Characterization of Real-Time Object Detection Workloads on Vehicular Edge

Sihai Tang*, Kaitlynn Whitney*, Benjamin Wang†, Song Fu*, and Qing Yang*
*Department of Computer Science and Engineering, University of North Texas
Email: {SihaiTang, kaitlynnwhitney}@my.unt.edu; {Song.Fu, Qing.Yang}@unt.edu
†Department of Computer Science, University of Texas at Austin, Email: benywang@utexas.edu

***Abstract*—As recent literature suggests the need for communication between autonomous vehicles, edge devices have emerged as a viable conduit to facilitate real-time data sharing. Edge devices strike a suitable medium between the alternatives of cloud centralization and full vehicle-to-vehicle decentralization, providing the computational savings of sending and receiving information from one place while also boosting speed by bypassing internet protocols. Given the novelty of both object detection models and autonomous vehicle-oriented edge device implementation, there are no standards for hardware and software specifications on the edge. In this project, we seek to address this void, investigating the GPU and CPU usage patterns of various object detection models and machine learning frameworks. We also aim to uncover optimization opportunities such as workload pipelining. One early difficulty was that only a few models tested achieved real-time (<33 ms) object detection. Our results show that the GPU utilization varies widely between models. One interesting is that only one CPU core is used during the inference process, suggesting the number of CPU cores will not be a bottleneck. Meanwhile, we find that increasing CPU cores proportional to the amount of traffic will likely be necessary to preserve real-time object detection.**

***Index Terms*—Autonomous Vehicle; Edge Computing; Real-Time Object Detection; Faster R-CNN; PyTorch; Workload Characterization; SSD; Darknet; YOLO.**

I. Introduction

Real-time object detection has been a hurdle for autonomous vehicles for some time. One of the primary challenges is the constantly changing roadside environment. Vehicles informed by an old environment can easily get into accidents. In the span of a single second, vehicles must produce real-time detection results in order to make accurate maneuvers. Thus, in order to ensure the safety of humans in a world of autonomous vehicles, both speed and accuracy need to be improved for real-time detection.

Real-time object detection is defined by Redmon, et al as achieving 30 fps or better [33]. In this study, as our primary focus was on image detection, we extrapolated a benchmark of 33 milliseconds for inference to be completed. Humans interact and process the information from the environment to make decisions and perform actions, this is how we function. Safe driving takes what we can do a step further by translating our decisions into actions performed by the car. In autonomous driving, the function that a human would perform is instead performed by the vehicle's on board processing unit (OBU). In the ideal world, a continuous and reliable method would be used to process the data and make safe driving decisions. However, in the real world, nothing is perfect. From sensors to the end decision, there exist multiple steps, with various latencies.

To facilitate the self-driving process, various sensors need to relay their sensing data of the surrounding environment to the on-board computing unit. This is usually handled by an array of sensors such as the LiDAR, cameras, radar, GPS, IMU, and more. Due to the vast array of sensors, it is estimated that an autonomous vehicle will generate 4 terabytes of data or more in two hours[39]. To enhance driving, connected and autonomous vehicle (CAV) technology enables raw-data level and feature-map level data sharing among vehicles [27, 26], which utilizes extraneous data from other vehicles to drastically improve the detection capabilities of a single vehicle.

The three frameworks examined during this research are ChainerCV, Darknet, and PyTorch. The three models paired with the frameworks for this research are Faster Region-based Convolutional Neural Network (Faster R-CNN), Single Shot Detection (SSD), and versions 1, 3, and 4 of You Only Look Once (YOLO). These models are popular for Region Proposal Netwoks (RPN) which takes any sized image as input and produces an output of the image with rectangular object proposals. In this research we are concerned with RPN models that utilize the Graphics Processing Unit (GPU).

Using the program NVIDIA Nsight™ Systems (NSys), a system-wide performance analysis tool, we are able to visualize each algorithm framework-model combination. We then will compare the results against the other framework-model combinations.

With the use of a consumer grade Graphics Processing Unit (GPU) as the external accelerator, we are interested in examining the variance of multiple frameworks and models processing real-time object detection. This will help us to gain a better understanding for the need of resource analysis and optimization for cooperative perception on edge for consumer grade GPUs. The goal is to obtain information on a framework-model combination such that can preform at a quicker speed without too drastic of a change to accuracy. If there is a way to increase speed without sacrificing accuracy on a more affordable consumer grade setup, the safety of autonomous vehicles on the road could become more readily accessible to both researchers and industries.

Our key observations are that over encompassing frameworks tends to run the same detection algorithm slower than

the specific detector for the same detection algorithm. We also find that the most commonly used framework, PyTorch, relies heavily on CUDA kernels and not as much on memory. We also find that the fastest inferencing method, Darknet based YOLOv3, relies more heavily on memory than on CUDA kernels.

