# EE-5351 Realtime Motion Detection with cuda

Anubhav Panda
Panda047@umn.edu
Sanjit Dash
Dash0030@umn.edu

***Abstract*—GPUs provide a platform for high scale parallel implementation of algorithms on large data sets in order to get fast and efficient results. In this project we utilize the ability of the GPU to accelerate one of the computer vision algorithm i.e. motion detection. We explore a GPU-based implementation of the algorithm that is fast enough to operate on the frames of a live stream video. The algorithm consists of steps like noise reduction, edge detection, thresholding, difference, erosion and detection. Some of the steps are parallelized to gain an effective speedup of about 5x-10x over the serial CPU based implementation.**

***Keywords*—GPU, kernel, OpenCV, pixel, NMS, thresholding**

## I. Introduction

In the past few years the field of computer vision has gained a lot of impetus. Any computations involving visual content – that means videos, images, icons, and anything else that is related to pixels comes under the umbrella of computer vision. Though this technology seems very interesting it is equally complex due to the combination of a number of algorithms. To facilitate smoother operation, robust algorithms needs to be formulated and in order to put these complex algorithms to real world usage, efficient systems need to be built. All computer vision algorithms are some sort of manipulation of pixel values. Machines interpret images very simply: as a series of pixel, each with their own set of color values. Each pixel in an image is usually represented by a number ranging between 0-255 (8 bits) for grayscale and a triplet of numbers, each ranging between 0-255 (24 bits) for color images. A normal sized 1280 * 560 color image is about 1280*560*24 bits, which is roughly equal to 17*10^6 bits or about 2.15 MB. That's a lot of memory to require for one image and a lot of pixels for an algorithm to iterate over. So in order to get successful results out of these algorithms, sheer amount of computing power and storage is required. Some of the core concepts of computer vision have already been integrated in to some of the major products that we use every day. Facebook is using computer vision to identify people from photos, Google is using maps to leverage their image data and identify streets, buildings, etc.

In this project we mainly focus on real-time motion detection which is one of the major areas of computer vision. This is a very widely used computer vision application. Some of the major applications of motion detection are surveillance, crime detection, pattern recognition such as pedestrian detection, vehicle detection, analysis of captured motion data (e.g.: helping athletes to improve their performance based on real time motion data analysis), and many more. Implementing this algorithm is not a complex task but the computational complexity of the algorithm makes the task challenging. It is highly essential to produce quick and accurate results in real-time scenarios. Despite significant advances in the CPU architectures over the years, it is still not possible to implement such an algorithm to handle huge amount of data and to produce quick and efficient results using only the CPU alone. Thus we need the computing power of the GPU to implement parallelism on every frame thus reducing the total operation overhead.

GPUs are gaining popularity as a platform for parallelized computations on massive sets of data. Since much of the computations in image processing and computer vision algorithms can be parallelized, graphics operation on the GPUs achieve significant speedup over their serial counterparts performed on the CPUs. Furthermore SDKs like NVDIA CUDA framework provide simple and easy to use APIs for leveraging the parallel computing power of the GPUs. However care should be taken to write efficient kernels to do meaningful work. Performance of the GPUs decreases with divergence, random memory accesses and data access synchronization. There is a need of large number of threads to do computations of each pixel in parallel without data corruption. So the main objective is to write work efficient kernels to produce quick and accurate results. Different techniques like shared memory usage, constant memory usage, streams, thread coalescing, pinned memory usage, etc. have been leveraged to accelerate the computations. The algorithm is highly parallelized and is tested over a range of videos with different sizes and resolution. The algorithm is also tested with real-time motion detection through webcam. OpenCV library is used in conjunction with CUDA to achieve the objective. Since OpenCV provides many useful functions to work with videos and images, it qualifies for the task at hand. Most of the serial implementation is done with inbuilt OpenCV library function (highly optimized). Finally the parallel implementation is compared with the serial implementation and analysis is done on the speedup and potential improvements of the parallel implementation.

