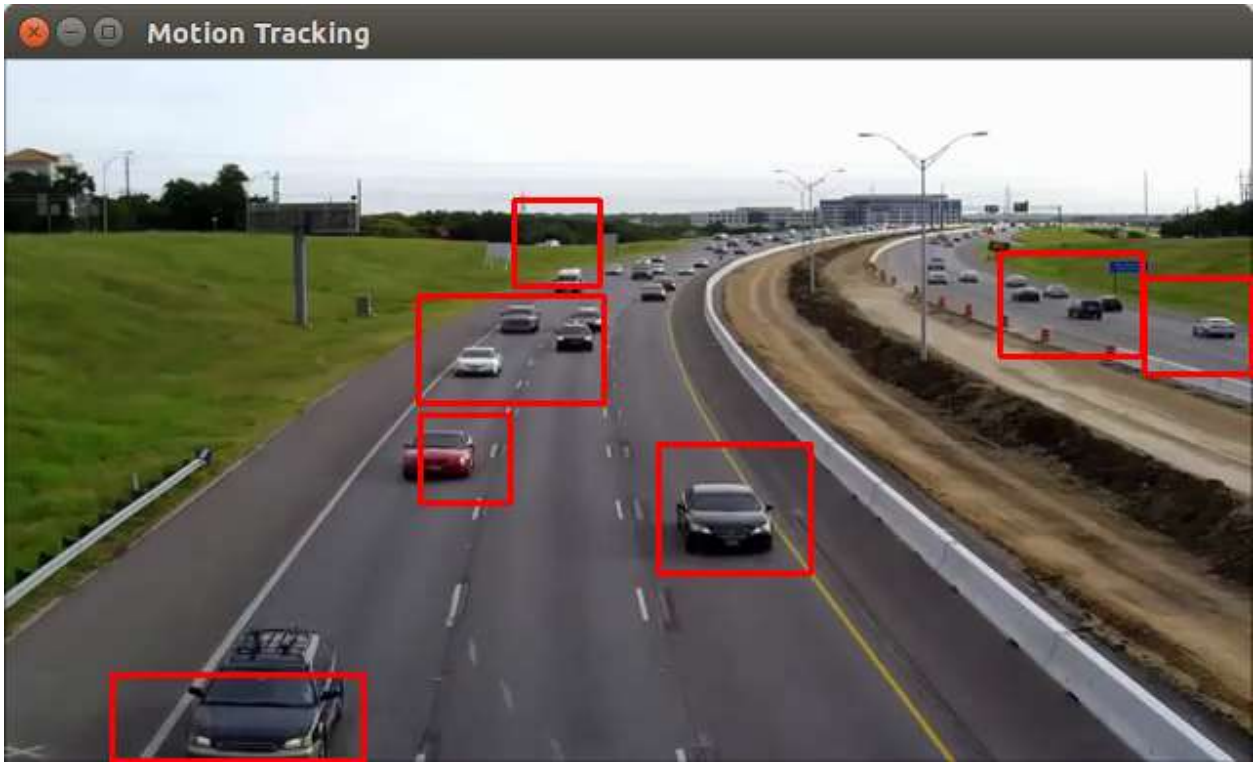

Figure 37 Tracking Frame 2

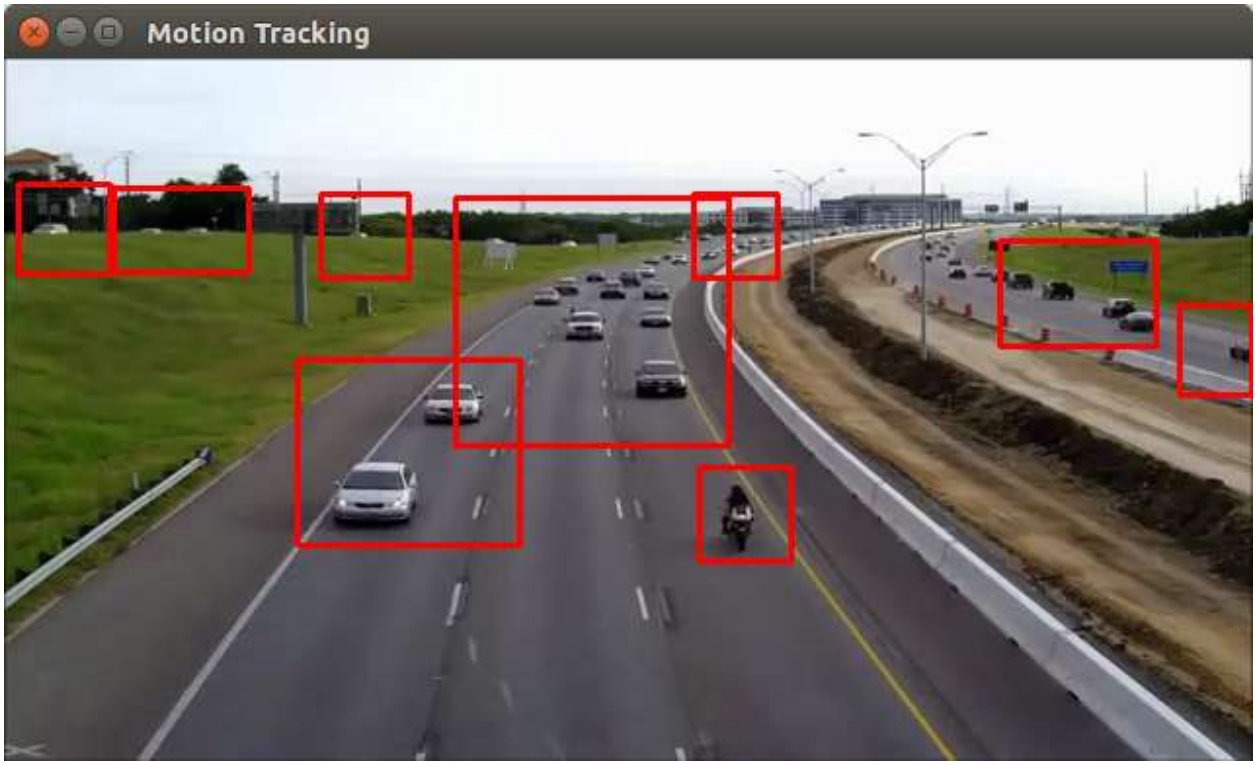

Figure 38 Tracking Frame 3

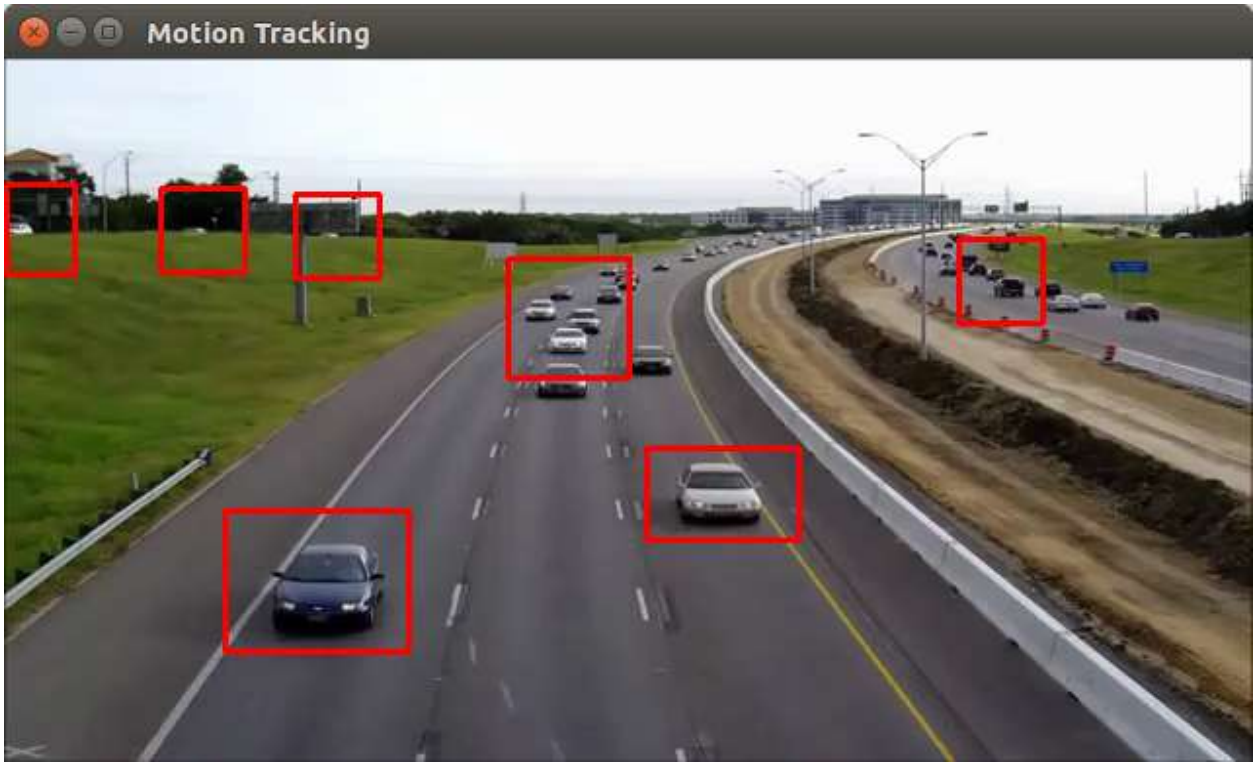

Figure 39 Tracking Frame 4

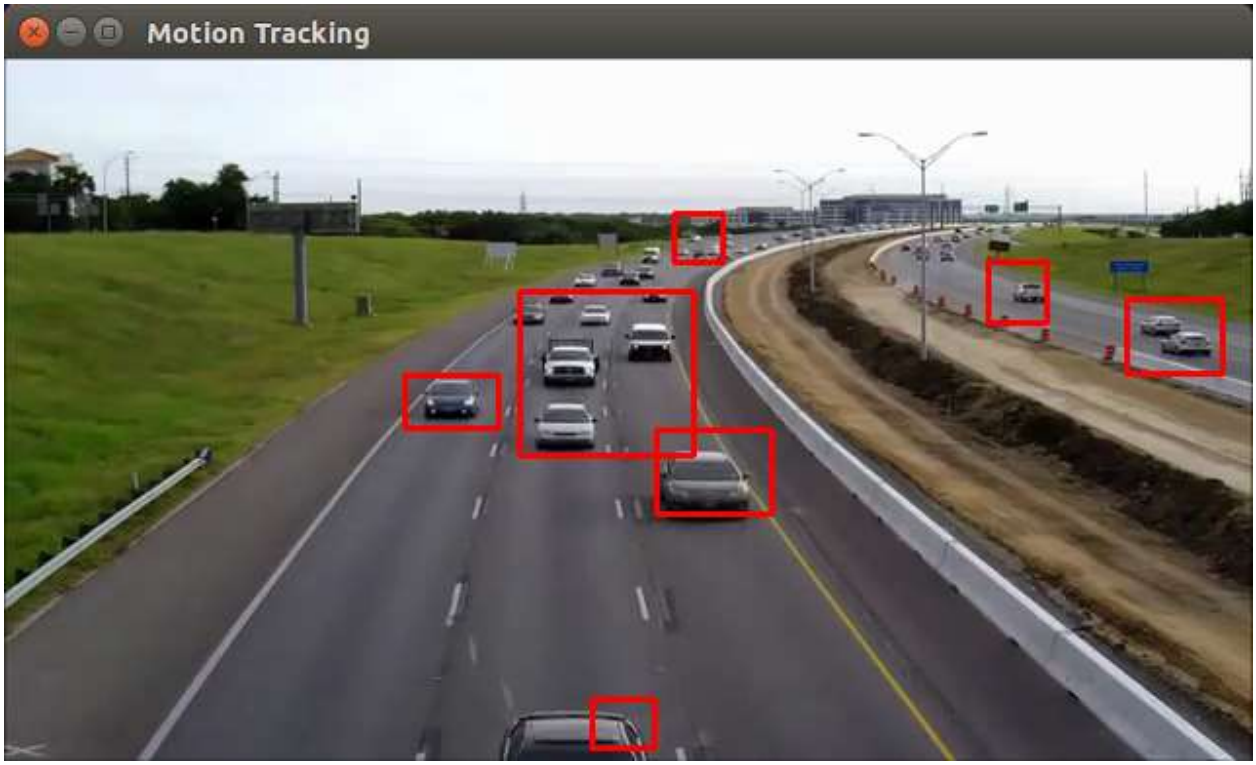

Figure 40 Tracking Frame 5

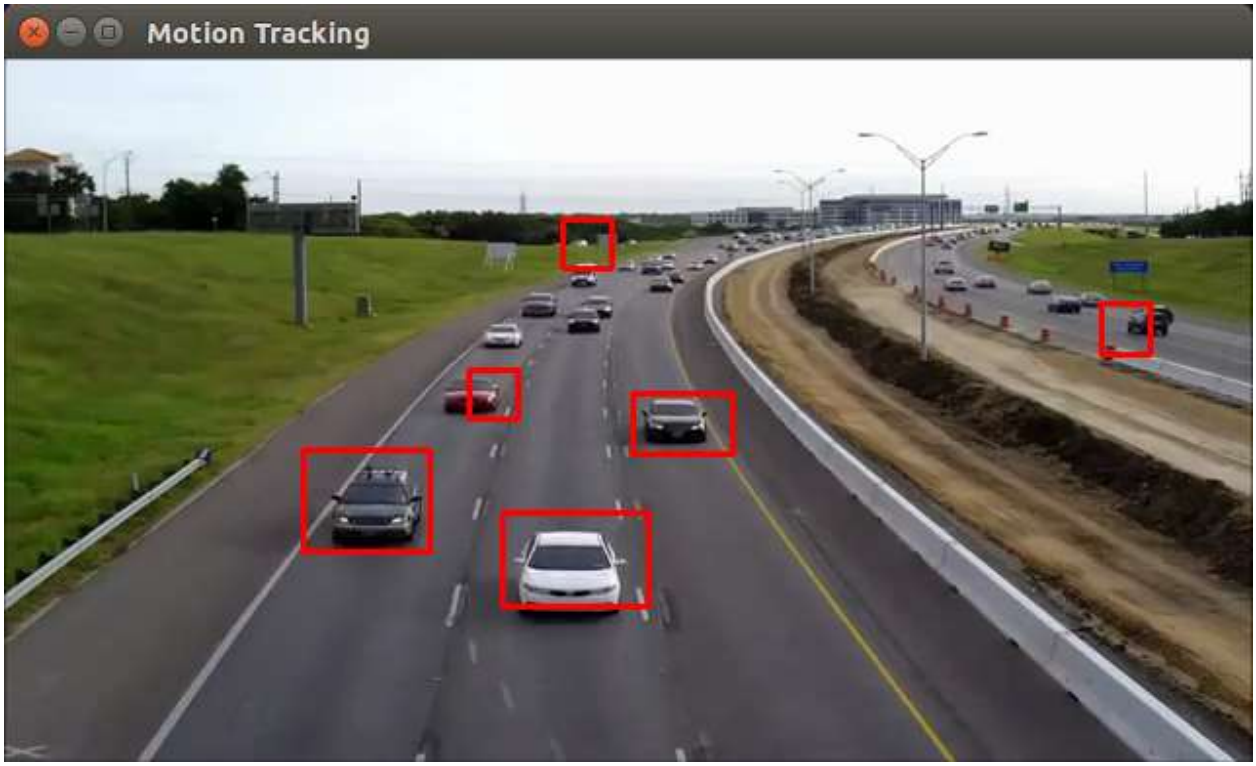

Figure 41 Tracking Frame 6

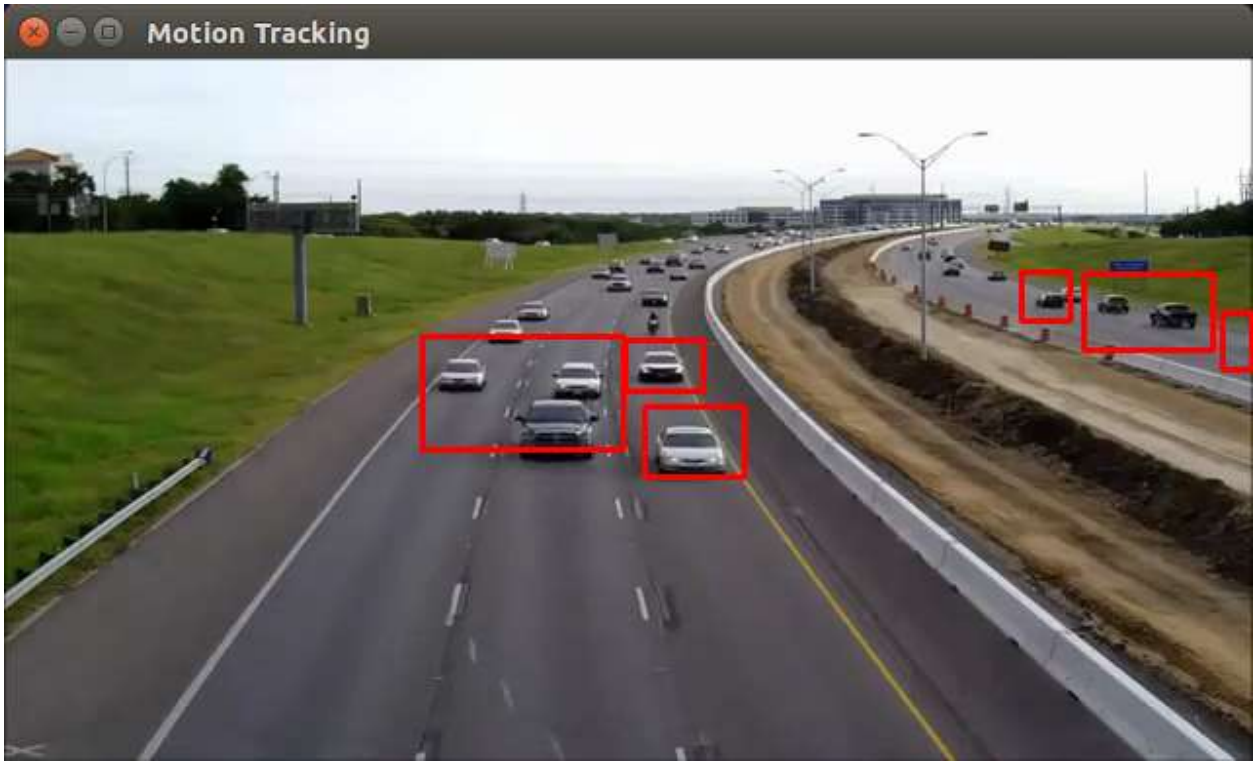

Figure 42 Tracking Frame 7