We organize the our paper as follows. We will discuss the related works in Section II and our motivation in Section III. We then introduce the frameworks and models characterized in Sections IV and V respectively. The key findings are presented in Section VII, and we discuss the potential impacts of said findings in Section VIII. Finally, we conclude our paper in Section IX.

II. Related Works

Setting the foundation for most of autonomous vehicles is the combined rise of both hardware and software advances. Starting from the early 2000s, techniques such as Gradient based neural networks begin to see popularity[1]. But it really was not until powerful GPUs and TPUs came into the picture before the autonomous vehicles became a real possibility [19].

A. Autonomous Vehicles

It is undeniable that autonomous vehicles are not ready for mass public adoption yet, but the future of having autonomous vehicles is also being paved by most auto makers. Besides the work being done to incorporate autonomous driving, we also see other challenging issues rise up. In works such as [36, 5], we see numerous opportunities for infrastructure, communication, object detection, fusion and more all play an important role towards the success of a autonomous future. However, it is also evident that all types of sensors, ranging from LiDAR to radar have been used to drive the data collection for autonomy over the years[32]. That leads to the main safety issue of actual detection for autonomous vehicles.

B. Object Detection

From open datasets such as [7, 3, 20], we see many motivated research institutions prioritize the rapid research and development of autonomous vehicles; particularly the facet of autonomous vehicles that deal with local environment perception, lane detection, traffic sign detection and detection for objects like cars, cyclists and pedestrians [9, 15, 25, 18].

In works such as [26, 27], the authors make clear of the fact that sensors from individual vehicles are inherently limited in their sensing range. Adding on top of this fact, we also see vast room for advancement even when using cutting edge Convolutional Neural Networks [4].

C. Edge Computing

When it comes to the main usage of on board computation for autonomous driving, the main components that matters are perception and path planning. Works such as [22] introduce the idea of fusing data from multiple sources for object detection and object tracking, but the idea of operating such tasks On-Edge has only been explored by few authors. In the inspiring work [17], the authors developed a shared real-time situational awareness system by aggregating crowd sourcing and edge computing together. Also, in [28], the authors explored collaborative learning On-Edge computing, however, the challenges that edge computing faces in specific applications for object detection are not explored in this paper.

Aside from the technical challenges faced, we must also consider the hardware challenges that exist for the integration of edge computing. For example, the typical load of a single autonomous vehicle is the combined data from all the sensors. This load will only increase with more sophisticated sensors and hardware. While innovative solutions for using feature maps or feature pyramid networks reduce the load on the hardware, this is not a long term solution [16, 12]. Advances in more capable real time detectors, such as the Single Shot Detector (SSD)[8], still rely on the underlying hardware.

III. Motivation

One of the primary challenges in object detection for autonomous vehicles is the constantly changing roadside environment. Latency in this process can lead to catastrophic accidents. Thus, in the span of a single second, vehicles must produce many real-time detection results in order to make accurate maneuvers. Real-time object detection is defined by Redmon, et al as achieving 30 fps or better [9]. In this study, as our primary focus was on image detection, we extrapolated a benchmark of 33 milliseconds for inference to qualify as real-time. To further bolster vehicles' perception, we envision connected autonomous vehicles combating object occlusion and expanding sensing range. Regarding connectedness, edge devices strike a happy medium between the alternatives of cloud centralization and full vehicle-to-vehicle decentralization, providing the computational savings of sending and receiving information from one place while also boosting speed by bypassing internet protocols.

As edge devices can facilitate both real-time detection and data sharing, they are the focus of our study. Our target system is F-Cooper, a system that uses PyTorch as its main machine learning framework. However, due to this, traditional static profiling methods will not work due to the dynamic graph nature of PyTorch, it will generate different network behavior at runtime for model optimization [29]. With static offline testing, it is not possible to glean all the available details. So we turn to NVIDIA's Nsight System (NSys), a sampling based profiler. With a sampling of 1 million events per sample at 1000Hz, we are able to directly gather information during run-time.

A. End to End profiling for Edge

To fully understand the workflow of something like the F-cooper framework, we must keep track of resources such as data, scheduling, and hardware interactions.

