Optimization of Distance Calculations for ADAS using Parallel Programming

Armaghan Asgar
Computer Science Department
Viterbi School of Engineering (USC)
Los Angeles, USA
aasghar@usc.edu

Onkar Anand Karle
Electrical Engineering Department
Viterbi School of Engineering (USC)
Los Angeles, USA
okarle@usc.edu

Abstract—**ADAS (Advanced Driver Assistance Systems) process computationally intensive data to generate feedback that improves driving experience and safety. In this project we perform an experiment for identifying objects using the You Only Look Once Model, a Deep Neural Network Model for multi object detection from an image and calculating distance to those objects using distance calculation formula. The report discusses details of our approach in implementing this project for object detection and distance calculation within each image frame of an input video. We demonstrate the comparison of our implementation when executed on CPU and GPU, and the performance gain achieved in this process.**

Keywords—Parallel Programming, Image Processing, Distance computation, Performance Optimization.

I. INTRODUCTION

Advanced Driver Assistance Systems (ADAS) are a group of automated technologies such as sensors and cameras that assist drivers in driving and parking functions. Through a safe human-machine interface, such technologies can detect nearby obstacles and driving errors and respond accordingly. ADAS safety features are designed to alert the driver to respond to driving problems and can even take control of the vehicle if necessary. ADAS was first incorporated in vehicles as an anti-lock braking system in the 1950s. Today, such systems utilize mechanical, electronic and software technologies of a vehicle to provide adaptive features to aid the driver in driving activities. Satellite navigation with traffic warnings, automated vehicle lighting, adaptive cruise control, lane departure, object and obstacle detection are common features found in many modern vehicles

The Society of Automotive Engineers categorizes ADAS into 5 levels based on the amount of system automation. Level 0 is the least amount of automation, and Level 5 is the most amount of automation. Examples of Level 0 are parking sensors, lane departure warnings etc. where the information is provided to the driver to self-interpret the information and respond. Level 1 and 2 are systems such as autonomous parking, emergency brake assist. Level 3 to 5, have the most amount of vehicle autonomy where level 5 is full autonomy which is yet to be proved in a non-lab environment [1].

ADAS systems can be passive or active/real-time. Active ADAS systems must process information more quickly and frequently especially for ADAS level 3-5 the demand for faster data processing to produce accurate result is of critical importance. Therefore, such adaptive systems, need to be able to process data using high-performance mechanical, electronic and software systems that work together to provide information to the vehicle autonomous control or driver for an appropriate response.

Advances in Computer Vision, Computational Photography algorithms and electronic sensors like LIDAR and RADAR technology produces data that needs to be computed on powerful computers which are low power consumers and have high computational intensity. Such systems can be used to implement real-time safety features like object detection, pedestrian detection, the movement and relative location of other vehicles and objects surrounding a vehicle. The motivation of the course project is also to look at a problem in this domain of calculating the distance of an object in a three-dimensional plane so that it can be processed quickly for appropriate response.

As the ADAS adaptive features are often safety critical, it is necessary to get the predictions accurate as quickly as possible. A promising way of improving compute speed if by optimizing algorithms, and by using parallel programming techniques and specialized hardware to accelerate the execution of the input data from multiple streams. ADAS systems require the images to be processed as they are captured so that the car can respond accordingly. The ADAS system must run as frequently as possible, and it needs to process new frames in real-time. With these reasons it is important to parallelize and accelerate real-time ADAS systems.

II. BACKGROUND

The motivation for this project is to calculate the distance of objects in an image which is being processed from a video stream frame by frame. This task involves two parts, Object Detection and Distance Calculation.

Once objects have been identified a serialized approach can be parallelized as the process of distance calculation is independent for every vehicle. Hence, multiple threads can be used to achieve parallelization where each thread can dedicatedly calculate the distance for each vehicle. In real time, vehicle/object density within an image is variable, and it is difficult to predict. When the vehicle or object density in the captured image is sparse then the parallel approach might underutilize the threads/cores being configured for the distance calculation. Therefore, thread/core optimization becomes challenging.

To achieve maximum performance, we should reduce the sharing of data between the threads/cores. So, the challenge is to write an independent code. Computational time can be reduced by increasing number of cores/threads. However, we need to utilize the invested hardware/software resources to honor the cost-performance ratio. Determining which programming model would be optimal considering the application needs is challenging.