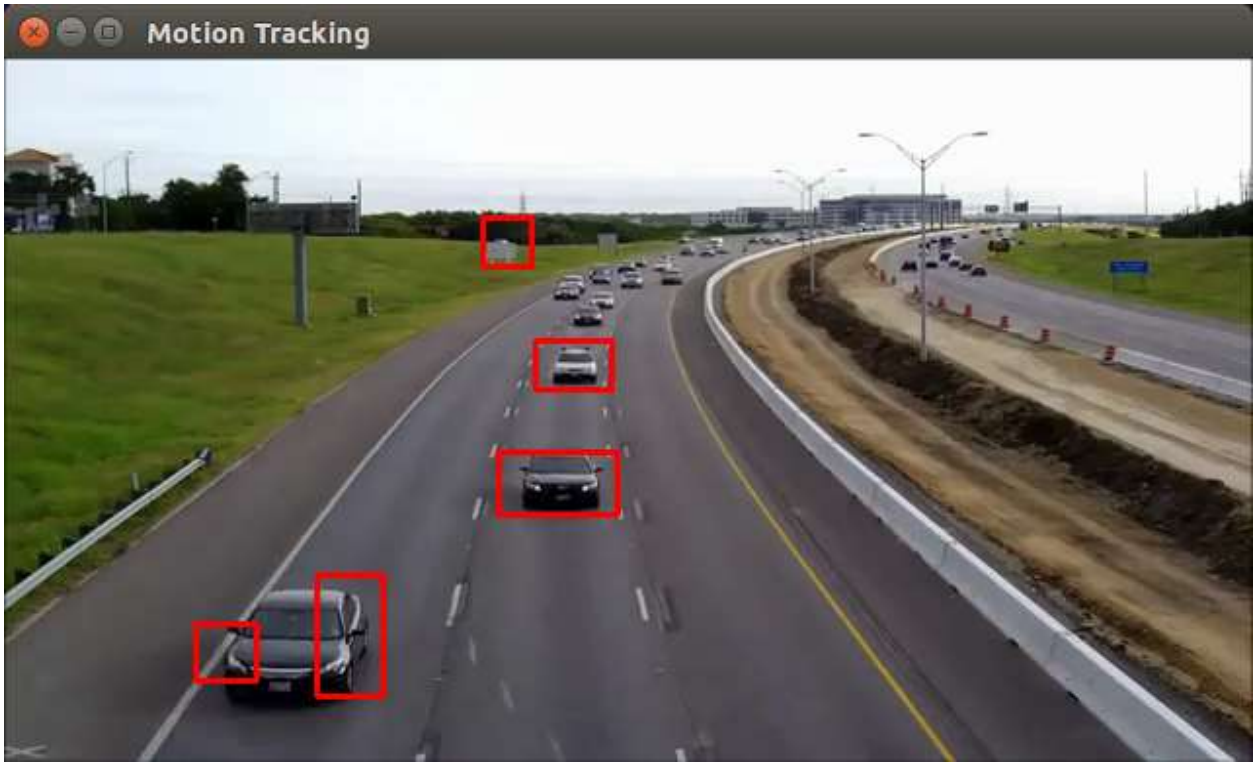

Figure 43 Tracking Frame 8

4.3 CUDA IMPLEMENTATION

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA. It is implemented using the graphics processing units (GPUs) NVIDIA produces. This section describes some basic CUDA APIs required for parallel processing through example codes. After that, we move to our parallel motion tracking algorithm implementation.