## II. DESIGN OVERVIEW

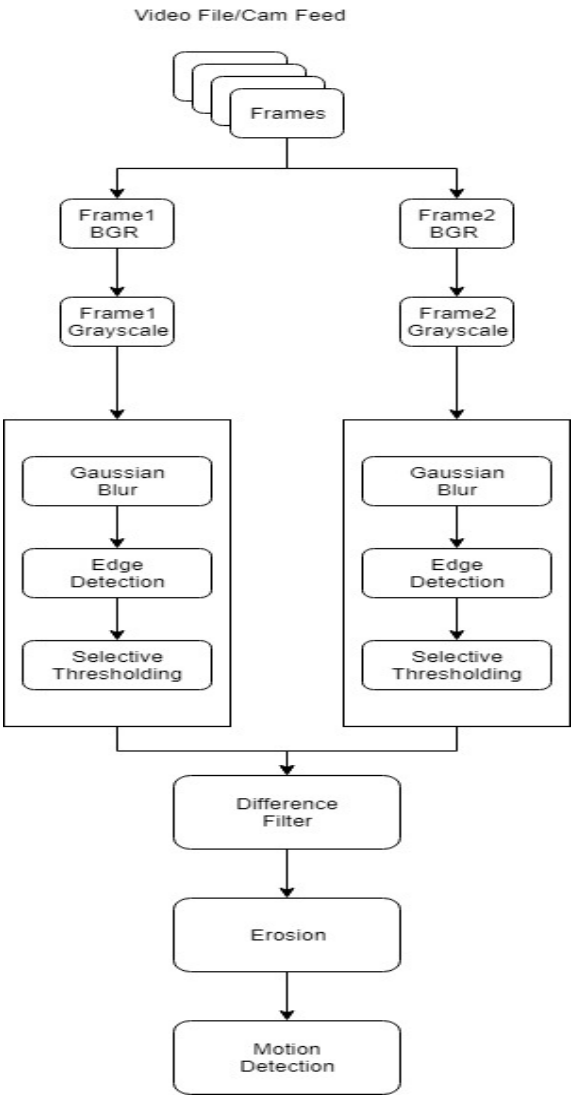


**Figure 1. Block Diagram of design architecture**

The basic idea of the design is to first read a video file and extract frames from the video. Then different filters (gaussian, Sobel and thresholding) are applied on the frames to detect the edges and then a pair of frames are compared to find differences in them. Difference between two consecutive frames indicate that there is a motion. Then erosion is applied on the difference result to make sure that we have successfully suppressed any noise or minor movements. Finally for each pixel of the filtered image we draw small rectangles on the original source frame to visually detect motion. We continue this process for all the frames in the video in order to produce a continuous video stream. In the next sub-sections we will go through the various steps involved in detail.

### A. Reading Video

To read a video file or web cam stream OpenCV is used. VideoCapture() class is used to read a video file. The OpenCV version that is being used in our project is 2.4.5 for the Linux machine and 3.4.1 for the Windows machine. To read a MP4 file FFMPEG is required. But in the Linux machines FFMPEG was not installed instead Gstreamer was installed. Gstreamer can read image files. So in order to check the performance in the Linux machine we converted the video file into a sequence of images and read the images as frames of the video in the same way that we do with a video. For this project we tested the output with many video files ranging both in size and resolution of the frames. Here the basic idea is to read two consecutive frames and convert the BGR frames to Grayscale for further processing. We continue this process for all the frames in the video, till we reach the end of the video or the user presses 'Escape'.

### B. Noise Reduction

Noise is a random variation of pixel intensity values. Digital images and videos are prone to various types of noise. It is the result of faulty image acquisition which results in values different from the original pixel values. There are many noise reduction techniques in the field of image processing. In our project we use a gaussian filter to remove the noise. Gaussian filter is a low pass filter. Basically it allows the low values to pass through and filters out the high frequency noise present in the frames. It uses a mask that has a gaussian distribution to filter an image instead of a simple averaging mask. This filter also introduces a smooth blurring of the image in addition to removing noise in the image. The gaussian mask that is being used in this project is as follows:

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

**Figure 2. Gaussian mask**

It is a 5*5 gaussian mask with an approximate standard deviation of 1. To compute the filtered output, the mask is convoluted with the source frame. Convolution is the process of adding each pixel of the image in its local neighbor weighted by the mask. The general expression of a convolution is :

$$g(x,y) = (\omega * f)(x,y) = \sum_{-a}^{a}\sum_{-b}^{b} w(s,t) f(x-s, y-t)$$

Where g(x,y) is the filtered image, w is the mask and f(x,y) is the original image. Every element of the filter mask is considered by -a ≤ s ≤ a and -b ≤ t ≤ b. For example, if we have a 3*3 filter mask and an 8*8 image as follows :

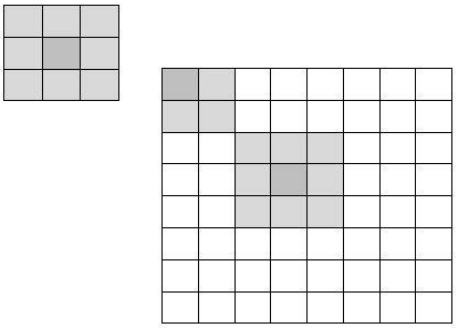

**Figure 3. Convolution of a 3*3 mask with an 8*8 image**

Each pixel of the 8*8 image is calculated by multiplying the 3*3 mask with the 3*3 neighborhood around each pixel and the summing up the results in that neighborhood. For the pixels that are in the border we pad zeroes and do the same computation.

### *C. Edge Detection*

Edge detection includes a variety of methods whose main objective is to find pixels in the image at which the brightness changes sharply or has discontinuities. Edge detection is one of the fundamental tools in computer vision and image processing and it is the primary step for algorithms like feature extraction, feature detection, motion detection, etc. Sometimes edges extracted from the images contains many non-trivial edges and it may also contain some discontinuities. So robust edge detection algorithms needs to be implemented to find the edges in the frames. Some of the popular edge detection techniques are Canny edge detection, Sobel operator, etc. In our project we implemented the Sobel operator. Edges correspond to the pixels with high gradient magnitude along the gradient direction. Gradient magnitude and angle at each pixel is computed using the following two formulas :

$$G = \sqrt{Gx^2 + Gy^2} \quad , \quad \theta = \tan^{-1}\frac{Gy}{Gx}$$

Where Gx and Gy are the partial derivatives along the x and y direction at every pixel location.

$$Gx = \frac{\partial A}{\partial x} \ , Gy = \frac{\partial A}{\partial y}$$

where A is the image and the partial derivative is calculated at every pixel.

Sobel filter is used to approximate the gradient of the image. The filter masks that are being used to approximate the x and y gradient are as follows :

| | | |
|---|---|---|
| +1 | 0 | -1 |
| +2 | 0 | -2 |
| +1 | 0 | -1 |

| | | |
|---|---|---|
| +1 | +2 | +1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

**Figure 4. Sobel x and y filter masks**

Then convolution is used to calculate Gx and Gy.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

**Figure 5. Convolution to find Gx and Gy**

After computing Gx and Gy we calculate the gradient magnitude and angle at each pixel then we send the frames for further processing.

### *D. Selective Thresholding*

After computing the gradient at every pixel location, the final step of the edge detection algorithm is to find out which pixels should be enhanced and which pixels should be suppressed. Since some of the edges detected by the Sobel filter are not useful, we need to suppress those non-maximum edges. We have implemented Non maximum suppression(NMS) technique which is used for this purpose. NMS accepts and rejects pixels based on the following criteria:

a) Pixels with gradient magnitude below a user defined threshold (low) are suppressed.
b) Pixels with magnitude above a user defined threshold (high) and also above the magnitude of the pixels in the gradient direction are enhanced.

Then after that we apply selective thresholding around the pixels to identify the points as edges that might have been missed by NMS. In selective thresholding we define a mask around pixel and for every pixel within the mask we check if the magnitude value is more than the low threshold. If the value is more, then those pixels are enhanced. By doing this for every pixel in the image we make sure that all the pixels are identified successfully as edges or non-edges. This technique attempts to catch the pixels that NMS might have missed to create continuous edges. After this step we get an image where important edges are enhanced and the non-maximum edges are suppressed. These edge frames are then used detect motion.