For example, in case of ADAS Object-Vehicle distance calculation doesn't need to share any data among them. As we just need to compare the fixed threshold every time, we calculate the Object-Vehicle distance. So, message passing between threads/cores is not beneficial. We choose CUDA programming model.

For example, in the case of ADAS, Object-Vehicle distance calculation doesn't need to share any data among them. As we just need to compare the fixed threshold every time, we calculate the Object-Vehicle distance. So, message passing between threads/cores is not beneficial. To achieve our objective, we choose CUDA programming model.

The current solutions industry uses for object detection and distance estimation use technologies such as radar/LiDAR sensors that are responsible for calibration of targets using azimuth range and Doppler effects for object detection and classification tools to support Advanced Driver Assistance Systems. For the scope of this project, we are using a monocular camera without advanced technologies such as radar/LiDAR to capture a stream of images as a video, and use that video to identify objects, and find the distance to identified objects [2] and [9].

A. Object Detection: YOLO Model

Deep Neural Networks are a collection of nodes, called Neurons, which are represented in a hierarchical organization of layers where Neurons are connected to one another, such that output of one layer may server as input for another. Such a Deep Neural Network consists of input, output layers, and N number of hidden layers in-between input and output layers. The diagram below shows an example of a simple Neural Network.

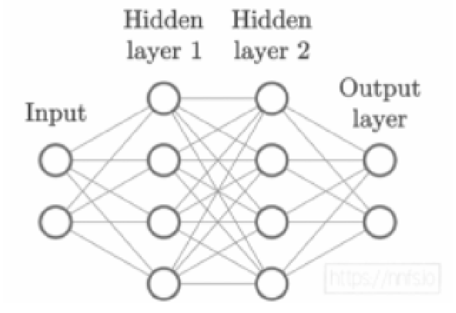


Figure 1: Basic Neural Network. Diagram Taken from book "Neural Networks from Scratch"

In a Deep Neural Network, layers of Neurons consist of input data, weight value associated with an input, a bias and activation functions. Calculations are performed on the combination of these inputs, which when pass a threshold result in a signal passing through to select Neurons in the next layer. For all the Neurons in a layer, a dot product Matrix Multiplication is performed for the sample input, respective weights, bias, in addition to the activation function which dictates a forward signal to the next layer of neurons. The below diagram shows the computation performed by each Neuron,

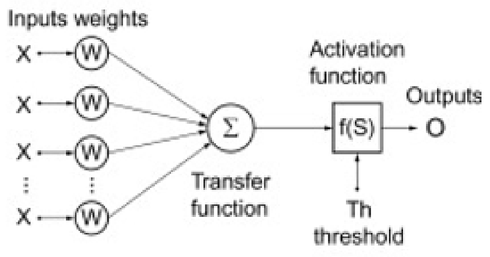


Figure 2: Input and Output View of Neuron, Diagram from Book "Intelligent Speech Signal Processing"

With a small number of Neurons in each layer, the GPU optimization may not result in significant performance improvement. However, in production scale Deep Neural Networks which have thousands of Neurons in each layer, and large number of dot product multiplications to compute, the deep neural network on GPU becomes essential for performance. According to the YOLO research paper [3], the GPU can provide at least 10x performance improvement over a CPU.

In this project, we explore and implement YOLO, You Only Look Once Algorithm, a Deep Neural Network for multi-object detection from an image. We explore the working of the theory behind the algorithm, and performance improvement received by using the algorithm with GPU. We also identify why the performance improved using GPU for our application of distance calculation. The YOLO model performs multi-object detection of 80 classes in an image in a single pass through the network. It uses regression and probability associated with each object in spatially separated bounding boxes that the algorithm forms out of an input image. As per the YOLO authors Using this strategy, the YOLO model reasons globally about the image when making predictions unlike other approaches for object detection such as classification techniques which are much slower in detecting, classifying objects in an image and require multiple passes through the network to identify multiple objects within an image versus YOLO which can identify multiple objects within a single pass.

The YOLO model performs multi-object detection of 80 classes in an image in a single pass through the network. It uses regression and probability associated with each object in spatially separated bounding boxes. As per the YOLO authors Using this strategy, the YOLO model reasons globally about the image when making predictions unlike other approaches for object detection, classification which are much slower in detecting, classifying objects in an image and require multiple passes through the network.