4.3.1 BASIC CUDA OPERATIONS

This section uses two simple tasks to illustrate the working of CUDA APIs. First, we parallelize a simple vector addition operation using CUDA and then discuss a more complicated task such as matrix multiplication. After speeding up the computations on random data, we'll move to inTEGRAting the OpenCV library with CUDA C to speed up the real vision tasks onto the GPU.

http://elinux.org/JETSON/Installing_CUDA is a complete guide on installing the CUDA Toolkit for JETSON TK1 board and also includes building and running some CUDA samples.

4.3.1.1 CUDA VECTOR ADDITION

Below is the code for simple vector addition using a GPU. Let us get ourselves acquainted with GPU programming model.

```
/*
Program Name: CudaVectorAdd
This program adds two vector arrays on GPU.
*/

#include <stdio.h>
#define N 512

// Device Vector Add Function.

__global__ void add(int *a, int *b, int *c)
{
   // Using threads only.

      int tid = threadIdx.x;

    c[tid] = a[tid] + b[tid];
}

int main()
{
      // Host side pointers.

      int *a,*b,*c;
```

```
// Device side pointers.

int *dev_a, *dev_b, *dev_c;

//Host side memory allocation.

a=(int *)malloc(N*sizeof(int));

b=(int *)malloc(N*sizeof(int));

c=(int *)malloc(N*sizeof(int));

//Device side memory allocation.

cudaMalloc( (void**)&dev_a, N * sizeof(int) );

cudaMalloc( (void**)&dev_b, N * sizeof(int) );

cudaMalloc( (void**)&dev_c, N * sizeof(int) );

// Initializing Vectors

for (int i=0; i<N; i++)
{
      a[i] = i; b[i] = i;
}

//Copying data to the GPU.

cudaMemcpy ( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
```

```
	cudaMemcpy ( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );

	// GPU kernel launch with one block and N=512 threads.

	add<<<1,N>>>(dev_a, dev_b, dev_c);

	// Copying results back to the Host.

	cudaMemcpy(c, dev_c, N * sizeof(int),cudaMemcpyDeviceToHost );
	//Printing results.

	for (int i=0; i<N; i++)
	{
		printf("%d + %d = %d\n", a[i],b[i],c[i]);
	}

	// Freeing memory to keep the atmosphere clean.

	free(a); free(b); free(c);

	cudaFree (dev_a); cudaFree (dev_b); cudaFree (dev_c);

    return 0;
}
```

Along with describing the code, let us get us familiar with the general CUDA flow. Please note that in the above code, we are referring to CPU when we say Host and GPU when we say Device. The major difference between the CPU and GPU architecture is that CPU has a complex

control logic and fewer computational units, while a GPU has simpler control logic and larger number of simpler computational units.

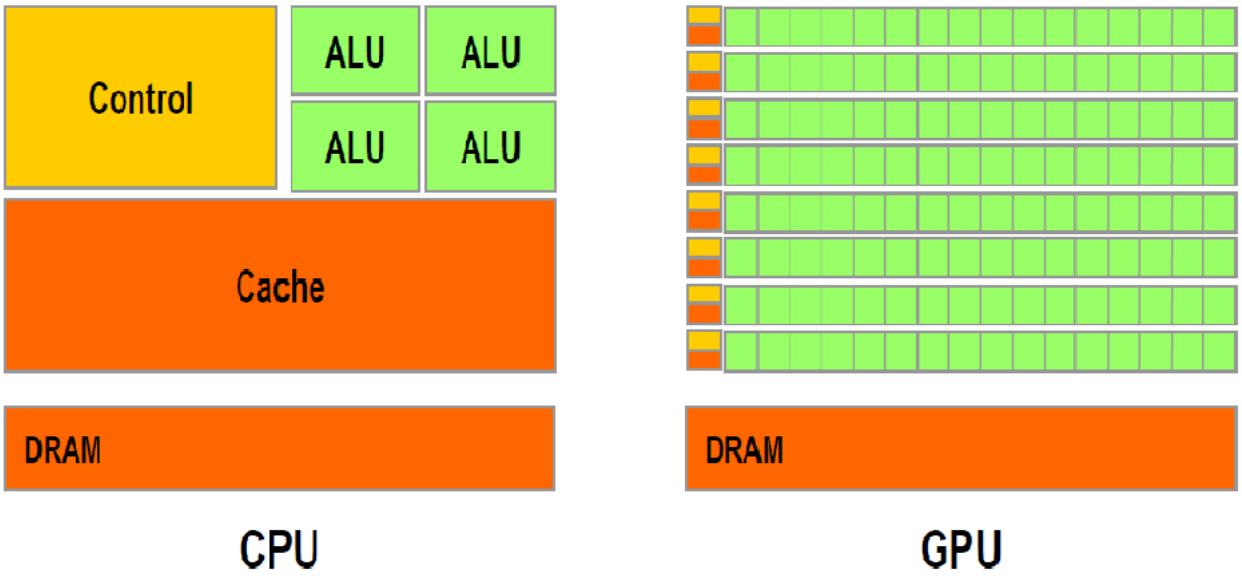

Figure 44 CPU Vs GPU

- Both CPU and GPU have their own memory, so we need to keep track of the data in both different locations. Therefore, we have host side pointers for CPU memory and device side pointers for GPU memory.
- `a=(int *)malloc(N*sizeof(int));` is used for allocating memory on host.
- `cudaMalloc((void**)&dev_a,N*sizeof(int));` is used for allocating memory on to device.
- `cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);`after initializing the vectors, we use cudaMemcpy to transfer the data from CPU memory to GPU memory.
- `add<<<1,N>>>(dev_a, dev_b, dev_c);` after transferring data to GPU, we launch kernel onto GPU. Host is responsible for memory allocation, memory transfer and kernel launch operations onto GPU.

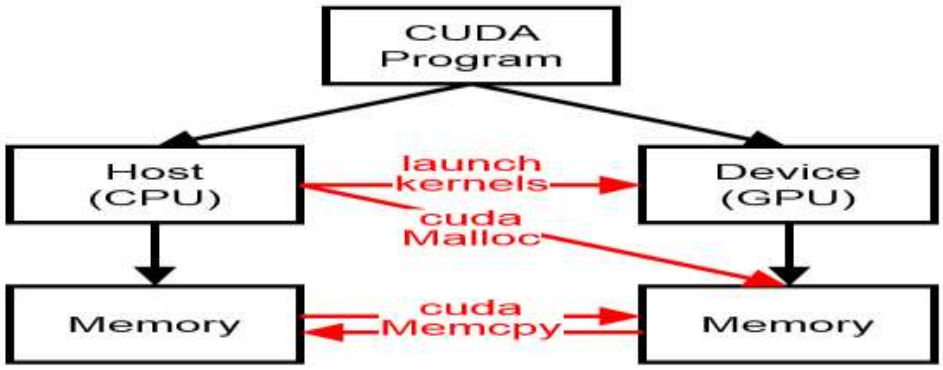

Figure 45 CUDA Program Flow

- The GPU then executes the vector add kernel in a parallel fashion. In this case, we are using only a single block comprising of multiple threads.

```
__global__ void add(int *a, int *b, int *c)
{
   // Using threads only.

      int tid = threadIdx.x;

    c[tid] = a[tid] + b[tid];
}
```

- `__global__` indicates that this function runs on GPU and is callable from host.
- `int tid = threadIdx.x;` threadIdx.x is a built in variable which keeps track of each thread's id.
- `c[tid]=a[tid]+b[tid];` each thread then operates in it's own set of data. Below is given the conceptual view of CUDA's threads and blocks in a grid.

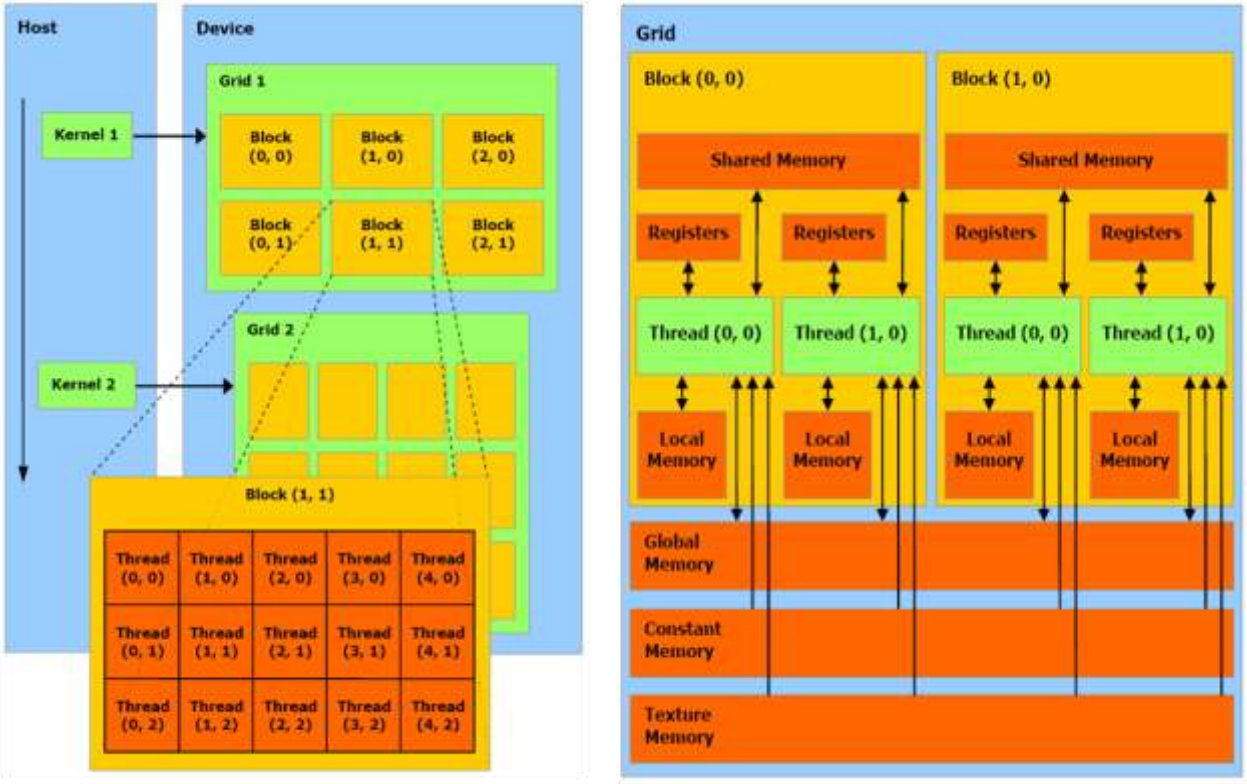

Figure 46 CUDA Blocks and Threads

Blocks and threads can be launched in each x, y and z dimensions depending on the task at hand. This figure illustrates multiple blocks in a two dimensional grid. Each block then have a two dimensional thread launch configuration. In the vector addition problem, we need only launch threads in a single dimension.