### *E. Difference Map*

After getting the edges of two consecutive frames using edge detection, their difference is checked to detect motion. If there is a large difference between the two frames then there is a motion. If there is no motion, then the pixel values for two consecutive frames will remain same. But if there is a motion then the pixel values of the two consecutive frames will be different. We compute the difference using the following formula :

$$D = |E1 - E2| == 0 ? 0 : 255$$

where E1 and E2 are the edges of two consecutive frames.

We use this formula for every pixel in the frames. We compare corresponding pixels from the two frames and assign value based on :

a) If the difference between the two pixel values is 0 then we assign 0 in the difference map.
b) If the difference between the two pixel values is not 0 then we assign 255 in the difference map.

So after this step we will have a difference map which is of the same size as that of the frames. Every pixel of the difference map will either be 0 which indicates no motion or 255 which indicates motion. Then after computing the difference map, we use this map to detect motion on the original source frames.

### *F. Motion Detection*

The difference map of two consecutive frames contains motion information. But some of the information may be faulty or of less importance. For example if there is a very slight motion which is not of our interest or if there is noise then those pixels indicating motion should be rejected. For this purpose we use an erosion filter. Erosion sets a center pixel to the minimum over all the pixels in the neighborhood. The neighborhood is defined by the structuring element, which is a matrix of 0's and 1's. Once we apply erosion on the difference map, any noise that was present or any minor motion that was previously detected are now rejected.

Now we loop over all the pixels in the difference and check whether its value is 0 or 255. For all the pixels whose value is 255 we draw a rectangle on the original frame for that corresponding pixel to show motion. So for all the pixels with a value of 255, rectangles are drawn on the original RGB frame. We continue this process for all pairs of frames and keep on displaying the output within a short interval. This creates a continuous effect and motion is detected over the frames

## III. *IMPLEMENTATION USING GPU*

We have applied the convolution kernel, Pythagoras kernel and Thresholding and separation for each frame and we have applied difference kernel for a couple of frames to identify the movement. The following figure is just a frame on which the following kernel operations will be applied.

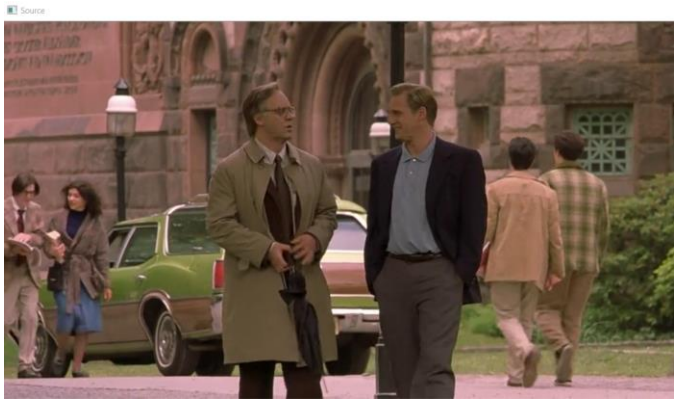

**Figure 6. original image**

### *A. Convolution Kernel*

This kernel is used to perform convolution operation for Gaussian filter, gradient X of Sobel filter and gradient Y of Sobel filter it takes the parameters such as source data, Source height, source width, Kernel size, the output pointer where it returns the data and the flag based on which it chooses the masks shared memory. For example for flag value 1 it performs convolution on the Gaussian mask, for flag value 2 it performs convolution on the Sobel x mask and for flag value 3 it performs convolution on the Sobel y mask. In order to achieve more speed up, we have stored the masks in constant memory. We have also used the shared memory to load the blocks so that we can use the maximum tile reuse to improve the performance. The shared memory size is equivalent to the tilesize+masksize-1 * tile size + mask size-1. Since we know the maximum no of threads per block can be 1024 we are dividing the block in such a way that it uses the maximum no of threads so based on the masks we have defined the tile size so that the block size can be 32*32.The following output image can be found after we have applied the Gaussian filter with the with the Gaussian convolution mask.

