Parallelization of Lane Detection and Obstacle Detection on Nvidia Tegra X1 SOC

Anuja Kulkarni
Ravi Teja Reddy
Sri Akhilesh Joshi
Department of Electrical and Computer Engineering
University of Florida
{anuja30k, rraviteja21, srijoshi} @ufl.edu

***Abstract*—This project focuses on developing, parallelizing and analyzing algorithms for road lane and obstacle detection, on the Nvidia Tegra X1 SoC. The performance of the system is analysed in depth with respect to speedup and granularity. Additionally, the performance of the system has also been analysed on the Fermi GPU, to observe Tegra X1's performance in comparison. We have also executed the application on OpenCV CUDA to determine the impact that pre-defined image processing libraries have on performance, as compared with Native CUDA C.**

I. INTRODUCTION

Road safety is an important issue in the current scenario, considering that most of the population in developed countries owns a personal vehicle. It is crucial for drivers to remain alert all the time. However, accidents do happen due to simple distractions. Drifting from lanes and inability to see obstacles in the path are some of the causes of a large number accidents. This highlights the importance of the applications of lane and obstacle detection.

The reliability of these applications is largely dependent on the speed at which a response can be generated. A system which issues a warning after a collision has occurred is no use at all. This is a potential problem in many current implementations that do not make use of parallel computing. Such systems are usually specialised modules integrated into expensive vehicles.

Our project puts into effect lane and obstacle detection for vehicles on a parallel computing platform, the Nvidia Tegra X1 SoC. The system detects if the vehicle strays away from the current lane, and also when there is an obstacle in close proximity, ahead of itself. The purpose of this project is to add a low cost, portable solution to the class of devices built for driver assistance. The objective behind parallelizing this application is to provide faster feedback to the driver, allowing him/her sufficient time to take necessary action to avoid road mishaps.

II. RELATED RESEARCH

The advent of computer vision has improved safety and comfort features in the automobile industry. Some existing applications monitor the driver and ensure that he/she is alert all the time. Signs of drowsiness, cellphone calls and other distractions are continuously monitored. In addition to providing support inside the vehicle, computer vision also finds application in monitoring of surroundings, such as lane detection to ensure that a driver does not unintentionally stray into another lane, obstacle detection for collision avoidance. Road signs can also be interpreted to give a friendly warning to the driver in case of violation. Other such complex applications include self-parking, accident mitigation, parking assistance, blind-spot detection [1].

Among these applications, research was conducted extensively in lane detection and obstacle detection. The paper [2] compares various techniques used in edge detection, specifically in the field of lane detection. They compare the following filters: "Canny, Prewitt, Sobel and Roberts." According to their results, the Robert Filter is the fastest and also simple to implement. However, this filter fails to detect dashed lane markings which are crucial to our application. Also according to [3] the Canny filter shows a much better resistance to noisy conditions, as compared to the Sobel and Prewitt filters. Therefore, in spite of slightly slower performance, we chose to use the Canny filter (which is the most commonly used in all lane detection applications) over others, giving priority to accuracy.

[4] Implements lane detection using the Canny filter followed by Hough transform. The Hough transform detects straight lines (lane markers) from the Canny filter output. It also combines discontinuous lines. Thus, the final location of boundaries for the lanes can be found. [5] deals with vehicle recognition using static cameras stationed above roads. To extract the background, multiple images are captured by the camera and their median is calculated. The final image is one which contains only the background, without any vehicles. After this initial phase, the system is ready to function. A subtraction with the background image is performed for every image captured. This leaves behind only new objects that have been introduced into the frame, such as vehicles. The image is then converted to binary and pixels in close proximity of each other are merged, to form a single "blob" representing each object. The same approach cannot be applied to our project, due to the fact that the background changes with each frame. However, the concept of creating blobs is retained and we use the morphological close operation to implement this.

III. PLATFORM AND TOOLS

The Tegra X1 is the first of its kind system on chip. It is a super computer built for tablet applications and consists of over 256 cores. It is designed based on the Maxwell architecture.
The Nvidia Tegra X1 SoC provides a low cost, portable alternative to the current framework. This, combined with its energy efficiency and ability to perform operations up to 1 TFLOP in just 4W makes it highly suitable for our application.

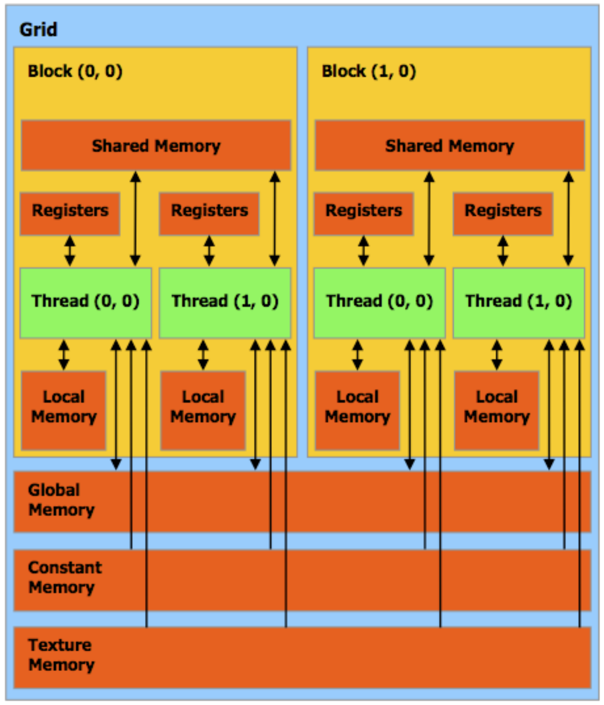


Figure 1 CUDA Architecture [9]