- `add<<<1,N>>>(dev_a,dev_b,dev_c);` launches kernel in such a way that we have only one block in a grid which contains N threads in x dimension.
- After the GPU computes the vector addition, the result is then copied back to the CPU memory and all the CPU and GPU memory is cleaned eventually using free and cudaFree APIs.

We could use `add<<<N,1>>>(dev_a,dev_b,dev_c);` to launch N blocks with one thread each block. The kernel function would then be modified like:

```
__global__ void add(int *a, int *b, int *c)
{
```

```
   // Using blocks only.

      int bid = blockIdx.x;

    c[bid] = a[bid] + b[bid];
}
```

And it would achieve the same vector addition without much worry. But what if we have large sized arrays? The thread count in a block would exceed its maximum limit i.e 512 or 1024 (in newer GPUs). In that case, we'd need to launch multiple threads in multiple blocks and our kernel shall be modified accordingly:

```
#define THREADS_PER_BLOCK 1024

__global__ void VectorAdd(int *a, int *b, int *c)
{
      int id= threadIdx.x + blockIdx.x*blockDim.x;

      c[id]=a[id]*b[id];
}
```

Kernel launch configuration shall also be changed:

```
VectorAdd<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a,d_b,d_c);
```

We are now launching 1024 threads per block and multiple blocks in x dimension. Each thread calculates its identity and works on its respective data.

4.3.1.2 CUDA MATRIX MULTIPLICATION

Unlike vector addition, matrix multiplication is a two dimensional problem. In the vector addition example we saw the use of one dimensional grid launch. Now we illustrate two

dimensional use of threads since image processing tasks are two dimensional. We will start with writing a sequential code for the matrix multiplication and then would transform it to be mapped onto GPU. Matrix multiplication has a simple algebraic math, and we'll dive straight into writing C code:

```
for(i=0;i<Width;i++)
      for(j=0;j<Width;j++)
      {
            Sum=0;

            for(k=0;k<Width;k++)
            {
                  a=H_M[i*Width+k];

                  b=H_N[k*Width+j];

                  Sum+=a*b;
            }

            H_P[i*Width+j]=Sum;
      }
```

We have two matrices on host H_M and H_N with m rows and n columns, the outer loops iterate to deal with each pixel using i and j loop variables while the inner loop with k variable calculates the dot product and eventually we write the dot product to the resultant H_P host matrix. The H_M and H_N matrices were initialized with random variables.

The GPU kernel works with the same idea. The difference is that we assign each matrix element to a unique thread which is now responsible for calculating the dot product and writing the result to the resultant matrix. Another notable difference is that now we need to launch a two dimensional grid, blocks and threads both in x and y dimensions:

```
__global__ void Matrix_Multiplication(int *D_M, int *D_N, int *D_P)
{
      int tx=blockIdx.x*Tile_Width+threadIdx.x;

      int ty=blockIdx.y*Tile_Width+threadIdx.y;

      int Sum=0;

      for(int k=0;k<Width;k++)
      {

            Sum+=D_M[ty*Width+k]*D_N[k*Width+tx];
      }
      D_P[ty*Width+tx]=Sum;

}
```

The GPU kernel is fairly simple, tx and ty are the thread ids in x and y dimensions. Each thread runs this kernel and writes the calculated element to the resultant matrix. D_M, D_N and D_P indicate the device side variables.

The other obvious modification is in the kernel launch:

```
dim3 dimBlock(Tile_Width,Tile_Width);

// Two Dimesional blocks with two dimensional threads.

dim3 dimGrid(Width/Tile_Width,Width/Tile_Width);

// 16*16=256, max number of threads per block is 512.
```

```
Matrix_Multiplication<<<dimGrid,dimBlock>>>(D_M,D_N,D_P);

// Kernel Launch configuration.
```

Dim3 is built in struct for holding the dimensions of threads and blocks, in our case we have launched 16 X 16 threads in a single block and a grid of blocks according to the matrix dimensions. dimBlock and dimGrid structs define the size of threads in a block and blocks in a grid. On a 1024 X 1024 matrix size, a Quad Core ARM CPU does the entire multiplication task in around 63 seconds, while the Kepler GPU on JETSON TK1 uses only 50ms. This is a tremendous speed up and a motivation for speeding up vision tasks onto GPU. The complete code is added in the Appendix section of the report.

4.3.2 OPENCV+CUDA IMPLEMENTATION FOR MOTION TRACKING

Unlike CPU based tracking, we have 4 different modules for GPU accelerated tracking:

- Main function which is the front end for video acquisition and works almost the same way as in CPU tracking.
- GPUResourceManager module, which is responsible for allocating memory onto GPU and transferring data to and back from GPU. It is responsible for kernel launch as well.
- GPUFrameProcessor, which runs purely onto GPU, is responsible for speeding the actual frame processing onto GPU.
- DrawContour module is the same as in the CPU tracking.

4.3.2.1 MAIN FUNCTION

The main function is almost the same as described in the previous section:

```
int main(int argc, char** argv)
{
	Mat InputA;
	Mat InputB;
	cv::VideoCapture capA;
```

```
	capA.open(0);



	while(1)
	{
		capA>>InputA;
		capA>>InputB;

		cv::Mat FrameFinalA(InputA.rows,InputA.cols,CV_8U);

		GPUResourceManager(InputA,InputB,FrameFinalA);

		DrawContour(FrameFinalA,InputA);

		cv::imshow("GPU Tracking CAM A",InputA);

		if(cv::waitKey(33)>=0) break;
	}
	return 0;
}
```

The main function reads two consecutive frames and pass these frames to GPUResourceManger. All of this code has been already described in CPU tracking section.

4.3.2.2 GPU RESOURCE MANAGER

```
void GPUResourceManager(const cv::Mat& FrameA, const cv::Mat& FrameB,
cv::Mat& FrameFinal)
{
```

```
	clock_t Time_Start, Time_End, Time_Difference;
	double Time;

	const int colorBytes = FrameA.step * FrameA.rows;
	const int grayBytes = FrameFinal.step * FrameFinal.rows;

	unsigned char *D_FrameA, *D_FrameB, *D_Frame;

	unsigned char *D_Gray, *D_Bin, *D_Ero, *D_Dil, *D_ExA, *D_ExB,
*D_ExC,*D_ExD,*D_ExE,*D_ExF,*D_FrameFinal;

	cudaMalloc<unsigned char>(&D_FrameA,colorBytes);
	cudaMalloc<unsigned char>(&D_FrameB,colorBytes);
	cudaMalloc<unsigned char>(&D_Frame,colorBytes);

	cudaMalloc<unsigned char>(&D_Gray,grayBytes);
	cudaMalloc<unsigned char>(&D_Bin,grayBytes);
	cudaMalloc<unsigned char>(&D_Ero,grayBytes);
	cudaMalloc<unsigned char>(&D_Dil,grayBytes);
	cudaMalloc<unsigned char>(&D_ExA,grayBytes);
	cudaMalloc<unsigned char>(&D_ExB,grayBytes);
	cudaMalloc<unsigned char>(&D_ExC,grayBytes);
	cudaMalloc<unsigned char>(&D_ExD,grayBytes);
	cudaMalloc<unsigned char>(&D_ExE,grayBytes);
	cudaMalloc<unsigned char>(&D_ExF,grayBytes);

	cudaMalloc<unsigned char>(&D_FrameFinal,grayBytes);

	Time_Start=clock();
```

```
	cudaMemcpy(D_FrameA,
FrameA.ptr(),colorBytes,cudaMemcpyHostToDevice);
	cudaMemcpy(D_FrameB,
FrameB.ptr(),colorBytes,cudaMemcpyHostToDevice);

	const dim3 block(32,32);

	const dim3 grid((FrameA.cols + block.x - 1)/block.x, (FrameA.rows
+ block.y - 1)/block.y);


	GPUFrameProcessor<<<grid,block>>>(D_FrameA,D_FrameB,D_Frame,D_Gra
y,D_Bin,D_Ero,D_Dil,D_ExA,D_ExB,D_ExC,D_ExD,D_ExE,D_ExF,

	D_FrameFinal,FrameA.cols,FrameA.rows,FrameA.step,FrameFinal.step)
;

	cudaDeviceSynchronize();

	cudaMemcpy(FrameFinal.ptr(),D_FrameFinal,grayBytes,cudaMemcpyDevi
ceToHost);

	Time_End=clock();
	Time_Difference=Time_End-Time_Start;
	Time=Time_Difference/(double)CLOCKS_PER_SEC ;
	printf ("GPU Frame Processing Rate = %f FPS\n",1/Time);


	cudaFree(D_FrameA);
	cudaFree(D_FrameB);
	cudaFree(D_Frame);
```

```
    cudaFree(D_Gray);
    cudaFree(D_Bin);
    cudaFree(D_Ero);
    cudaFree(D_Dil);
    cudaFree(D_ExA);
    cudaFree(D_ExB);
    cudaFree(D_ExC);
    cudaFree(D_ExD);
    cudaFree(D_ExE);
    cudaFree(D_ExF);
    cudaFree(D_FrameFinal);
}
```

- `const int colorBytes=FrameA.step*FrameA.rows;` since we are transferring color RGB frames onto GPU, therefor we use the OpenCV step function to measure the size of the color frames in bytes which shall be used to allocate memory onto GPU.
- `const int grayBytes=FrameFinal.step*FrameFinal.rows;` after the initial subtraction, the rest of the processing is carried out with gray frames, therefore we calculate the size of gray frames, which shall be used to allocate memory for the frame processing pipeline.
- `unsigned char *D_FrameA, *D_FrameB, *D_Frame;` then we have the device side pointers for the input and output frames. *D_Gray, *D_Bin, *D_Ero, *D_Dil are the device side pointers for holding the results of gray conversion, gray to binary conversion, erosion and dilation operations while the frame is processed onto GPU.
- `*D_ExA, *D_ExB` etc. are the device side pointers for extra frame processing tasks.
- `cudaMalloc<unsigned char>(&D_FrameA,colorBytes);` is for allocating the memory for color frames.
- `cudaMalloc<unsigned char>(&D_Gray,grayBytes);` is for allocating the memory for gray frames.
- `const dim3 block(32,32);` launching 1024 of total threads in a single block.

- `const dim3 grid((FrameA.cols + block.x - 1)/block.x, (FrameA.rows + block.y - 1)/block.y);` calculating the frame size and launching the blocks accordingly.
- After that we transfer the frames to the GPU memory, launch the kernel, copy back the processed frame and free the allocated memory. This function executes for every frame.