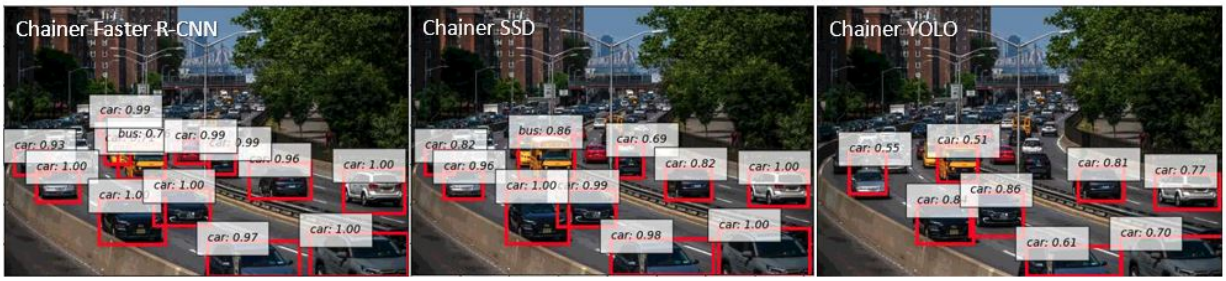

Fig. 1. ChainerCV Output Images

1) Data: In modern day autonomous systems, multiple sensor types are utilized, with the most commonly seen being the combination of LiDAR and Camera for the main driving input. Take tesla for example, to help reduce latency, they utilize embedded soc on the sensors to help pre-process the information. More on single vehicle here On the other hand, other methods include V2X infrastructures and frameworks. Take F-Cooper for example, the data is extracted from the vehicles before being sent to the edge. In many cases, there will also be the overhead of network latency on top of everything else.

2) Scheduling: Scheduling is often a matter handled by the framework, such as PyTorch, or the system, such as native kernel functions. Normally this approach is fine as modern systems are quite well equipped to handle loads. However, when we add in the factors for real-time driving and real-world scenarios with passenger lives on the line, we must take utmost caution.

This variable is embedded in every step of the way, from deciding which vehicle to service all the way to how the edge unit needs to process the data. In our profiling, we must take care to address this variable.

3) Hardware: As for the hardware component, the necessity for profiling is even more prudent than the previous two factors added together. From the perspective of manufacturers and customers, this issue is a simple matter of how much they are willing to spend. However, even with the best hardware, we will find that even the best hardware will be limited by a plethora of bottlenecks. Take yolov4 for example, while it is capable of real time detection on the developer's platform with top of the line hardware. On our hardware, TU106 Silicon with a memory bandwidth of 327.89 GiB/s, we see an average of 30 to 40 frames per second(fps) inference on video data, but when we perform the same inference on a live video feed, the performance drops down to 6 fps, a loss of performance by 500 %.

Specifically for the purpose of this research, we relied on a consumer grade NVIDIA GeForce RTX 2070 (mobile) as the GPU for the external accelerator of all the codes tested. The driver version associated with the GPU is 465.19.01 with CUDA version 11.3. The architecture on the machine we used is Intel x86_64. The Central Processing Unit (CPU) was an Intel(R) Core^{TM} i7-1050H CPU @ 2.6GHz which contains 6 cores with 2 threads each, for a total of 12 threads.

IV. Frameworks

A framework provides fundamental low-level functionality to a programmer so that they can focus on the high-level functionality aspects of their application. Each of the frameworks use different architecture to calculate the real-time object detection of an image. The frameworks selected for this research all utilized the Graphics Processing Unit (GPU) for computation power.

In this section, a brief description of the architecture and functions for each framework is explained. First, a description of the ChainerCV framework, followed by Darknet, and finally PyTorch. A brief mention of the models used for each framework will also be provided in this section. Though a more thorough look at the models will be discussed in the *Models* section. The datasets that the frameworks used in this research are the PASCAL Visual Object Classes (VOC) [2] and Common Objects in Context (COCO) [11]. Each dataset used for each framework-model combination will also be mentioned.

A. ChainerCV

ChainerCV is an add-on package to the Chainer framework, which is intended to provide non-experts fast prototyping by utilizing pre-trained weights, state-of-the-art models, and training scripts [30].

The deep neural network requires a significant amount of power in order to perform floating point numeric calculations. Therefore, an external accelerator, such as a GPU, is necessary in order to fully leverage the computation power. Chainer specifically relies on the open-source Python library, CuPy which provides the computational power of specifically NVIDIA GPUs and was initially developed as the back-end of Chainer. Only in June 2017 did CuPy become independent of Chainer. CuPy fully utilizes the GPU architecture with the CUDA platform that is provided by NVIDIA. Since CuPy has a NumPy-like syntax, it is highly compatible with NumPy and can often be interchangeable with a NumPy import. CuPy implements many functions that supports linear algebra, sparse