In the YOLO model, an input image is segmented into multiple grids where in each grid the algorithm forms bounding boxes. Each grid is responsible for identifying the objects of its bounding boxes. In the bounding boxes, objects are detected using class probabilities and confidence. If no object is found, then the confidence for the bounding boxes with the grid is zero. Instead of classifying objects, the YOLO model uses regression
to identify objects, which is also the cause of algorithmic speedup.

The diagram below, shows the working of the algorithm for the and the predictions are encoded as a tensor of size. The input image is segmented into SxS number of grids where each grid is assigned B number of bounding boxes. Each SxS grid is responsible for finding the target class for its assigned bounding boxes.

$$SxSx(B * 5 + C)$$

Where,

- *SxS is size of grid*
- *B is bounding boxes per grid*
- *C is class labels or targets*

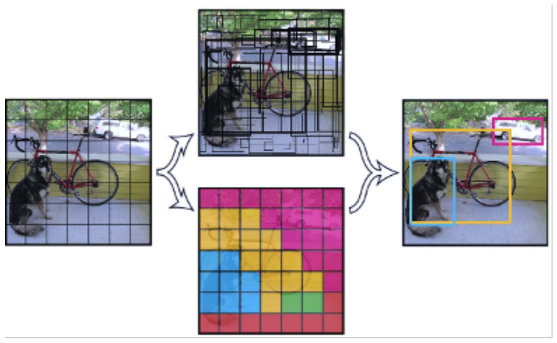


Figure 3: YOLO Model Image breakdown into SxS grid. Image Take from YOLO research paper

Within each SxS grid, the bounding boxes the algorithm identifies with a confidence score whether an object is present in the bounding box or not. If the object is present, that object is represented by its confidence score, class prediction and coordinates within the bounding box, and relative to the global image.

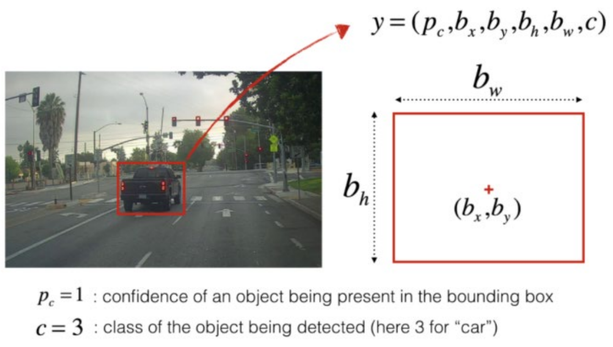

$y=(p_c,b_x,b_y,b_h,b_w,c)$
b_w
b_h
(b_x,b_y)
$p_c=1$: confidence of an object being present in the bounding box
$c=3$: class of the object being detected (here 3 for "car")

Figure 4: Bounding box of YOLO algorithm

B. Distance Calculation

After identifying the object using the YOLO deep neural network model, we received coordinates and reference of the objects with the image. We use distance calculation formulas, mentioned in detail in later section to calculate the distance to each object within an image.

III. Related Work

Distance calculation is the core part of all machine learning algorithms because the closer the query is to some data samples (i.e., observations, records, entries), the more likely the query belongs to the class of those samples.

One popular algorithmic approach in calculating distances is K-Nearest Neighbors (k-NNs) search which is a powerful distance-based tool for solving classification problem. It is required to train the local model-oriented classifiers. Faster distance computations can significantly boost the performances of such classifiers. GPUs are specially made for the purpose of speeding up the computation. Hence acceleration can be exploited with GPU based sorting algorithms for k-nearest neighbors. Performance of these sorting algorithms vary in accordance with the provided sequence of inputs. The GPUKNN is one of the novel approaches which proposes GPU based distance computation algorithm and automatically picks the suitable sorting algorithm according to characteristics of provided input sequence.

Most multicore GPU architectures tend to perform well in case of data level parallelism and task level parallelism processing is also combined into one integrated solution to improve the speed performance of multiclass related classifications and cross validations. This approach helps to reduce the time cost involved in each iteration of SMO (Sequential Minimal Optimization) during the training phase. In this approach, all these violators are shared among various tasks to reduce duplication of kernel computations in the multiclass classification and cross-validation [4].

With this approach, the obtained speed performance results have shown that the achieved speedup of both the training phase and predicting phase are ranging from one order of magnitude to three orders of magnitude times faster compared to the state-of-the-art LIBSVM software on some well-known benchmarking datasets.