The Tegra X1 supports programming in CUDA, the standard platform for most Nvidia devices. The standard CUDA based system comprises of a host processor and a GPU device (the accelerator). Processing on the GPU is partitioned into large units called grids. Each grid is comprised of several blocks, which in turn are composed of threads. The number of threads is customisable, based on programmer needs. Generally, one thread is assigned to every operation that is supposed to occur parallelly across the system. CUDA provides directives to synchronize these threads as required.
Streaming Multiprocessor: The streaming multiprocessors (SMs) are the part of the GPU that runs our CUDA kernels.
CUDA scheduler: CUDA scheduling comes into picture in places where the the number of threads were less than the product of height and weight. It follows "round-robin" fashion of scheduling.
cudaMalloc(): cudaMalloc() allocates the specified amount of memory on the GPU.
cudaMemset(): cudaMemset() initializes to a value on the chunks of memory specified with the command.
cudaMemcpy(): It copies blocks of data from "hosttodevice" or from "devicetohost".
cudadevicesynchronize(): This makes sure that all the threads wait here before executing the next kernel. It is a "barrier synchronization" for all the threads in the all the blocks.
__*syncthreads():* This makes sure all the threads in a block wait at this point.

All the threads in the block use shared memory and transfer the data via shared memory. But if threads in different block need the data then it has to be fetched from global memory.
For image processing applications, OpenCV provides GPU support to allow parallel image processing on Nvidia's accelerators. However, there is less programmer control over optimization and host-device communication in the OpenCV libraries. In our project, we have chosen to implement all algorithms using native CUDA C, without using any in built libraries.

IV. APPROACH

Since our chosen application falls in the domain of image processing, it has good scope for parallelism. We execute most of the major operations occur on the GPU, where the functionality is defined by CUDA. Since we are implementing the application in native CUDA C, we have a large amount of control over the thread and block assignment for every task and also the amount of communication that occurs between the processor and the accelerator.

In order to perform real time lane detection as well as obstacle detection, we first extract frames from a running video using OpenCV. The raw image of the road is then converted to grayscale and thus provides as an input for Canny Edge detection Kernel, for further processing. CUDA is used to copy the large matrices from the host (the ARM processor) to the GPU.

This CUDA execution of the project has been divided into three main kernels:

Canny Edge Detection

The Canny Edge detection consists of the following stages, each of which has been implemented as a separate kernel. The reason for using individual kernels rather than a single one is discussed in section 5.

- *Gaussian Filtering*: Gaussian filtering involves using a mask of a certain size and performing a Gaussian convolution operation on the entire image. The computation of the mask from a given value of sigma is performed on the CPU, rather than the GPU. This is because the size of the mask is usually very small (around 10x10), and the cost of computing it serially on the CPU is less than the communication cost incurred in sending it and assembling the results from the GPU. The computed mask is sent to the GPU for convolution with the image, where each thread calculates the value of one pixel of the image. The Gaussian Filtering reduces salt and pepper noise, caused due to grainy-ness of the image. Thus the appearance of unnecessary edges in the output is avoided.
- *Sobel Filtering:* The Sobel filter is used to perform preliminary edge detection on the image. The x and y gradients of the image are computed by performing a convolution with sobel masks which are given by:

$$\mathbf{Gx}= \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

$$\mathbf{Gy}= \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

  Convolution is performed parallely in the same manner as the previous step.Each thread calculates the x and y gradients of the corresponding pixel. The gradient of the pixel is then calculated by using the formula: |Gx^2+Gy^2|.
- *Edge Direction:* The direction associated with each pixel is determined using $\Theta= \tan^{-1}(Gy/Gx)$. With the direction obtained, each pixel can be oriented in an approximate direction (0, 45, 90 or 135 degrees).

- *"Non maximum Suppression":* This procedure helps eliminate blurry edges. Each thread determines the edge strength of the corresponding pixel and is compared with those along the same gradient. If the strength is found to be less than the maximum value, it is suppressed, else retained.
- *"Double Thresholding"* : This is used to create a continuous edge line from dashed, uneven lines. Two thresholds T1 and T2 (T1>T2) are set. Pixels with edge strength > T1 are marked as edges. Pixels that are in contact with these and have a strength >T2 are also marked as edge pixels. Remaining pixels are set to 0 (non-edge).

2. Hough Transform
This kernel is used to isolate certain standard features such as lines, curves, circles, etc. In our case, it is used to detect lines in the output image of the Canny Edge Detector. To detect a line in a image, a voting scheme is employed using the polar coordinates (r,θ) by each thread where r is the perpendicular distance from the line to the origin and θ is the angle between the axis and the line (**r = x cos θ + y sin θ**). All the threads check whether the corresponding pixel satisfies the condition for the straight line, and then a vote is placed atomically in the voting array if it satisfies. In many cases, the number of votes for a particular pixel in a polar coordinate is more than 255. In order to scale this down to 255, the maximum member of the voting array is calculated atomically and every element in the voting array is divided by 255 by every thread. The figure formed with this process is known as "Hough Space Graph."