Please note that we did not need a separate function when solely working with CUDA C language. OpenCV is a C++ library while CUDA C is just C with CUDA APIs, therefore we need some sort of wrapper function that helps inTEGRAting both types of codes.

4.3.2.3 GPU FRAME PROCESSOR

The GPU frame processor does the following operations:

- RGB Frame Subtraction
- RGB to Gray Conversion
- Gray to Binary Conversion
- Erosion Multiple Times
- Dilation Multiple Times

For the sake of illustration, the original code has been trimmed down to fewer lines:

```
__global__ void GPUFrameProcessor(unsigned char* FrameA,unsigned char*
FrameB,unsigned char* Frame,unsigned char* Gray,unsigned char*
Bin,unsigned char* Ero,unsigned char* Dil,unsigned char* ExA,unsigned
char* ExB,unsigned char* ExC,unsigned char* ExD,unsigned char*
ExE,unsigned char* ExF,unsigned char* FrameF,int width,int height,int
colorWidthStep, int grayWidthStep)
{

    const int xIndex = blockIdx.x * blockDim.x + threadIdx.x;

    const int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
```

```
	if((xIndex>2)  && (yIndex>2) && (xIndex<width-2) &&
(yIndex<height-2))
	{
		const int color_tid = yIndex * colorWidthStep + (3 *
xIndex);
		const int gray_tid  = yIndex * grayWidthStep + xIndex;

		Frame[color_tid]=FrameB[color_tid]-FrameA[color_tid];
		Frame[color_tid+1]=FrameB[color_tid+1]-FrameA[color_tid+1];
		Frame[color_tid+2]=FrameB[color_tid+2]-FrameA[color_tid+2];

		__syncthreads();

		const unsigned char blue    = Frame[color_tid];
		const unsigned char green   = Frame[color_tid + 1];
		const unsigned char red     = Frame[color_tid + 2];

		const float gray = red * 0.3f + green * 0.59f + blue *
0.11f;

		Gray[gray_tid] = static_cast<unsigned char>(gray);

		__syncthreads();

		if(Gray[gray_tid]>220)
			Bin[gray_tid]=255;
		else
			Bin[gray_tid]=0;

		__syncthreads();
```

```
        const int tidA = (yIndex) * grayWidthStep + (xIndex);
        const int tidB = (yIndex-1) * grayWidthStep + (xIndex);
        const int tidD = (yIndex) * grayWidthStep + (xIndex-1);
        const int tidE = (yIndex) * grayWidthStep + (xIndex+1);
        const int tidF = (yIndex-1) * grayWidthStep + (xIndex-1);
        const int tidG = (yIndex-1) * grayWidthStep + (xIndex+1);

    if((Bin[tidA]>100)&&(Bin[tidB]>100)&&(Bin[tidD]>100)&&(Bin[tidE]>
100)&&(Bin[tidG]>100)&&(Bin[tidF]>100))
        Ero[gray_tid]=255;
        else
        Ero[gray_tid]=0;

        __syncthreads();

        int i,j;
        float Sum=0;

        for(i=-3;i<=3;i++)
        for(j=-3;j<=3;j++)
        {
            if(ExF[(yIndex+i)*grayWidthStep+(xIndex+j)]>100)
            Sum++;
        }


        if(Sum>0)
            FrameF[tidA]=255;
        else
            FrameF[tidA]=0;
```

```
    }
}
```

- `Frame[color_tid]=FrameB[color_tid]-FrameA[color_tid];` subtracts the blue color component of the consecutive frames.
- `Frame[color_tid+1]=FrameB[color_tid+1]-FrameA[color_tid+1];` subtracts the green color component of the consecutive frames.
- `Frame[color_tid+2]=FrameB[color_tid+2]-FrameA[color_tid+2];`subtracts the red color component of the consecutive frames.
- `Const float gray=red*0.3f+green*0.59f+blue*0.11f;` converts the subtracted color frame to gray frame.
- `if(Gray[gray_tid]>220) Bin[gray_tid]=255;elseBin[gray_tid]=0;` does the conversion from gray to binary.
- `const int tidA =(yIndex)*grayWidthStep+(xIndex);` has been used to access the neighbouring pixels of the image to do erosion and dilation which is a form of 2 dimensional convolution. tidB, tidC etc have been used for the same purpose.
- After that, we apply a check that if all the neighbouring pixels are one, make the centroid pixel equal to one otherwise zero. This is one way of doing erosion and it helps remove the noise.
- In the end, we apply the dilation filter, it uses a looping mechanism to access the neighbouring pixels, if only a single of these pixels is one, the centroid pixel is made equal to one. If all the pixels are zero, then only the centroid pixel is made equal to zero. It helps expand the size of the objects in the frame.

4.3.2.4 CONTOUR FINDER

This function is the same as was used in CPU tracking OpenCV code with same results as shown previously.

CHAPTER 5

EXPERIMENTS AND RESULTS

This chapter explains the actual hardware system for conducting the experiments, the performance evaluation scenarios and the actual findings against real time motion tracking.

5.1 HARDWARE

Three different hardware systems have been used in all the experiments.

5.1.1 NVIDIA JETSON TK1

JETSON TK1 is a fully featured development platform from NVIDIA. It is based on TEGRA SOC which comprises of a Quad Core ARM Cortex A15 processor along with a 192 CUDA Core KEPLER GPU. To enable development of computer vision and other camera based embedded applications, JETSON TK1 is capable of supporting multiple cameras through a variety of interfaces.

5.1.2 INTEL Xeon with GEFORCE GTX950 DESKTOP

The second system used is a 2.76GHz desktop Intel Xeon W3520 (Quad Core with 8 threads) along with an NVIDIA GEFORCE GTX950 machine. GTX950 is 768 CUDA Core GPU.

5.1.3 ASUS G53JW NOTEBOOK

ASUS G53JW has a 1.7GHz OCTA Core Intel Core i7 processor along with a 192 CUDA Core GEFORCE GTX 430M graphics card.

5.1.4 CAMERAS

Two low cost 640 X 480 resolution cameras with a maximum of 30FPS rate communicating via USB 2.0 interface have been used in these experiments.

5.2 SOFTWARE

The development environment used in both the platforms is Linux UBUNTU 14.04. Moreover, OpenCV was used for all the CPU based tasks such as video acquisition and video display and CUDA Toolkit 6.5 was used for GPU software development.

5.3 PERFORMANCE ANALYSIS

Different test scenarios have been used against all the hardware platforms. The tracking frame rates are first measured using a single camera setup and then have been expanded for two cameras. A performance to cost ratio analysis has been made against CPU only tracking, JETSON based tracking and a Desktop CPU+GPU based tracking.

5.3.1 SINGLE CAMERA BASED TRACKING

For CPU based tracking, the FPS rate is measured for the complete algorithm except the contour finding and rectangle drawing part. For GPU based tracking, there are three different cases:

A. Including all the time for memory allocation and data transfer to GPU.
B. Without memory allocation and data transfer time; meaning the GPU kernel execution time only.
C. Without memory allocation but with data transfer to and from GPU time.

Table 1 summarizes the detailed findings:

Cases	**ARM/ OpenCV FPS**	**Intel Xeon/ OpenCV FPS**	**ASUS/ OpenCV FPS**	**JETSON/ OpenCV& CUDA FPS**	**Desktop/ OpenCV& CUDA FPS**	**ASUS/ OpenCV& CUDA FPS**
Including memory allocation and data transfer time.	14	250	130	130	240	170
Without memory allocation and data transfer time.	14	250	130	4464	750	336
Without memory allocation but with data transfer time.	14	250	130	270	420	250

Table 1 Single Camera Performance

It can clearly be seen that Intel Xeon is way more powerful than the small low powered ARM processor when the things are done sequentially. In case of the hybrid platforms, JETSON TK1 is comparable to a powerful desktop machine in terms of performance to cost ratio.