Another existing approach is a whole concept of ADAS including autonomous driving capabilities that has been developed by NVIDIA. NVIDIA DRIVE® platform is a full-stack solution for highly automated, supervised driving. It includes active safety, automated driving, and parking. Mostly the Euclidean distance is used as the metrics [5] and [6].

IV. Hypothesis & Key Idea, Architecture, Algorithm

The goal of this project is to develop a framework to parallelize ADAS system for object detection and distance calculation in the image processing and computer vision domain, so that such real-time system can process data faster using parallel programming techniques.

The inputs of this project would be images or vector coordinates using which we would be able to identify the coordinates and calculate the distance in multi-dimensional space.

Achieving optimal results in terms of speed and accuracy depends both on the algorithm, parallel programming models and hardware platform. In this project we are exploring different parallel programming techniques, such as MPI, OpenMP and CUDA or a mix of these we would be able to implement a technique that is computational more robust than serial execution.

As ADAS is a safety critical system, it is important to have quick and accurate results. We are dealing with large number of datasets, input from monocular camera as image as part of a streaming video. Same operations are being performed on every image frame, i.e., object detection and distance calculation (Euclidean distance).

The hypothesis is based on exploiting the parallelization opportunity as the input datasets regions are independent. Also, there is possibility of reduction in the computational time using GPU architecture with SIMT model. The Euclidean Distance calculations can be accelerated by exploiting the available GPU resources (Threads per block).

Distance Calculations don't have any dependencies as those are the co-ordinates of vehicle being detected. So, task parallelism can be implemented with multiple threads contributing for CUDA kernel execution. Independent operations can be accelerated by increasing TPB (threads per blocks)

Key Idea

Object Detection and Distance Calculation are important features in most of the ADAS applications which we are experimenting in this project.

Below metrics are used in many machines learning and ADAS oriented algorithms for distance calculations on large datasets,

- Euclidean Distance

$$d(p,q) = \sqrt{\sum_{i=0}^{n}(q_i - p_i)^2}$$

Where 'p' can be assumed as the fixed co-ordinates of host vehicle from which the object distance will be referred 'q' can be treated as the detected vehicle co-ordinates.

- Manhattan Distance

Weighted Manhattan distance is another popular distance measurement like weighted Euclidean distance. It is also referred as weighted L1-norm distance, which is defined as

$$d_{ij} = \sum_{k=0}^{m} w_k |a_{ik} - b_{jk}|$$

- Hamming Distance

It is one of the metrics used in ADAS oriented algorithms where the distance is calculated on number of bit positions at which the two binary distance vectors differ.

The main metric on which the performance will be measured is execution time. We will have an execution time calculated for serial approach where next distance calculation is done only after the previous calculation is finished. This is the baseline version on which we will measure the performance.

After CUDA implementation, we expect a significant reduction in the execution time for same number of calculations.

Performance in terms of Execution time i.e.,

$$Speedup = \frac{\text{Execution time on CUDA implementation}}{\text{Execution time on baseline implementation}}$$

The 2D co-ordinate system is one to one correspondence between each vehicle at point P in 2D space. We are assuming host vehicle's position as a 2D co-ordinate as well. The distance between host vehicle and detected vehicle can be calculated by distance formula which is square root of the sum of the squares of the differences between corresponding coordinates. In the experimentation, the host vehicle's distance from ground and detected vehicle's distance from ground is assumed to be same. So, we assume the z is constant, we can ignore the z component in case of 3D co-ordinate systems.

If P1 (Host vehicle) with co-ordinates (x1, y1) and P2 (Detected vehicle) with co-ordinates (x2, y2) then the Euclidian distance formula,

$$d(P1, P2) = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

Above is the mathematical approach by which we will be calculating the distance between host vehicle (P1) and detected vehicle/object (P2).

Parallel Programming Models

During the analysis phase of possible parallel programming models that can be used for faster object detection and distance computation, we analyzed OpenMP, MPI and CUDA programming models.

In the past decade, out of many existing parallel programming tools and models proposed for various architecture of computer systems, Message Passing Interface (MPI) and OpenMP are two of the most widely used parallel models. Both MPI and OpenMP are designed for main stream computing systems. A model in which computing nodes do not share memory with each other is known as MPI model. Most common usage of this model is in a distributed environment such as a clustering system. The communication happens over messages.