Mathematically, a curve is drawn in polar coordinate system for every pixel which satisfies the straight line condition. The straight line condition is that the value of r has to be greater than zero and has to be less than square root of sum of height and width($\sqrt{x^2 + y^2}$).

Now, in order to draw the straight lines in the x-y coordinate system, each value of r is computed for every possible theta (-90 to 270 degrees) is computed by every thread and checks whether the value is 255 or not. If the value is 255 and greater than the threshold, then a 255 is put in the pixel corresponding to (**r**,**θ**) in x-y plane. Collection of all the dots forms straight lines in the x-y plane.

3. Obstacle detection
The next goal is to detect vehicles or other obstacles that are immediately in front of the car, and too close for safety. The output from the Canny Edge detector as well as the Hough Lines can be used for the detection of the obstacle. The Hough Lines output is used to determine the lane boundaries. In the bottom half of the image, region lying within these boundaries is then scanned in the Canny Edge detector output. Obstacle detection is carried out through a "morphological closing" operation, which is implemented using two CUDA kernels, wherein every output pixel value is calculated by individual threads:

1. *Dilation:*

- A structural element (square matrix consisting of 1s) of a suitable size is defined.
- The value of every output pixel in the image is computed by every thread as the maximum value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to the value 1, the output pixel is set to 1.
- A variable "count" holds the number of 1s found in the neighborhood of a pixel. If greater than or equal to 1, the output pixel is set to 1.

2. *Erosion:*

- Erosion is the "dual" of the dilation operation. The same structural element used in dilation is used here to erode away excess edges.
- Threads determine the value of the output pixel as the minimum value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to 0, the output pixel is set to 0.
- The same variable count is used in this case too. In this case, the output pixel is set to 1, if the value of count is equal to the mask size.

3. The morphological closing of the image results in a “blob” like structure, representing the obstacle we wish to detect.

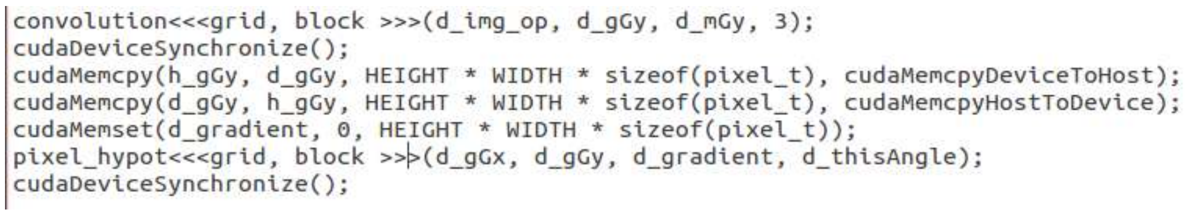
V. ROADBLOCKS

Canny Edge Detection

The Canny algorithm consists of five major steps. In order to reduce memory usage by multiple matrices and also communication time to transfer these matrices to and from the GPU, we attempted to use a single kernel for the entire algorithm, with reuse of matrices. Reuse of matrices and random thread scheduling leads to incorrect outputs, caused due to race conditions in various stages, especially in convolution operations, where a single thread needs to access neighbouring pixels to determine an edge. To avoid this, we moved on to using separate kernels wherever possible, and separate matrices for intermediate inputs and outputs. The drawback of this methodology is an increase in communication overhead, which leads to lower speed of the system.

Hough transform

1. The votes that are placed in the voting array are simply an increment of previous value in the array. This doesn’t cause any problems in serial code but in parallel code it causes race conditions. To tackle this issue, the “atomicAdd” function in CUDA was found to be useful. However, this operation serialized all the increments which resulted in significant performance loss.
2. It took us a long time to discover that one of the kernels involved in the Hough Transform, required more threads in the polar coordinate system than in the x-y coordinate system. For this reason, this kernel was separated from the rest because it requires more number of threads than any other kernel. This will save power because unnecessary launching of unnecessary threads is not done. But, the disadvantage is that communication time is more for the kernel because the previous kernel has to transfer its output to this kernel.



```
convolution<<<grid, block >>>(d_img_op, d_gGy, d_mGy, 3);
cudaDeviceSynchronize();
cudaMemcpy(h_gGy, d_gGy, HEIGHT * WIDTH * sizeof(pixel_t), cudaMemcpyDeviceToHost);
cudaMemcpy(d_gGy, h_gGy, HEIGHT * WIDTH * sizeof(pixel_t), cudaMemcpyHostToDevice);
cudaMemset(d_gradient, 0, HEIGHT * WIDTH * sizeof(pixel_t));
pixel_hypot<<<grid, block >>>(d_gGx, d_gGy, d_gradient, d_thisAngle);
cudaDeviceSynchronize();
```

For example, in the code snippet shown above,we can see that between the two kernels *convolution* and *pixel_hypot* some serialization is performed by *cudaDeviceSynchronize()* and *cudaMemcpy()*. These operations significantly affect the performance and the computation to communication ratio.

Morphological closing

When we tried using the Canny Edges alone to detect obstacles, the output obtained was not sufficiently reliable. For this reason, morphological closing operation is made dependent on both the previous algorithm outputs. This implies that it cannot be performed until the previous two are complete. Hence, we compromised on some of the speedup to maintain the reliability of the application.