**Figure 7 original image After gaussian blur**

### *B. Pythagoras Kernel*

After applying the Sobel x and Sobel y convolution kernels the Pythagoras kernel takes the output of Sobel x and Sobel y convolution calculate the gradient magnitude and the gradient angle formed at that pixel. It takes the input Each thread of the block is responsible for a single pixel. and

calculates We have tried the fast math methods for square root and tan2 methods. The following figure shows the Sobel edge detection output after applying the convolution filters (Sobel x, Sobel y) and the Pythagoras filter.

**Figure 8. original image after gaussian edge detection**

*C. Thresholding and suppression kernel*

This kernel is used to calculate the threshold of the images based on their neighborhood values. We are taking the input values of the frames Gradient magnitude and Gradient angle calculated by the Pythagoras kernel and we are loading its 3*3 neighborhood information of each point into the shared memory and then we are calculating the currents pixel value based on the input(high threshold and low threshold value).If the pixel value is below the low threshold we are making the pixel value to zero if the pixel value is above the high threshold limit we are making it 255 and then we are also looking for a 3*3 neighborhood and the gradient angle to find the pixel whose value is in between low threshold and high threshold value and making them 255. The following image is obtained after applying the Threshold and suppression filter.

**Figure 9. original image after thresholding**

*D. Difference kernel*

This kernel is used to calculate the difference of pixel values of the output of two threshold frames from the previous kernel. It also takes the threshold parameters as an input where it identifies the difference of pixel values of a neighborhood instead of a single pixel. We are taking two 32*32 block size matrixes to detect the motion. The following image is obtained after we apply the difference filter

**Figure 10. image after difference filter**

Then we apply the erode filter to remove high-frequency noises and then we add blue color rectangles to detect the motion of the output based on the output of the difference filter.

**Figure 11. original image after all the processing**

## IV. *IMPLEMENTATION USING CPU*

A. cv::GaussianBlur(source, blurred, cv::Size(5, 5), 0)

This library method of open cv is used to apply the Gaussian blur of mask size 5*5 on each frame.The first parameter is input and the second parameter represents output and the third parameter represents the details of the mask.

*B. cv::Sobel(blurred, grad_x, CV_8U, 1, 0, 3, scale, delta, cv::BORDER_DEFAULT);*

This library method of open cv is used to calculate the gradient magnitude of the Sobel filter.

*C. cv::convertScaleAbs(grad_x, abs_grad_x)*

Convert the fractional value to the absolute value.

*D. addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0, edges)*

This method is used to calculate the total gradient value by taking the square root of the input values.

*E. serial_thresholding_and_suppression(unsigned char *output, int input_width, int input_height, unsigned char *g_x, unsigned char *g_y, int high_threshold, int low_threshold)*

This method is used to calculate the threshold values of each pixel based on the 3*3 neighborhood value. It takes the input image its height ,width its gradient magnitude in x direction and gradient magnitude in y direction and takes the low and high threshold values.

*F. serial_difference_filter(difference, edgesThreshold.data, edgesThreshold2.data, frame.size().width, frame.size().height,0.5)*

It takes the two threshold data frames and find the difference between them based on their pixel values.

*G. cv::erode(difference_img, difference_img, elementErode)*

This library method of open cv is used to remove the high frequency noises from the given image

*H. cv::rectangle(motionFrame, cv::Point(j - 5, i - 5), cv::Point(j + 5, i + 5), cv::Scalar(255, 0, 0), 1);*

This open cv library function is used to plot the rectangle on the given input image.

## V. *PERFORMANCE*

The Initial aim was to achieve 3x speed up from the sequential version. As mentioned in the Acceleration of Edge-Based Motion Detection and Machine Learning-Aided Facial Recognition with NVIDIA CUDA" paper the speedup with the resolution can be found as follows.

| Image dimensions | CPU time (s) | GPU time (s) | Speedup |
|---|---|---|---|
| $100 \times 100$ | 0.067883 | 0.061132 | 1.110433 |
| $500 \times 500$ | 0.226192 | 0.094812 | 2.385690 |
| $1000 \times 1000$ | 0.811302 | 0.199005 | 4.076792 |
| $2000 \times 2000$ | 2.849201 | 0.562820 | 5.062366 |
| $3000 \times 3000$ | 6.236422 | 1.218879 | 5.116523 |
| $4000 \times 4000$ | 10.794892 | 2.142449 | 5.038576 |