All data sharing and exchange must be done through explicit message communications. A master node and a group of slave nodes are the core components of Message Passing Interface. The role of the master node is to scatter the data to the slave nodes and after the computation gather the results back finished on the slave nodes. The Master node is responsible for synchronization related activities. Also, intra-connection networks used in the application plays important role to speed-up of MPI based systems as the large amount of data might be transferred. Hence, in order to optimize performances of MPI models we should focus on minimizing data exchanges. The ADAS application that we are dealing with doesn't have shared data that needs to be communicated most of the time. But in application where computing nodes require sharing data between them then to compensate the absence of shared memory significant amount of work needs to be done on application design.

The next model we studied was OpenMP which supports shared memory. It is more commonly used in the multi-core personal computers and standard workstation systems. Shared memory system usually has a smaller scale compared to the distributed system. If the application needs to share a lot of data, then the scope of parallelization is limited. The application having less sharing and more independent operations where we can be exploit parallelism to its maximum abilities.

In terms of scalability, OpenMP is more restrictive model compared to MPI. Also, there is strict requirement on precise thread cycle management which won't allow OpenMP to spawn many threads as it involves thread creation overhead, context switching and synchronization of threads when needed. The resources are limited in case of OpenMP and MPI. If we exceed the number of threads than number of computing cores of CPU, we need to deal with additional overhead of time-division multiplexing to switch between the physical threads. It is less likely to have significant improvement in performance as we are bounded by limited number of computing resources and threads. Hence, the scalability is limited in OpenMP.

We can use OpenMP if the application that we are dealing with must be parallelized with minimal number of modifications in the original algorithms and we need to the boost on the performance of it on multi-core system. For example, algorithms running data independent tasks in a large loop structure can be easily accelerated with OpenMP.

GPU's programming model is kind of a mixture of both messages passing and shared memory with some of its own unique features. More recently, several major industry giants including Apple, Intel, AMD/ATI and NVIDIA have jointly developed a standardized programming model called Open Computing Language (OpenCL). OpenCL shares many common aspects from CUDA, but it is still not very mature which makes it less popular on NVIDIA GPUs. Therefore, CUDA is used for implementing the proposed solution to achieve the maximum speed gain by using the latest hardware from NVIDIA.

The CUDA device code can access/modify corresponding threads registers, also, can access/modify the shared memory corresponding to block, can access constant memory corresponding to grid. The host code can send/transfer the data to global and constant memory of the grid. Global memory access is costlier in terms of time. Constant memory access is faster for CUDA threads. In the CUDA data structure hierarchy, the top level is known as a grid which has group of thread blocks. There can be at most 65535 blocks in either of x or y dimensions or in total. Each thread block can contain at most 1024 or 512 threads in either x or y dimension. The maximum number of threads in all these dimensions must be less than or equal to 512 or 1024. It depends on which hardware and supported CUDA version. The thread organization is shown in below figure,

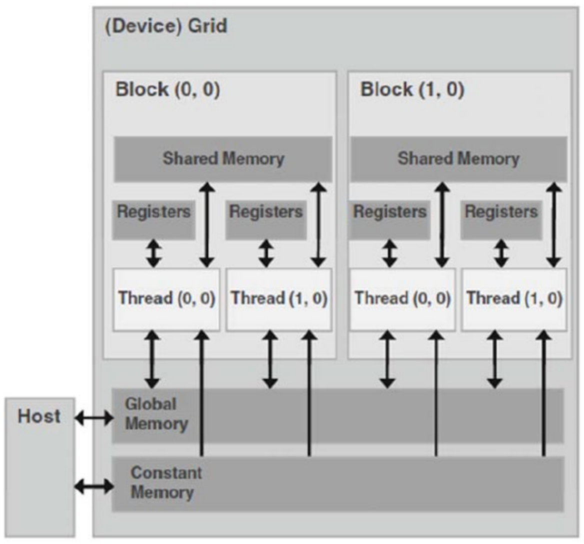


Figure 5: CUDA Device Memory Architecture

The CPU host launches the kernel function on the GPU device in the form of grid structure. After computation, the device can be used to execute another kernel function. In case of multiple devices availability, we can manage every kernel function through one CPU thread. It is simpler to execute a grid structure that has thousands of threads in it [8].

In OpenMP or MPI programming the thread creation is expensive. However, in CUDA the threads can be created with less overhead. The optimum number of threads and block configuration depends on the application. To have better performance, the thread usage should be in thousands or ten-thousands in each grid. GPU excels in performance because of computational resources availability. It would not be beneficial if we use a smaller number of threads. It is also important not to have excessive threads because it can affect the efficiency because of overheads.