VI. RESULTS

The images shown in Figure 2 were found to be satisfactory outputs for our application and were used for verifying correctness and applicability of each of the discussed methods to our proposed system. (b) shows the edges present in the image. (c) shows the Hough image which is in polar coordinate form. The Hough image is a collection of curves of all the points passing through a straight line. (d) shows the two outer lanes of the road which converge at a point. Even the middle lane is shown in the image. (e) presents the obstacle (in white) which is the car. For morphological closing, only half of the image was considered. This helps in easy identification of the potential obstacle.

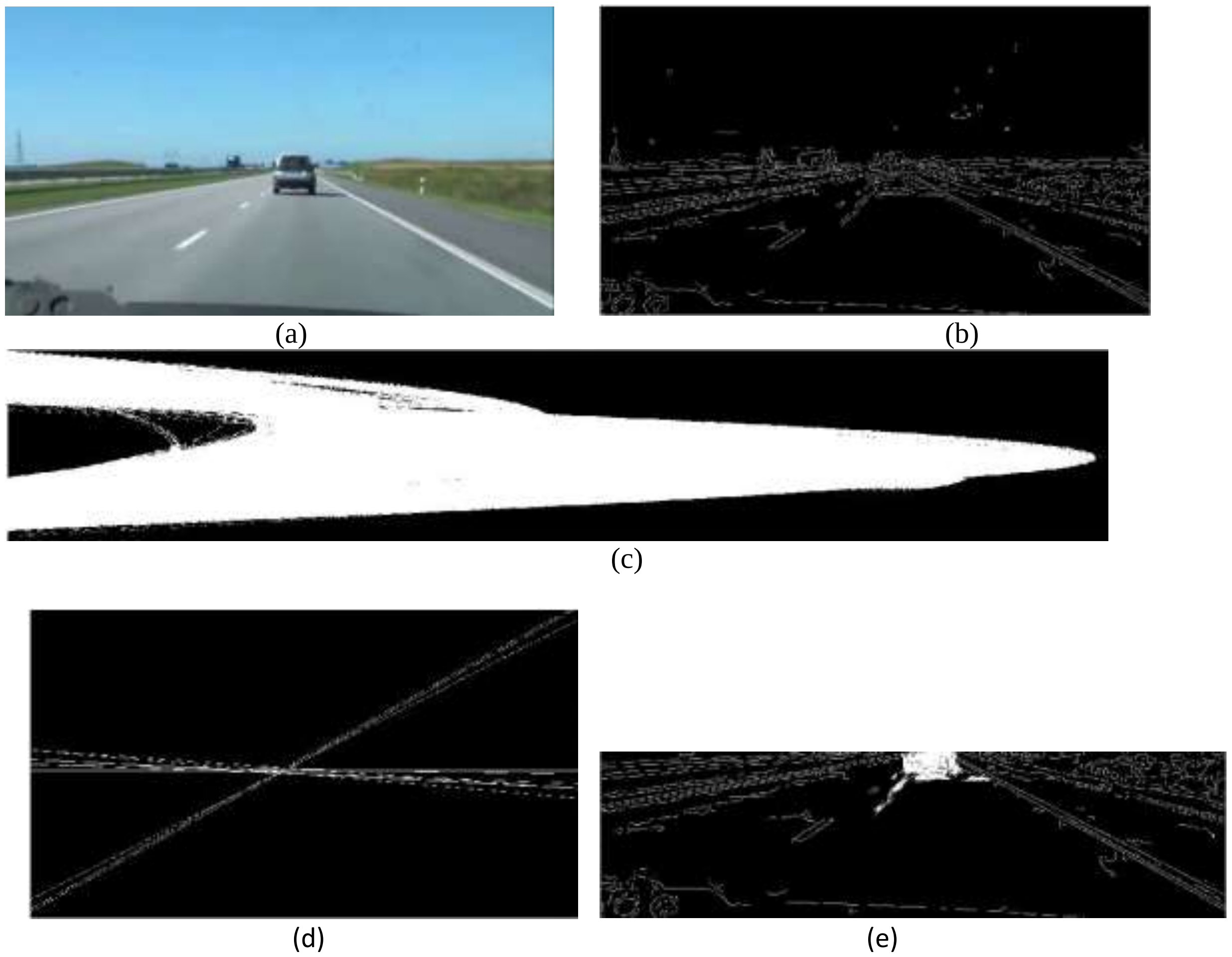


Figure 2 Outputs of Lane Detection: (a) Original Image (b) Canny Edges (c) Polar Hough representation (d)Hough lines (e) Morphological closing

VII. EXPERIMENTS

The following experiments were performed in order to analyze the performance and scalability of our application. We have considered standard mobile and tablet screen resolution sizes for our analysis:

1. Speedup of the individual kernels for varying image sizes: The speedup of the entire application depends on the performance of its individual kernels. If one kernel is much slower than others, the overall speedup is hurt. For this reason we wish to analyze each kernel's performance, to identify bottlenecks. We expect the Hough kernel to be the fastest, since it is computationally intensive, as against the other two, which are memory intensive.
2. The speedup of the system as a whole for varying image sizes: We analyze the speedup of the integrated system and draw conclusions based on what we observed in the previous section.
3. Effect of varying the number of threads per block:

- Impact on speedup: As the number of blocks decreases, fewer memory accesses are made by threads in order to access contents of another block. This improves speed and we expect to see an increase in speedup.
- Impact on granularity: We wish to analyze this case to understand how the computation and communication are affected by block size.