But we have achieved 5-10x speed up. We are gaining the speed up based on the resolution of the image and size of the image. For small size and small resolution of the image the speedup is minimal and if we increase the size we are gaining more speed up. With the different size of the image, we are getting different performance. we are getting more speed up with the increase in the size of the image data. for example for the size of 5.28 MB we are getting the speed up of 4.7x However for the file size of 211 MB we are gaining the speed up of 4.9x and for the file size of 317mb we are gaining speedup of 5.1x. The time taken for each of the each of the above image is shown as the following figure.

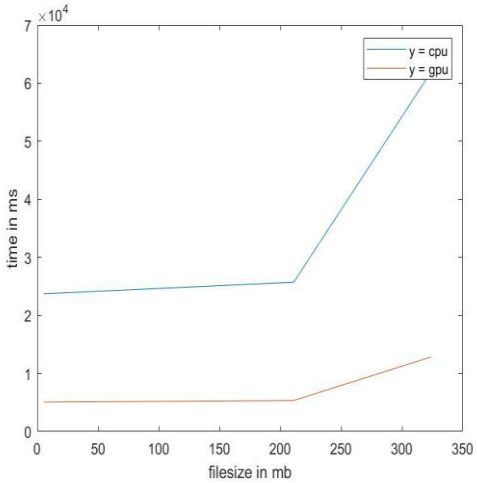


**Figure 12 file size vs Time plot**

As shown in the Above graph the time of computation increases as we increase the size. The red line represents the time taken by the GPU while the blue line represents the time taken by the CPU. The time taken for GPU for the size of 5.28mb, 211mb, and 324 MB is 5058 milliseconds,5331 milliseconds,12838.1 milliseconds. The CPU time taken for the above-mentioned size are 23773 milliseconds, 261222 milliseconds and 654740 milliseconds respectively. With the increase of resolution, the speedup also increases for example for the resolution of (720*400) we are gaining the speed up of 4.82 with the image of resolution (1008*560) we are gaining the speed up of 4.91x and with the image of resolution (1280*560) we are gaining the speed up of 5.8x. The time taken vs the resolution graph is obtained as shown in the following Figure. The Blue Line represents the time taken by the CPU while the red line represents the time taken by GPU. The time taken for GPU for the image of resolution (720*400), (1008*560) and (1280*560) are 2880 milliseconds, 5058 milliseconds and 5671.2 milliseconds. The time taken by the CPU for the above three resolutions are 13880 milliseconds 24781 milliseconds,32893 milliseconds respectively.

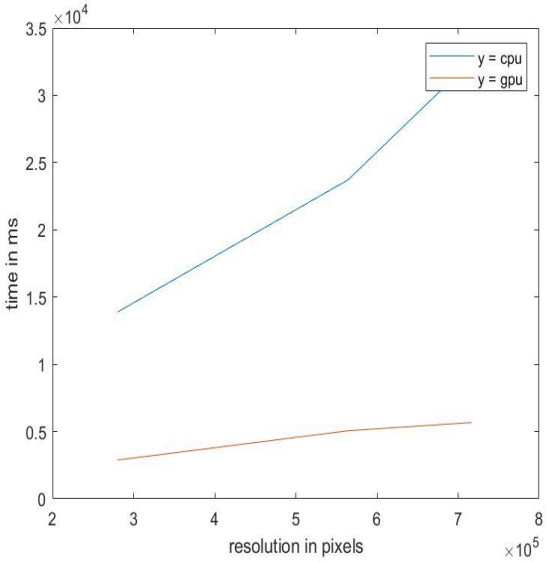


**Figure 13 resolution vs Time plot**

As we can see the above figure the time taken by CPU increases as the resolution of the picture increases compared to the GPU's time. If we analyze each and every kernel of the project it can be found that the project is compute bound. The occupancy and limiting factor of each kernel can be represented in the following table.