In CUDA terminology, the processor can handle 32 parallel threads called warps. The processor is responsible for creation, management, scheduling and execution of warps. As we have warp size of 32 threads, it is a good choice to have threads per block as a multiple of 32. Threads inside the same block has limited shared memory that they use to communicate with each other.

Considering all the architectural advantages of CUDA with regards to heavy computation, we are choosing it for this application. We have identified the kernel (Logic to calculate distance between host vehicle and detected vehicle). This kernel is independent for every other distance calculation. Hence, it can be parallelized. The parallel portion of the application is executed N times in parallel by K different CUDA threads, as opposed to only one time like regular C/C++ functions. Here, we can exploit the parallelism to greater extent using CUDA approach.

Graphical Processing Units i.e., GPUs are microprocessors that are designed to perform parallel processing of tasks. Contrary to a CPU, a GPU is designed to process data in bulk using SIMD, or in case of GPU, SIMT. In Flynn's taxonomy, SIMT/SIMD means Single Instruction Multiple Data/Thread. In this approach there are multiple threads that are executing a single instruction on multiple data items, thus achieving speed by concurrency.

As ADAS is one of the safety critical features implemented in an Automotive industry [7], the correctness of the program is important. Also, the distance calculations ultimately going to lead to some decisions that decides the vehicle behaviors. The position of the vehicles is always moving in real-time, so it is always relative speed considering that the host vehicle is also moving. Cost to Performance ratio is important so while writing the software for such applications the developers should try to implement optimal strategies to utilize hardware/software to their fullest capabilities [10].

V. Evaluations (Experements)

A. Experimental Setup

Our implementation of the algorithm involves two parts. One part is the implementation of the YOLO Deep Neural Network, and the second part is the calculation of the distance to the objects that have been identified. To implement the YOLO Model, we have used OpenCV has the base framework, and used pre-trained weights and configuration from COCO dataset which can detect 80 classes.

To utilize the GPU support with, the OpenCV framework needs to be build using the correct flags from scratch. To do this we have used Google Colab Cloud environment, as the CARC setup had limitations that the OpenCV framework was not built with GPU support.

The Google Colab Colub Environment provided us with the following CPU and GPU compute compatibility using this we were able to test our implementation on both the GPU and CPU.

- GPU: NVIDIA Tesla K80, CUDA 11.2, cuDNN v8.
- CPU: Intel Xeon with single core (2.3 GHz).

The project code, consist of sample files, reference images and a video which is used as input to the program, the video, is taken frame by frame, where in each frame the following steps are performed.

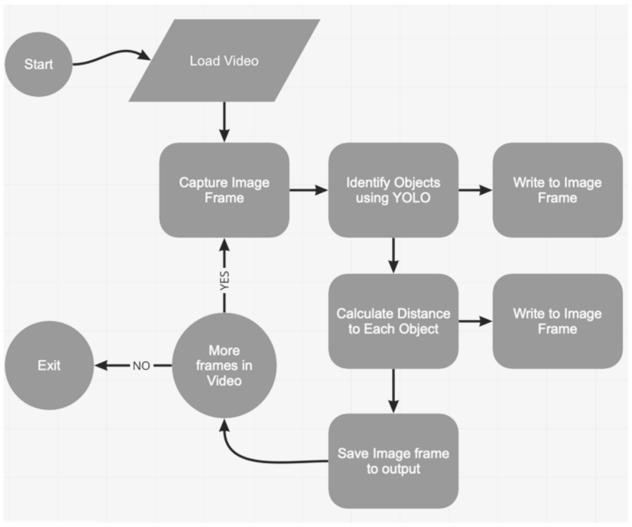


Figure 6: Flow of the code in algorithm.

The input to both parts of the experimentation can be described as follows:

Part A:

Distance calculations on samples by extracting co-ordinates from image using computer vision. In this approach, we are using computer vision-based to get the co-ordinates of the image which in real vehicle might be captured with the LiDAR and calculating distance on it. For the purpose of our experimentation, we would be using a monocular camera with an input video can be broken down into the following data set.