4. Effect of varying the size (number of pixels) of the image on granularity: As the problem size increases, number of threads increase too. An increase in computation is expected to be observed.

5. Comparing performance of the application on TX1 against its performance on the GPUs in Hipergator (Fermi): Tegra X1 is an SOC specially designed for mobile platforms. Its performance is significantly higher than predecessor Tegra K1. Its performance capabilities come close to Fermi GPUs, designed for high end computing. Therefore it is interesting to compare the Tegra with Fermi.
6. Comparing performance of the system implemented in native CUDA-C with the same implemented using CUDA libraries in OpenCV

VIII. ANALYSIS AND OBSERVATIONS

1.Speedup of the individual kernels for varying image sizes:
All the three kernels appear to show significant speedup with respect to the serial case for all sizes of the input pixels. But the rate of improvements among all the kernels are not same. For instance, "Canny edge detection algorithm" shows the least speed up of 7 while the "Hough Transform" shows the highest at 375. The "morphological closing" operation has a speedup of 25.

The speed up of "Canny algorithm" is significantly less due to the following reasons. Firstly, the algorithm is a lengthy one, requiring GPU memory allocation for at least 10 matrices prior to launching the first kernel. Cudamallocs are slow and allocate memory in the global space, thus increasing the read and write time. Secondly, during the convolution operation, each thread accesses the global memory more frequently than in other algorithms. This is because iteration through neighboring pixels adds to computation time. Barrier synchronization is used wherever necessary. This step is only as fast as the slowest thread, therefore affecting the speed.

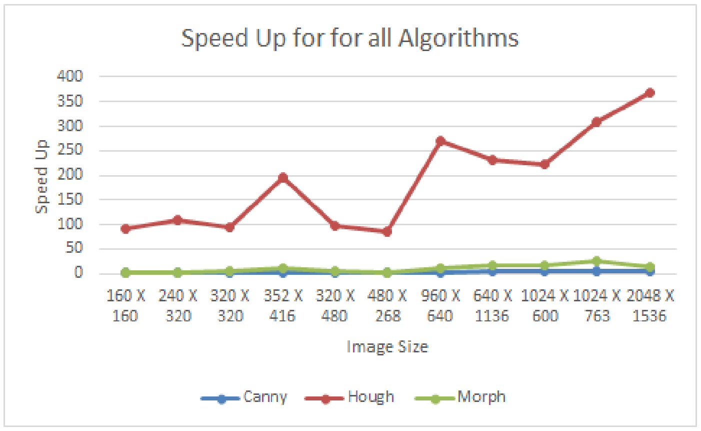


Figure 3 Speedup of kernels

On the other hand, the Hough Transform is an embarrassingly parallel application. Even though 6 Cudamallocs were required, the inherent massive parallelism compensates for their negative impact. The Hough transform is computation intensive rather than memory intensive. Therefore, as number of threads increase, hough transform becomes faster. This behavior is also found in Figure 3.

Morphological closing comprises of dilation followed by erosion to detect the obstacles on the road. The bottleneck in this algorithm is similar to the one in Canny. It occurs when each thread checks the surrounding pixels in order to compute the current pixel value. So, the computation per thread is of order O(k*k) where k is the structural element (SE) size. Once again, barrier synchronization is required and it impacts speed.

2. The speedup of the system as a whole

It is observed from the graph the speedup increases almost linearly as the problem size increases. The GPU launches more threads to deal with the increase in number of pixels. Since each thread operates on an individual pixel, the time consumed by the parallel program does not scale up linearly as was the case in the serial program. The time complexity of each kernel reduces almost to a constant value (if no data dependencies).

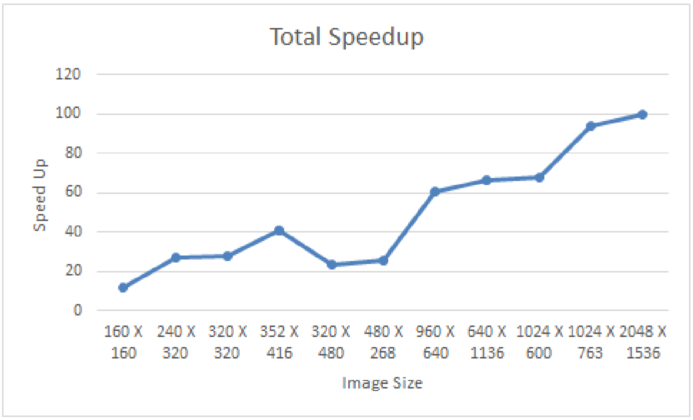


Figure 4 Image size vs Total Speedup

It is also observed that all the three algorithms shows a local peak in speedup for the image sizes of 352 x 416 and 960 x 640. It is observed that if the number of pixels are exactly divisible by the maximum number of threads (1024 * number of blocks), the speedup increases significantly. In case, it's not perfectly divisible (for example 480x268), most of the threads in the last block wait idly without performing any computation. This incurs a loss in speed up, computation to communication ratio and power consumed because global memory is accesses. This leads to bus overload. It can be concluded that input size must be divisible by 1024 for optimal performance.

Also, the speed up shows a slow linear increase with problem size, since the massive speedup provided by the Hough Transform is equalized by the lower speed Canny and Morphological operations. To further reduce computation time, latency hiding can be employed. This can be achieved by making the Hough Lines and Morphology transforms independent of each other by using only the Canny output for the closing operation. This modification allows to reduce the over load and improves the speedup. However, the output of the Canny edge detection should be appropriately enhanced at the cost of performance.