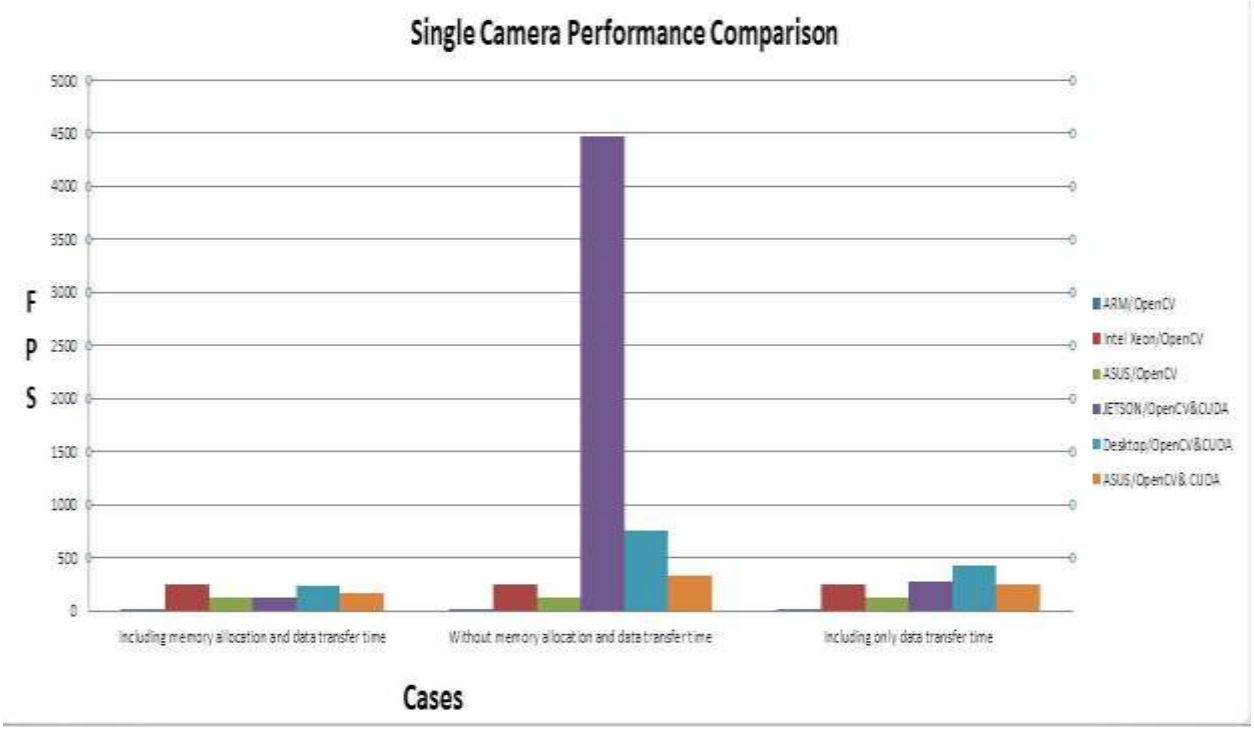

Figure 47 Single Camera Performance 1

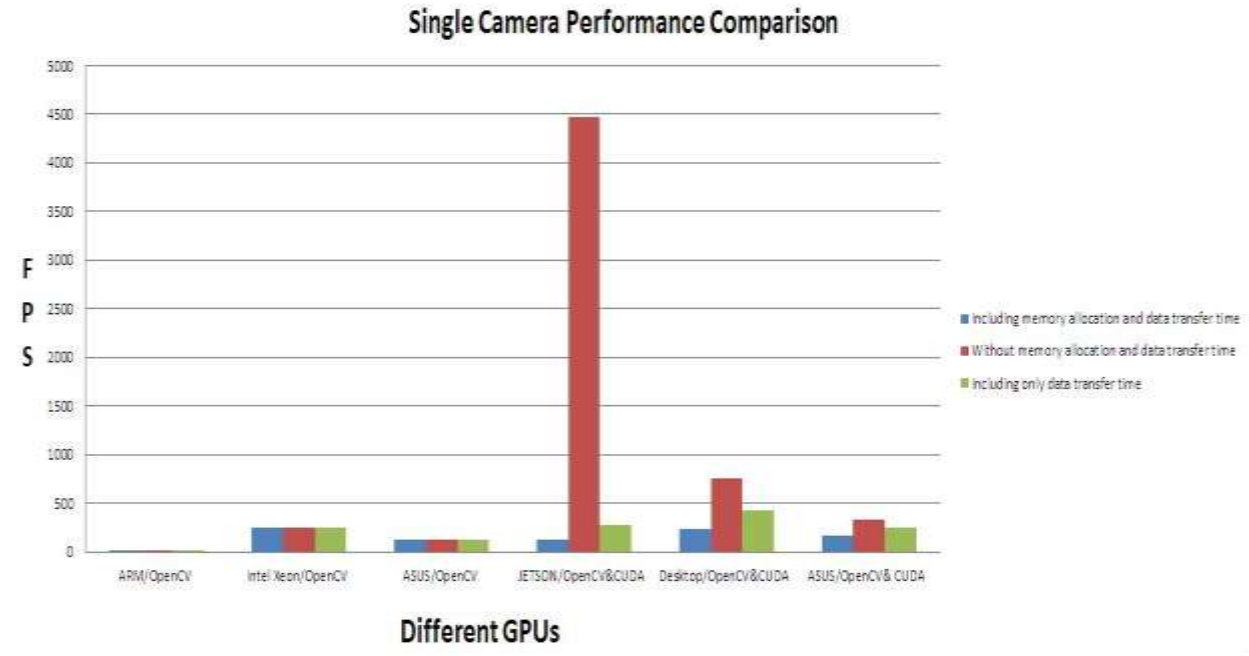

Figure 48 Single Camera Performance 2

5.3.2 MULTIPLE CAMERAS BASED TRACKING

Multiple cameras based tracking performance evaluation has been carried out in the same manner as single camera based. The difference is that two different camera frames are processed sequentially both onto CPU as well as GPU. Table 2 shows the summary:

Cases	**ARM/ OpenCV FPS**	**Intel Xeon/ OpenCV FPS**	**ASUS/ OpenCV FPS**	**JETSON/ OpenCV& CUDA FPS**	**Desktop/ OpenCV& CUDA FPS**	**ASUS/ OpenCV& CUDA FPS**
Including memory allocation and data transfer time.	7	140	80	46	110	46
Without memory allocation and data transfer time.	7	140	80	1902	390	168
Without memory allocation but with data transfer time.	7	140	80	123	200	128

Table 2 Multiple Camera Performance

The findings show that JETSON TK1 is capable of delivering almost 60% of the performance of Intel Xeon + GTX 950 and almost an equal performance to ASUS G53JW machine.

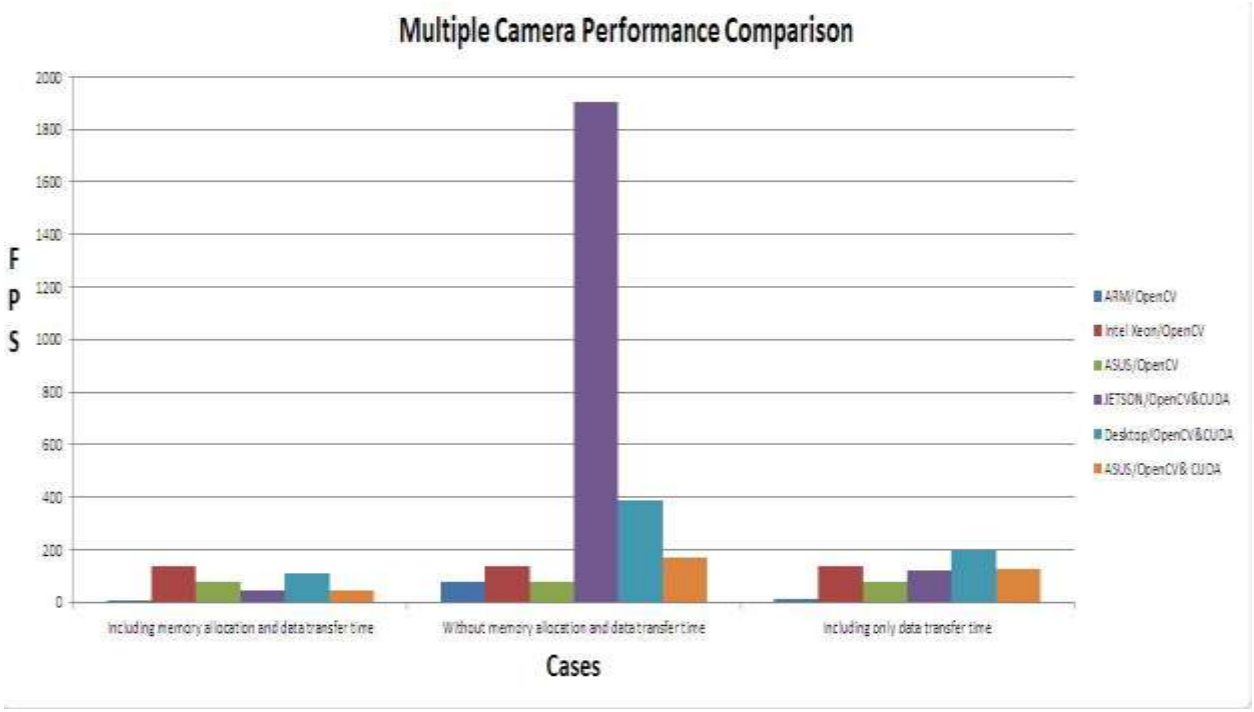

Figure 49 Multiple Camera Performance 1

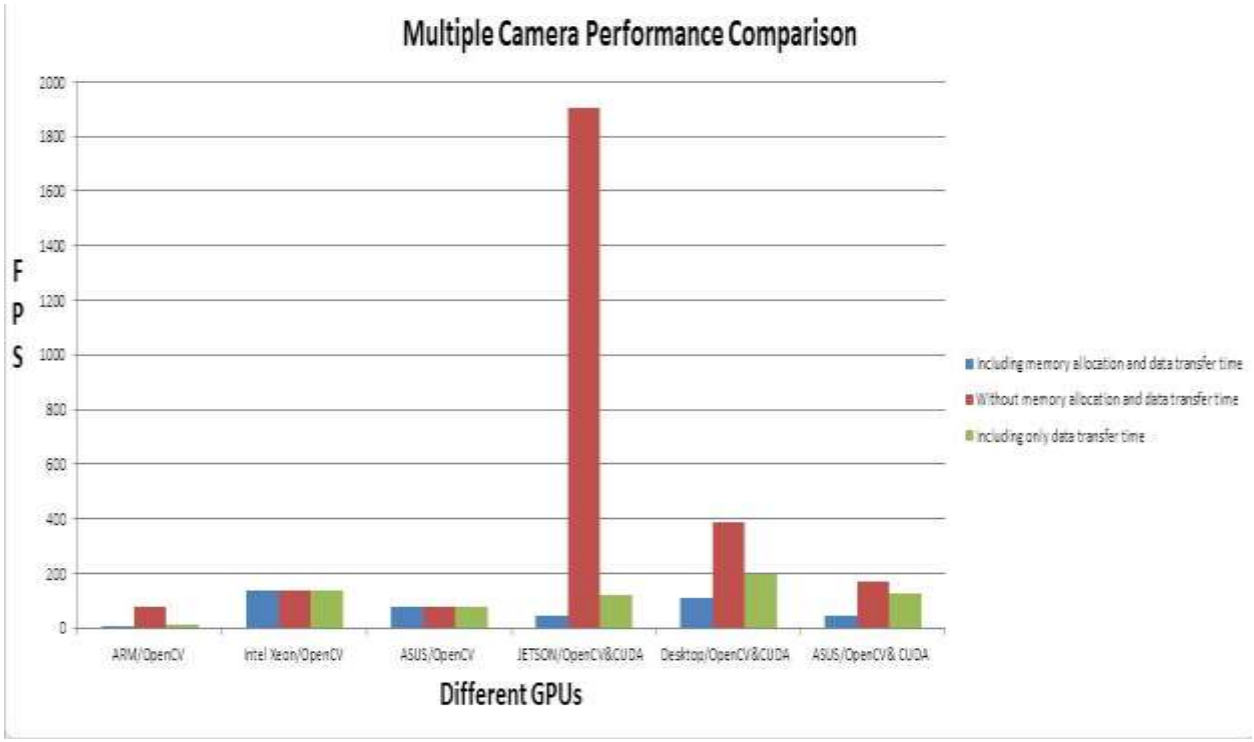

Figure 50 Multiple Camera Performance 2

5.3.3 PERFORMANCE TO COST RATIO

JETSON TK1 is a low cost, low powered embedded platform, bringing the power of GPU to embedded computer vision applications. The GPU accelerated computer vision applications were previously deployed in desktop systems or in clusters containing the graphics processor. The powerful desktop based GPU accelerated systems have a significantly higher cost when compared to JETSON TK1. Moreover, these desktops aren't a fair option for immediate mobility or for deployment as a standalone embedded platforms. A performance to cost ratio comparison has been therefore performed.