Datasets:

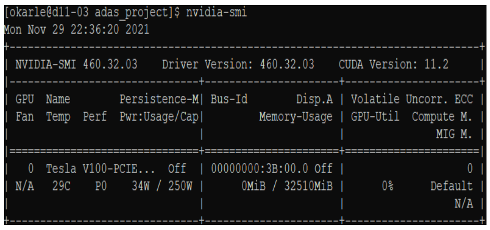
Sample size:

frames per seconds x number of seconds x number of objects per frame = 30 frames/seconds x 12 samples/frame x 75 seconds = 27000 samples

Part B:

Calculations on huge number of objects.

- CUDA, C Language, Time profiling tools
- GPU: NVIDIA Tesla V100, CUDA 11.2, cuDNN v8.
- Sample size: 1048576
- Threads per block used to execute kernel: 1,4,16,32,64,128,256,512,1024.
- Below image depicts the GPU specifications,



```
[okarle@d11-03 adas_project]$ nvidia-smi
Mon Nov 29 22:36:20 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-PCIE...  Off  | 00000000:3B:00.0 Off |                    0 |
| N/A   29C    P0    34W / 250W |      0MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Figure 7: GPU Configuration

B. Results and Analysis

The code flow described in the Experimental Setup is executed on the mentioned Setup using both exclusive CPU and CUDA.

Part A: The following graphs and diagrams reflect the execution of the algorithm on both CPU and GPU.

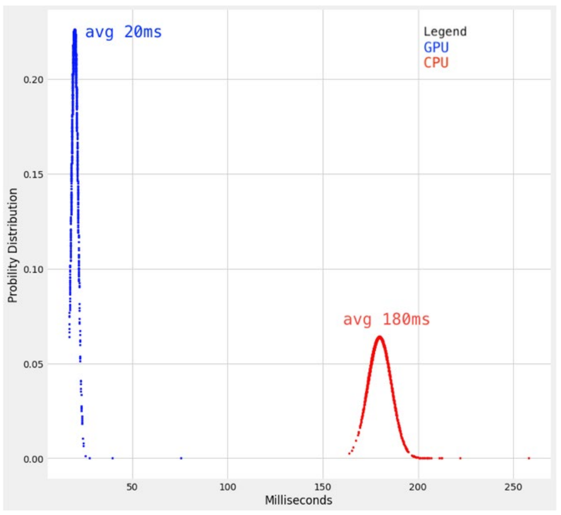


Figure 8: Bell Curve for GPU and CPU

The diagram below shows the standard deviation of object detection and distance calculation using GPU and CPU. The average time for the GPU object detection and distance calculation is 20ms and for CPU is 180ms. The main performance improvement is observed is in interface time of the Deep Neural Network.

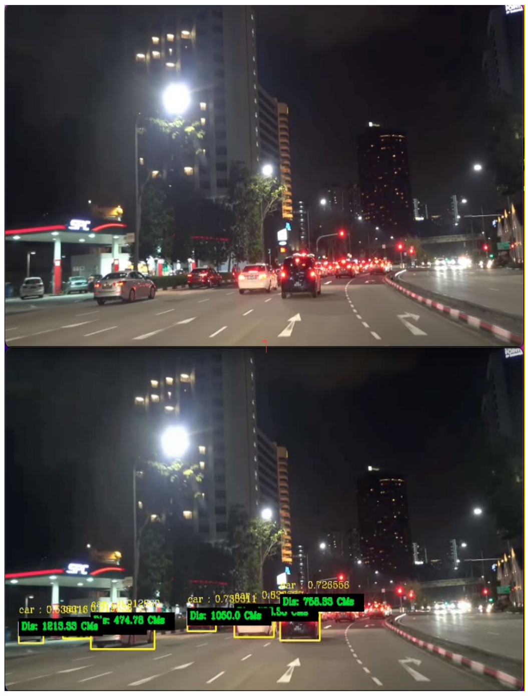


Figure 9: Object detection and Distance Calculations on real time data

The following diagram shows the holistic overview of the running time of complete execution of input data which is video of 30 seconds at 30 fps on both CPU and GPU,

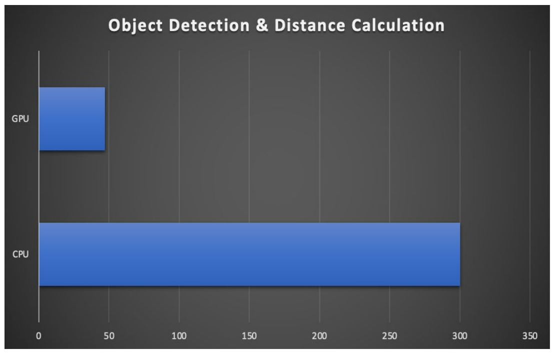


Figure 10: Holistic Computation of CPU vs GPU

Yolo Algorithm performance on CPU vs GPU:

- 30 seconds input video at 30 frames per seconds is used as input for both CPU and GPU
- Speedup = 300/47seconds = ~ 6x Speedup

It's faster in GPU because,

- Thread parallelism
- Higher Memory Bandwidth
- Optimal for matrix multiplication for inference of the Deep Neural Network

Part B: The following graph show the performance improvement when we increase the threads per block in CUDA setup.

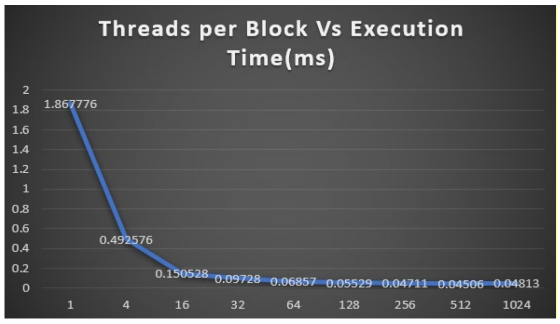


Figure 11: Distance Computation performance analysis on GPU

- Very small block sizes (i.e., 32 threads per block) may limit performance due to occupancy. Very large block sizes for example 1024 threads per block, may also limit performance, if there are resource limits (e.g., registers per thread usage, or shared memory usage) which prevent 2 thread blocks (in this example of 1024 threads per block) from being resident on a SM.