3. Effect of varying number of threads per block

a. Speed Up

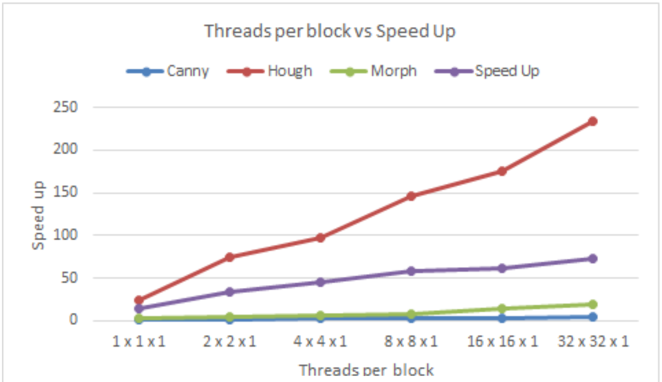
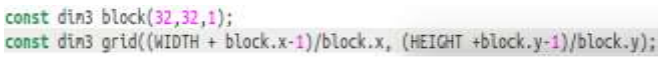


Figure 5 Threads per block vs Speedup

As the number of threads in a block increases, the total number of blocks decreases (since the image is of a fixed size) Calculation of the number of blocks is shown in the code snippet below. We also know that threads within a block can communicate while threads within different blocks cannot.



```
const dim3 block(32,32,1);
const dim3 grid((WIDTH + block.x-1)/block.x, (HEIGHT +block.y-1)/block.y);
```

Therefore, with increase in number of threads in a block, the likelihood of a thread attempting to access data that is present in other blocks is lessened. This in turn reduces number of accesses to global memory (when a thread cannot access data from another block, it fetches it from the global memory). Because of this there is an improvement in speedup with decreasing number of blocks.

b) Granularity

We consider a 1024 x 1024 image for analysis. Block size is calculated based on the number of threads per block. We see that as the threads per block increases, the computation to communication ratio decreases exponentially by a factor of 2 for a fixed image size. The communication time for the above sample remains fixed, since the total number of pixels transferred from CPU remains fixed. Therefore, the computation time is inversely proportional to the number of threads per block in both x and y directions. When a block has only one thread it alone runs on the GPU for the whole computation. This data explains the fundamental difference between a CPU and a GPU.

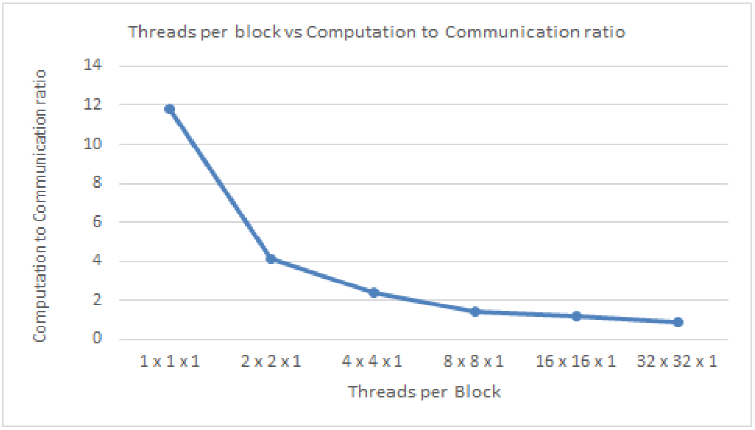


Figure 6 Threads per block vs granularity

Tegra X1 can support a maximum of 1024 threads per block. Since an image is a two dimensional entity, the maximum threads per block can be 32 x 32 x 1. So the finest granularity can be found at this configuration. One can also infer that lower granularity implies less computation time (since communication time is constant). This indicates better speedup of the algorithm.

It is obvious from the graph that 32 x 32 x 1 would be the best configuration for TX1 for our application. It should be noted that application specifications and hardware specifications dictate these configurations. For instance, 32 x 32 x 1 configuration would throw an error on Fermi GPU due to load handling capabilities.

4. Effect of image size on granularity

Computation to Communication ratio or granularity of a parallel algorithm gives some indication of the efficiency of the algorithm. In our implementation, we performed a study of the granularity of the application by varying the input size. As expected, the communication linearly increases with input size. We attempted to further reduce this linear rise in communication. However, doing so resulted in race conditions (discussed in section 5). We gave priority to obtaining race free computation, thus compromising on communication costs. However, the near-exponential rise in computation time makes up for these losses, resulting in a granularity curve that is exponential. The application can thus be classified as medium or coarse grained.