5.3.3.1 Single Camera System

Cases	**Platform**	**Total Cost $**	**Performance FPS**	**Ratio FPS/Cost**
Including all the time for memory allocation and data transfer to GPU.	Xeon/ OpenCV	250	250	1
	ASUS/ OpenCV+CUDA	500	170	0.34
	JETSON/ OpenCV+CUDA	193	130	0.67
	Desktop/ OpenCV+CUDA	400	240	0.60
Without memory allocation and data transfer time.	Xeon/ OpenCV	250	250	1
	ASUS/ OpenCV+CUDA	500	335	0.67
	JETSON/ OpenCV+CUDA	193	4464	23.1
	Desktop/ OpenCV+CUDA	400	750	1.87
Without memory allocation but with data transfer to and from GPU time.	Xeon/ OpenCV	250	250	1
	ASUS/ OpenCV+CUDA	500	250	0.5

	JETSON/ OpenCV+CUDA	193	270	1.4
	Desktop/ OpenCV+CUDA	400	420	1.05

Table 3 Single Camera Performance to Cost Ratio

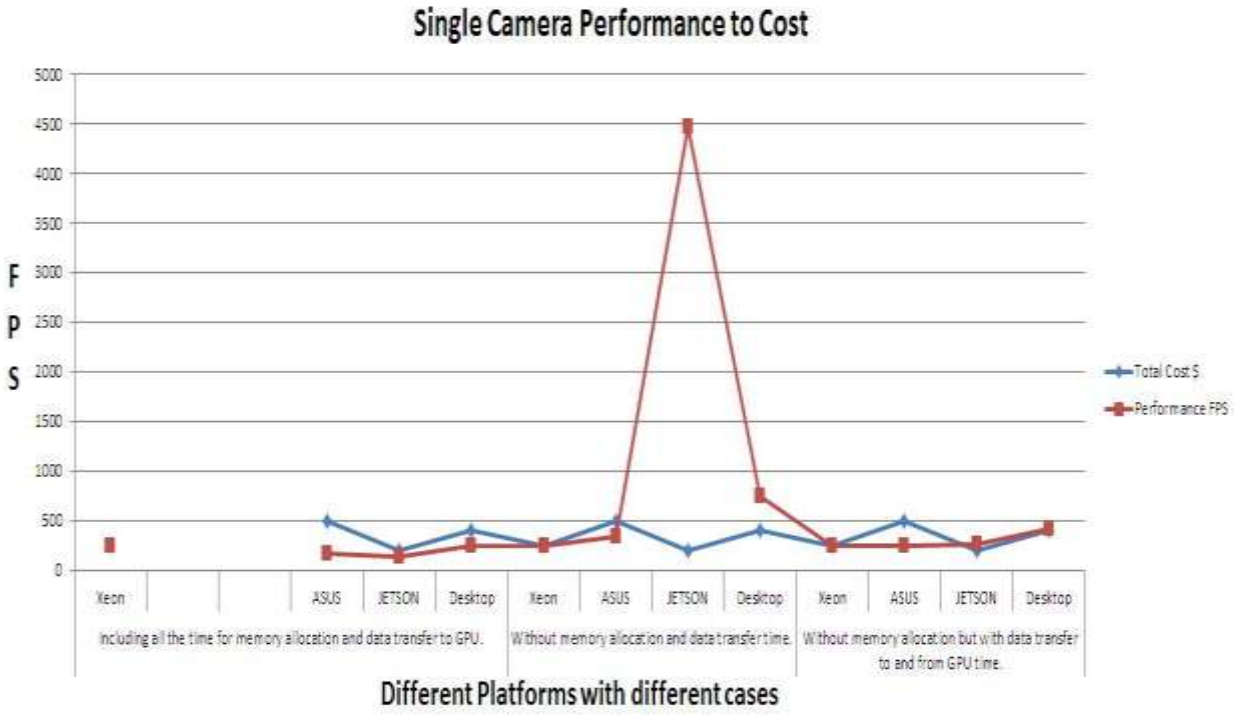

Figure 51 Single Camera Performance Ratio

5.3.3.2 Multiple Camera System

Cases	Platform	Total Cost $	Performance FPS	Ratio FPS/Cost
Including all the time for memory allocation and data transfer to GPU.	Xeon	250	140	0.56
	ASUS	500	46	0.092

	JETSON	193	46	0.24
	Desktop	400	110	0.26
Without memory allocation and data transfer time.	Xeon	250	140	0.56
	ASUS	500	168	0.336
	JETSON	193	1902	9.85
	Desktop	400	390	0.975
Without memory allocation but with data transfer to and from GPU time.	Xeon	250	140	0.56
	ASUS	500	128	0.256
	JETSON	193	123	0.64
	Desktop	400	200	0.50

Table 4 Multiple Camera Performance to Cost Ratio

JETSON TK1 clearly has a significantly higher performance to cost ratio. It doesn't only show a very good speed gain over a very strong CPU, but has also won over two very powerful CPU+GPU machines in terms of performance to cost ratio.

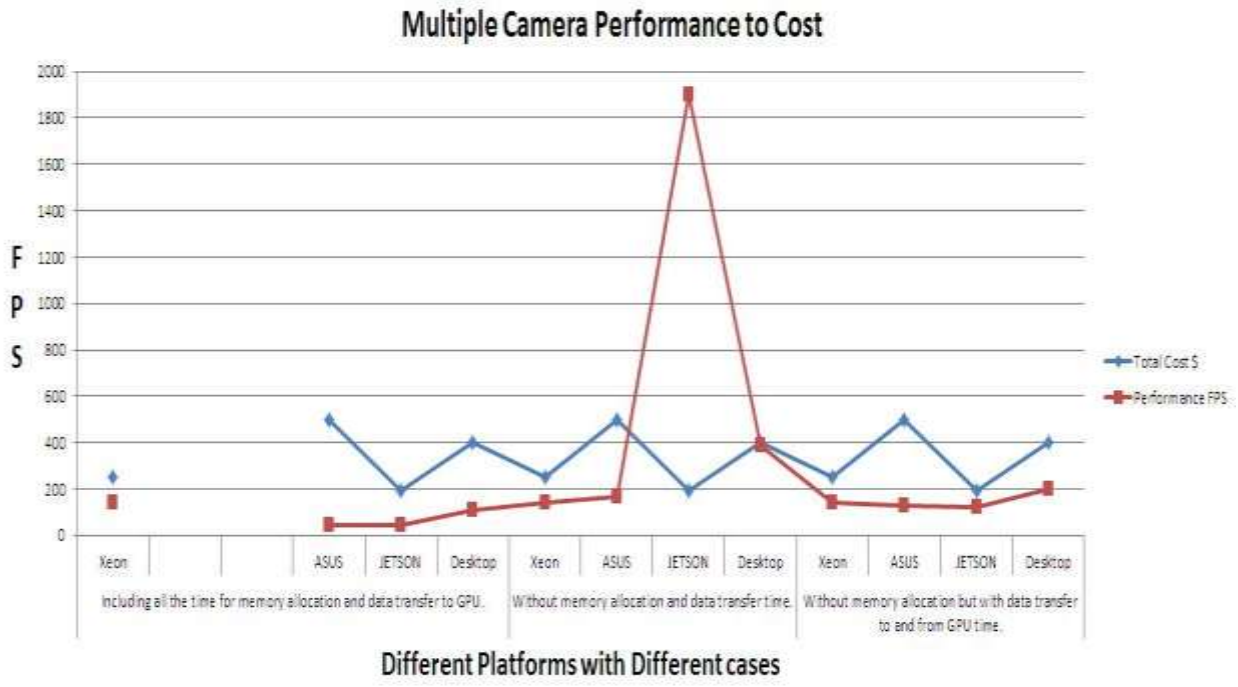

Figure 52 Multiple Camera Performance Ration

5.4 REAL TIME RESULTS

In the figures below CPU is tracking white moving pages, which have been placed below a running fan. Please note this code is using JETSON's ARM CPU only.

Figure 53 CPU Tracking White Pages 1

Figure 54 CPU Tracking White Pages 2

CPU versus GPU tracking quality is shown in these figures. It can be seen that the GPU can track the fast moving blades of that running fan when a CPU fails.