- Thread block size choices in the range of 128 - 512 are less likely to run into the previously mentioned issues. Usually there are not huge differences in performance for a code between, say, a choice of 128 threads per block and a choice of 256 threads per block.

- Due to warp granularity, it's always recommended to choose a size that is a multiple of 32.

Assuming the threads per block used in baseline machine is 1, then we can observe below performance improvement.

$$\text{Speedup} = \frac{Execution\ Time\ at\ TPB(1)}{Execution\ Time\ at\ TPB(512)} = 41.4$$

VI. Conclusion

In case of object detection and distance calculation on CPU vs GPU, we observe the speedup of 6x.

If we compare the distance calculation performance on single thread vs 256/512 threads per block, we get performance improvement of 41.4x.

Learnings, challenges, and takeaways:

Identification of kernel for optimization, Amdahl's law, Usage of Real GPUs for optimizations, Debugging and understanding reasons for performance improvement on GPUs.

Acknowledgment

This project is presented as a part of Parallel Programming coursework. We would like to thank the Prof. Viktor Prasanna and Yuan Meng for the mentorship. We are thankful to Center for Advanced Research Computing team at University of Southern California providing lab support.

References

[1] Galvani, Marco. "History and future of driver assistance." IEEE Instrumentation & Measurement Magazine 22 (2019): 11-16.

[2] Optimal Performance Prediction of ADAS Algorithms on Embedded Parallel Architectures by Romain Saussard, Boubker Bouzid. Conference: 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS) and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

[3] Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[4] Fast Parallel Machine Learning Algorithms for Large Datasets Using Graphic Processing Unit, Qi Li, Virginia Commonwealth University.

[5] NVIDIA, "Nvidia kepler GK110 architecture whitepaper," 2012.

[6] NVIDIA, "Cuda C programming guide," 2014.

[7] Deep Neuro-Vision Embedded Architecture for Safety Assessment in Perceptive Advanced Driver Assistance Systems: The Pedestrian Tracking System Use-Case by Francesco Rundo, Sabrina Conoci, Concetto Spampinato, Roberto Leotta, Francesca Trenta and Sebastiano Battiato.

[8] Programming methodologies for ADAS applications in parallel heterogeneous architectures by Djamila Dekkiche. Computer Vision and Pattern Recognition [cs.CV]. Université Paris Saclay (COmUE),2017.

[9] Hardware accelerator IP cores for real time Radar and camera-based ADAS by Sergio Saponara Journal of Real-Time Image Processing volume 16, pages 1493–1510 (2019)

[10] Predicting ADAS algorithms performances on K1 architecture Romain SAUSSARD1,2 – Boubker BOUZID1 – Roger REYNAUD2 – Marius VASILIU2 1Renault, 2Université Paris Sud