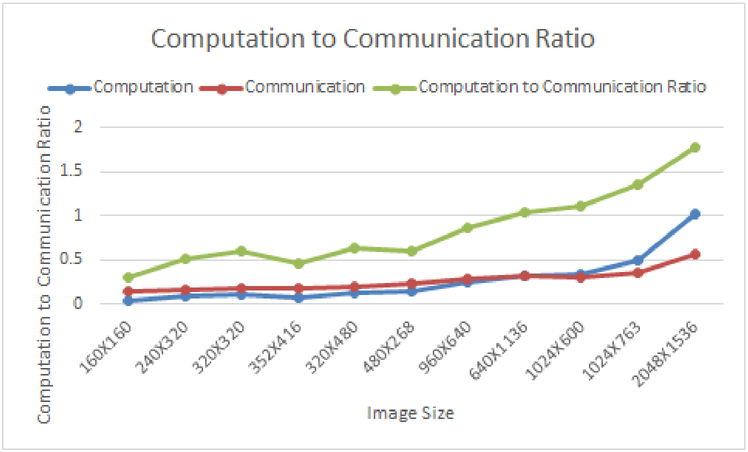


Figure 7 Image size vs Granularity

It can be observed that once a GPU kernel is called, the CPU is blocked and does not perform any computations. A solution to this could be to implement communication latency hiding. This is achieved by completing the mallocs required for the next kernel while the present kernel is running on the GPU. In the example below, mallocs can be done in place of the cpu_do_something()

```
kernel<<<grid, block >>>(par1, par2, par3….);
cpu_do_something();
cudaDeviceSynchronize();
```

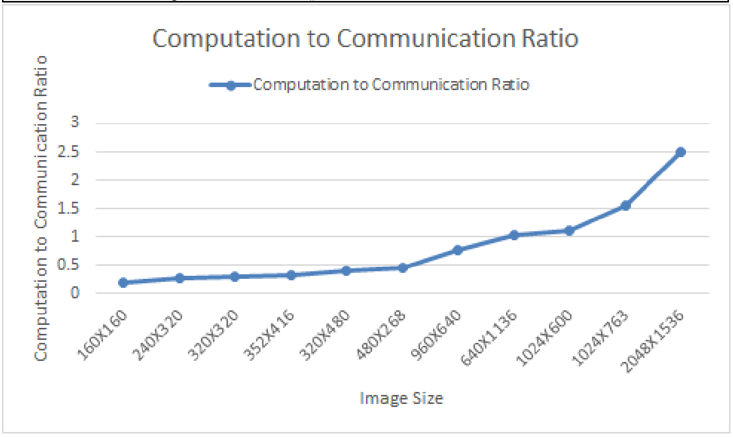


Figure 8 Improved Granularity with latency hiding

Figure 8 shows the effect of overlapping the computations of the CPU and GPU. It is observed that the curvature of the graph increases, thus providing better granularity. Latency hiding hides the communication costs involved in mallocs behind the computation of the current running kernel.
Another note that can be made from both the granularity graphs is that there is a sudden improvement in granularity from 960 x 240 size images. This could imply that these algorithms are better suited for tablets and mobile phones because their screen resolutions start from 960X640.

5.Comparing Tegra X1 and Fermi GPU

a.Speed Up:

The performance of Fermi GPU is higher than Tegra X1 for the obvious reason that Fermi has more computational capacity. The table below summarizes the fundamental differences between Fermi and Tegra.

	FERMI	**TEGRA X1**
Cores	512	256
Cache	Configurable L1 (48 or 16kB)	No L1. 2MB shared L2 cache with ARM CPU

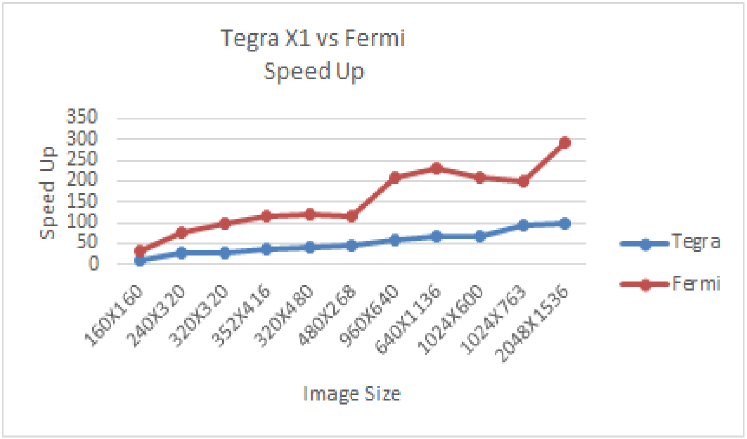


Figure 9 Tegra vs Fermi Speedup

It is seen that the computational capacity of Fermi with respect to number of CUDA cores is almost twice that of Tegra X1. However, the speedup shown in Figure 9 does not reflect the same. It is interesting to note that the average power consumed by Tegra X1 is 2.67W and by Fermi is 130W. That is, though the power consumed by Fermi is 48.7 times more than that of Tegra X1 , Tegra X1's speed is just 0.5 times less compared to Fermi. [6]

b. Granularity

Since there is no L1 cache in the Tegra X1, the costs involved in fetching and data and in write backs are more than that of Fermi. Also the shared L2 cache also deteriorates the performance by more than half. The significant effect of cache can be seen in computation to communication ratio.

It is observed that the Tegra X1 and Fermi follows the same pattern in granularity. For small size images until 320 x 320 images, the performance of Fermi is double of Tegra X1. However, for medium sized images (320 x 480 and 480 x 268) the computation to communication ratio of Fermi is not doubled. For large size images(from 960 x 640), the computation to communication ratio of Fermi is again double that of Tegra X1.