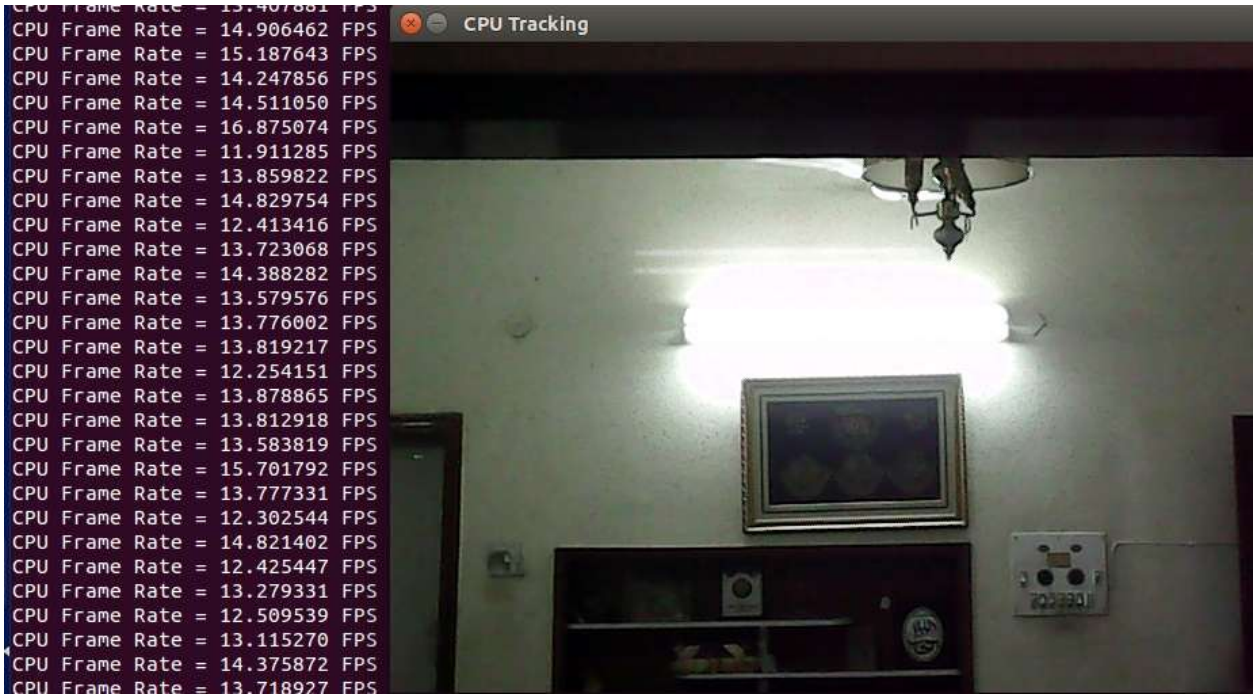

Figure 55 CPU Tracking Fast Moving Fan

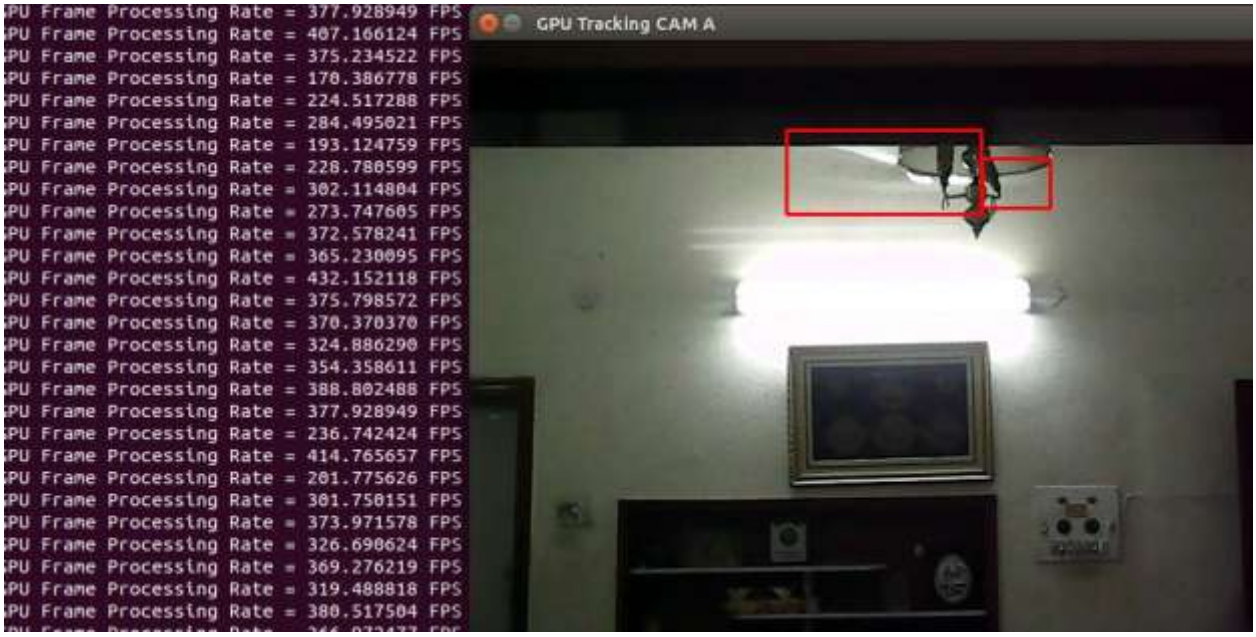

Figure 56 GPU Tracking Fast Moving Fan

Figure 4.5 shows how well the system adapts to changing illumination effects and keeps tracking the fast moving fan.

Figure 57 GPU Adapting to Illumination Changes

Figures 4.6 and 4.7 show how well the system tracks a moving arm and a walking person.

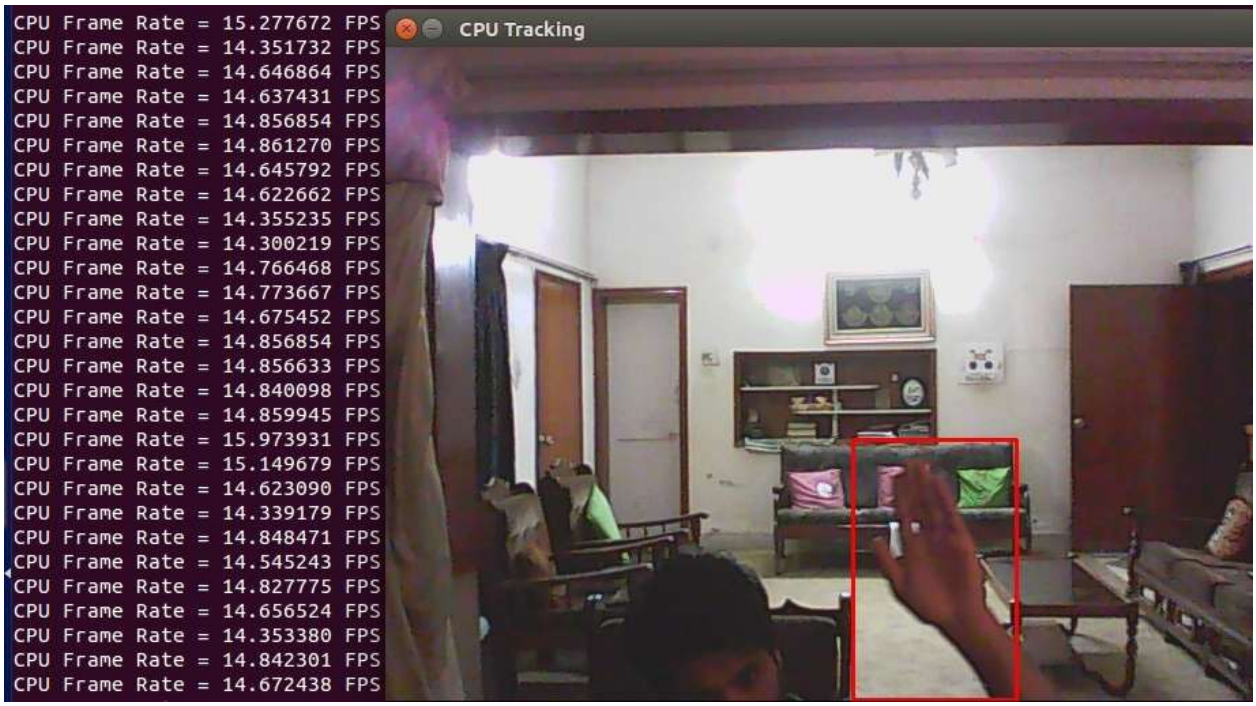

Figure 58 Tracking Human Hand

Figure 59 Tracking Human Body

Figure below shows the tracking results on a video created over a Highway.

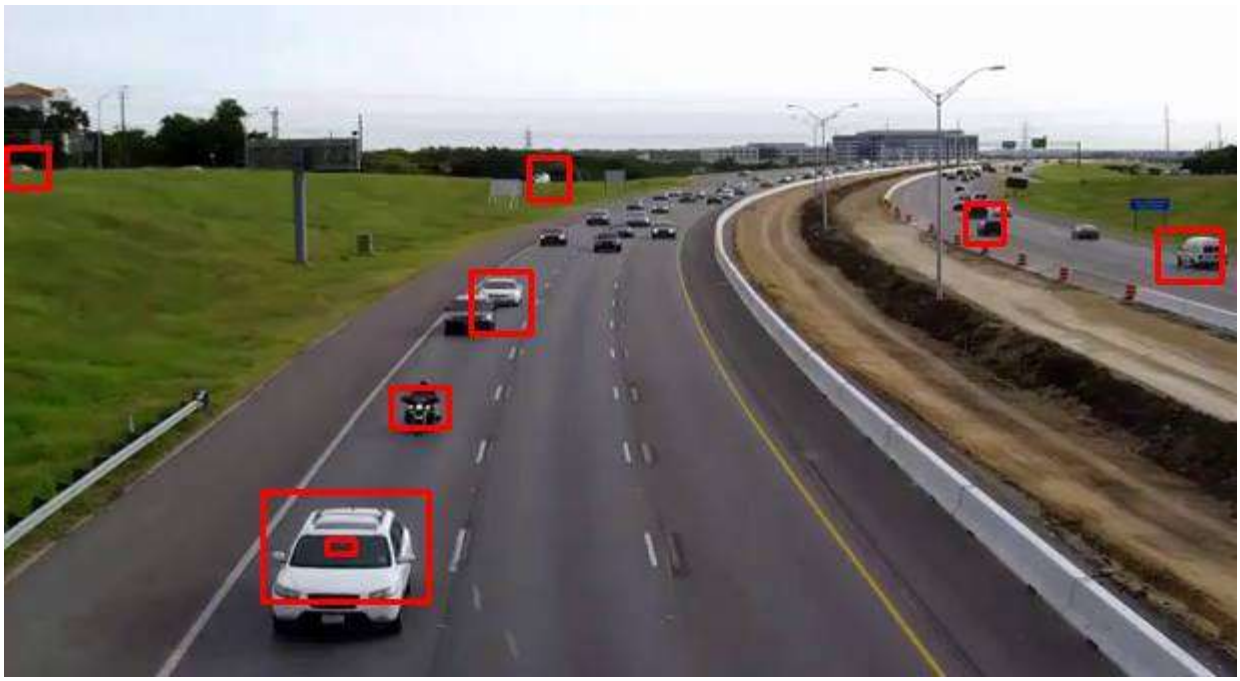

Figure 60 Tracking Moving Vehicles

Figure below shows the tracking results of a fast moving missile in a video.

Figure 61 Tracking Moving Missile

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

The work presents a novel technique as well as platform to achieve fast motion tracking using multiple cameras simultaneously. The algorithm presented not only has a significant performance gain against the CPU based systems but also has better tracking results. The proposed algorithm also showed a self adjusting capability to changing backgrounds. Moreover, a comprehensive performance to cost ratio analysis shows that the JETSON TK1 is the best hybrid CPU+GPU platform against its counter desktop CPU+GPU hybrid platforms when fast motion tracking is to be deployed.

6.2 FUTURE WORK

In this work, tracking has been analyzed for up to two cameras working simultaneously. JETSON TK1 has the capability of working with up to six cameras simultaneously through a variety of interfaces. The camera resolution was 640 X 480 which can be expanded to HD, Ultra HD or even 8K (7680 × 4320) frame resolutions. The algorithm can further be improved and more parallelism can be introduced.

REFERENCES

1. Zhiyi Yang, Yating Zhu and Yong Pu, "Parallel Image Processing Based on CUDA", 2008 International Conference on Computer Science and Software Engineering, Dept. Computer of Northwestern Polytechnical University, Xi'an, Shaanxi, China.
2. Ir. Rudi GIOT and Ing. Abilio RODRIGUES E SOUSA, Msc, "Image processing algorithm optimization with CUDA for Pure Data", 150, Rue Royale, Bruxelles, Belgique, 1000.
3. Lawrence Chan, Jae W. Lee, Alex Rothberg, and Paul Weaver, "Parallelizing H.264 Motion Estimation Algorithm using CUDA", Final report for 6.963: CUDA@MIT in IAP 2009, Massachusetts Institute of Technology.
4. Peter Kuchnio and David Capson, "GPU-Accelerated Foveation for Video Frame Rate Tracking", 2011 Canadian Conference on Computer and Robot Vision.
5. Jing Huang, Sean P. Ponce, Seung In Park, Yong Cao, and Francis Quek, "GPU-Accelerated Computation for Robust Motion Tracking Using the CUDA Framework", Center of Human Computer Interaction, Virginia Polytechnic Institute and State University, Blacksburg, VA 24060, USA.
6. Sidi Ahmed Mahmoudi and Pierre Manneback, 'Multi-GPU based Event Detection and Localization using High Definition Videos', University of Mons, Faculty of Engineering, Computer Science Department Place du Parc, 20, 7000 Mons, Belgium.
7. Min Su, Jianjun Tuan and others, 'Constructing a Mobility and Acceleration Computing Platform with NVIDIA JETSON TK1', High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS).
8. Yash Ukidave, David Kaeli, Umesh Gupta and Kurt Keville 'Performance of the NVIDIA JETSON TK1 in HPC', 2015 IEEE International Conference on Cluster Computing.
9. 'Programming Massively Parallel Processors' by David B. Kirk, Wen-mei W. Hwu, 1st Edition, ISBN:978-0-12-381472-2

10. http://www.NVIDIA.com/object/JETSON-tk1-embedded-dev-kit.html
11. http://elinux.org/JETSON_TK1
12. http://opencv.org/