| Kernel Name | Occupancy | Limiting Factor |
|---|---|---|
| Convolution | 100% | No of threads per block |
| Pythagoras | 100% | No of threads per block |
| Difference | 100% | No of threads per block |
| Thresholding and suppression | 100% | No of threads per block |

*VI. Challenges And Improvements*

We have faced some challenges in running the code in the GPU lab machines. Since the GPU lab machine does not have a camera we can't process the Realtime image in the GPU lab. So We tried to process the videos in it. But we could not process it since it has the older version of openCV and it does not contain FFmpeg which is used to read the video. So we converted the video into a set of images using https://www.filezigzag.com and applied our algorithm to measure the speed up. We tried to use fast math to increase the performance but we didn't get much performance improvement. We also tried to improve the performance by using maximum no of registers. But we did not get much performance improvement by doing it.We have run NVPROF to analyze the kernels and the result is found in the following figure.

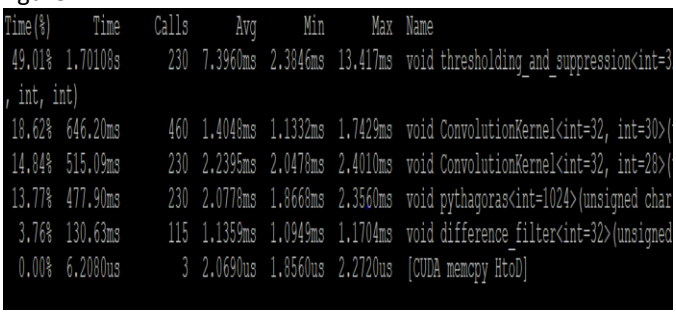


**Figure 14 NVprof results**

As we can see the void Thresholding and suppression is taking the maximum time and this method can be optimized.We are using 3*3 shared memory for reading the neighborhood of input data. The shared memory can also be used to calculate the output and store it in global memory once instead of accessing the global memory multiple times.To remove the high pass noise we are using Erosion filter of OpenCV but we can implement it using the GPU for more performance gain. We are using the rectangle plot function of OpenCV to identify the movements in the image. This function can be implemented in the GPU for more performance gain.

If the input data is very large we can divide the data into chunks and techniques like double buffering and GMACS can be applied.

## VII. Conclusion

The implementation was successful as we achieved the objective of 3 times speed up with 5-10 times speed up in practice. We were successful in using the tools of parallelism learned in the class like usage of shared, constant memory, streams, fast math, pinned memory, Used float instead of double, thread coarsening and used maximum no of registers. The parallelism helped to handle two frames simultaneously which helped to speed up the process. However, despite the successful implementation, there are still more improvements required to make the code applicable in real-time operations. Improvements can be achieved by using better data transfer techniques, parallel implementation of Erosion filter and parallel implementation of Rectangle drawing function.

The various versions throughout the development of this project can be found on our Github repository here: https://github.com/anubhavpanda2/EE-5351-Realtime-Motion-Detection-with-cuda.

*VIII ACKNOWLEDGEMENT*

We are very thankful to Professor John Sartori and Teaching Assistant Himanshu Shekhar Sahoo for their constant guidance and support. We are also thankful to the ECE-IT department for the access to GPU LAB computers throughout the course to run our codes.

*IX. REFERENCES*

[1] Hana Ben Fredj , Mouna Ltaif , Anis Ammar , Chokri Souani," Parallel implementation of Sobel filter using CUDA"

[2] ICCAD'17, Hammamet - Tunisia, January 19-21, 2017

[3] Rasmus Rothe, Matthieu Guillaumin , and Luc Van Gool,"Non-Maximum Suppression for Object Detection by Passing Messages between Windows". ACCV 2014: Computer Vision -- ACCV 2014 pp 290-306

[4] Emilio Del Vecchio, Kevin Lin, Senthil Natarajan, "GPU Acceleration of Edge-Based Motion Detection and Machine Learning-Aided Facial Recognition with NVIDIA CUDA"

[5] "Hands On GPU Accelerated Computer vision with openCV and cuda" by bhaumik vaidya

[6] Converting the files to images https://www.filezigzag.com

[7] CUDA Toolkit Documentation :

[8] http://docs.nvidia.com/cuda/