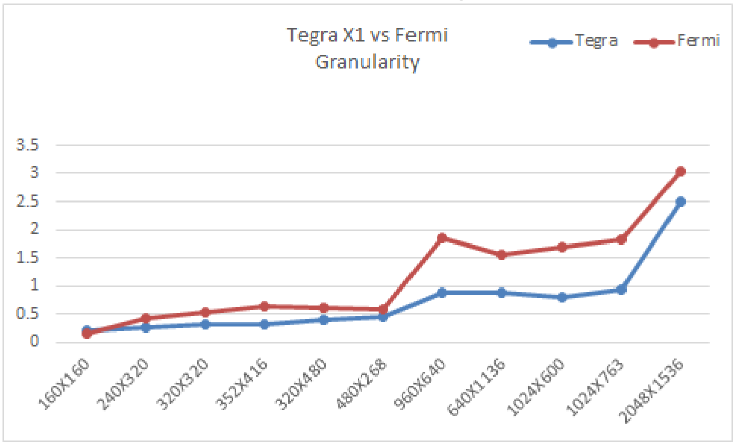


Figure 10 Tegra vs Fermi Granularity

6. Comparing CUDA C and GPU enabled OpenCV

Since GPU enabled OpenCV libraries are optimized at the lowest level, we are comparing our algorithms to get an insight of their efficiency. On observing the graph we see that OpenCV and our kernel follow the same pattern of speedup. OpenCV performs slightly better than our version upto a image size of 1024 X 763. The reason for this could be that OpenCV has pre-optimized libraries for parallelism, the exact details of which are unknown to us. However, for larger sizes of images (over 1024x763), our kernels begin to performs better.

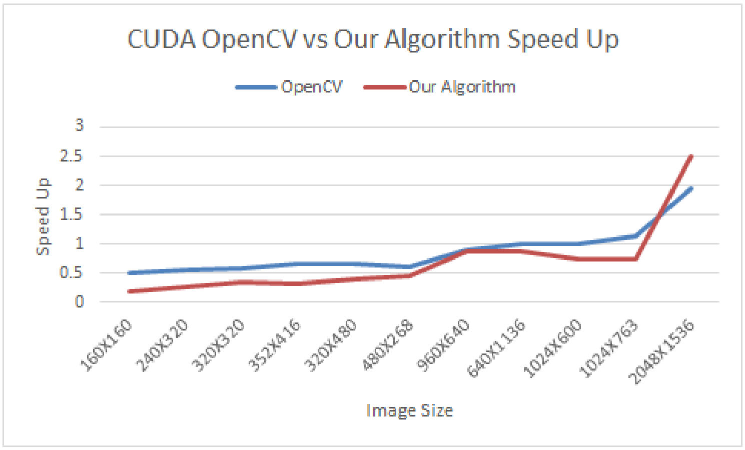


Figure 11 Comparison with GPU enabled OpenCV

IX CONCLUSIONS AND FUTURE SCOPE

We successfully parallelized lane detection (using Canny edge detection and Hough Transform) and obstacle detection (Morphological closing). Significant speedup is found in all the algorithms with Hough transform being the best because Hough Transform is computationally intensive while the other two are memory intensive. Even though the number of threads are increased much improvement can't be found in the latter two and this can't be avoided. Our application has a better performance in terms of speedup as the problem size increases. We observe that for image sizes that are multiples of 1024(maximum number of threads per block), there is a local peak in speedup. This trend is also observed in the comparison with GPU enabled OpenCV, where our design starts to perform better beyond a certain image size. Thus it can be concluded that our algorithm shows a good performance for tablet and mobile applications.

We observe that though the specifications of Tegra are lower, its performance is comparable with the Fermi. In spite of this, the power consumption is almost 50 times lesser. Therefore, it can be concluded that the Tegra is an extremely powerful SoC, with performance capabilities which are the highest in its league.

In order to save computation time, the Canny edge and Morphological closing kernels can be combined and executed simultaneously. However to do this, the Canny detection needs to be further enhanced or tradeoff must be made thus sacrificing. This project can be further enhanced to develop a mobile application, which captures real time videos from the camera while driving and provides swift feedback to the driver.

X. ACKNOWLEDGEMENT

We would like to thank Dr. Alan George for providing continuous support throughout the project. We would also like to thank Yathindra Kota for providing hardware support for the Tegra X1 and responding to our queries swiftly.

XI. REFERENCES

[1] www.embedded-vision.com/applications/automotive
[2] Phueakjeen, W., Jindapetch, N., Kuburat, L., Suvanvorn, N, *"A study of the edge detection for road lane*", Computer, IEEE Transactions, May 2011.
[3] Sharifi, M.,Fathy.M,Tayefeh Mahmoudi.M, "*A classified and comparative study of edge detection algorithms*", Information Technology, IEEE Transactions, April 2002.
[4] Chan Yee Low, Zamzuri. H, Mazlan.S.A, "*Simple robust road lane detection algorithm*", Intelligent and Advanced Systems, IEEE Transactions, June 2014. "
[5] Jungdong Jin,Dongkyun Kim, Ji Ho Song, Vinh Dinh Nguyen, Jae Wook Jeon, "*Hardware architecture design for vehicle detection using a stereo camera*", Control, Automation and Systems, IEEE Transactions, October 2011."
[6] Kiran Kasichayanula, Dan Terpstra, Piotr Luszczek, Stan Tomov, Shirley Moore, Gregory D. Peterson, "*Power Aware Computing on GPUs*", IEEE, July 2012."
[7] www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
[8] www.opencv.org
[9] www.caam.rice.edu/~timwar/RMMC/CUDA.html