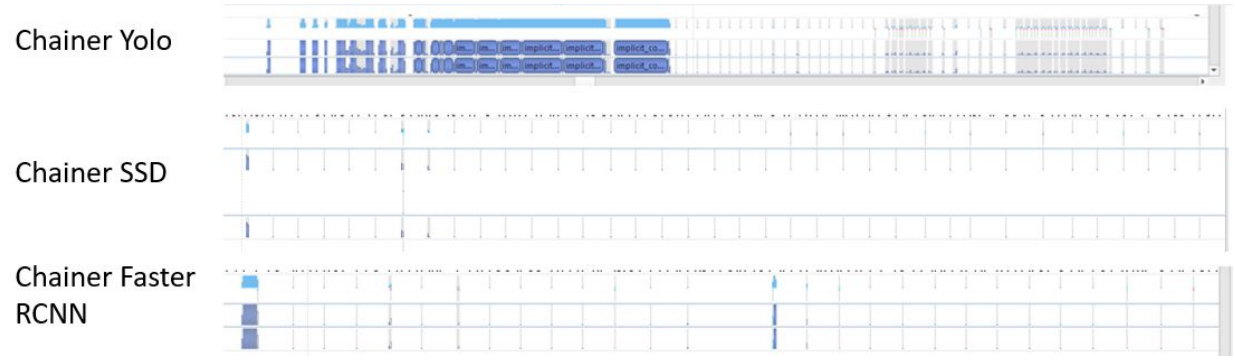

Fig. 2. ChainerCV GPU Nsys Results

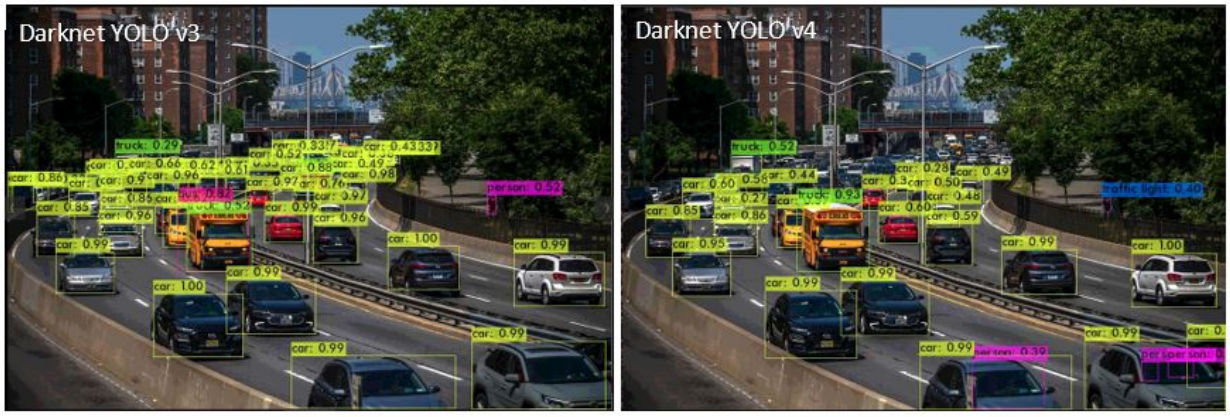

Fig. 3. Darknet Output Images

matrices, and sorting which all assist with the computations needed for a deep learning framework. [30]

For this research, three models were examined utilizing the ChainerCV framework; Faster R-CNN, SSD, and YOLO version 1. ChainerCV is a single download that gives access to each of the models mentioned and as such all models utilize the same datasets. The datasets used for this framework are the VOC 2007 and 2012 [2].

B. Darknet

One of the goals of the Darknet framework was to allow conventional consumer grade GPUs for training and detecting objects. To accomplish this task, Darknet uses a small number of groups (1-8) in convolutions layers and relies on the use of grouped-convolution while refraining from the use of Squeeze-and-excitement (SE) blocks [31]. According to [31], the hope in doing so is to find the optimal balance among the input network resolution, the convolutional layer number, the parameter number, and the number of layer outputs. While Darknet is not the most proficient in object classification, it claims to excel at the detection of objects utilizing the COCO dataset [31].

For this research, only one model was examined utilizing the Darknet framework, YOLO. There is, however, two versions of YOLO examined, version 3 and version 4. Darknet is a single download that gives access to both version 3 and 4 of YOLO as as such uses the same dataset. The dataset used for this framework is COCO 2017 [11].

C. PyTorch

Most deep learning frameworks will favor usability over speed or vice versa. PyTorch uses hardware accelerators, like GPUs, to favor both usability and speed. According to [29], some of the design goals for PyTorch include: keeping interfaces simple and consistent with the already popular Python language; prioritizing research by ensuring that writing models, data loaders, and optimization is as easy and productive as possible; provide tools that will allow researchers to manually control their code's execution in order to find their own performance improvements outside of what the library already provides; and keeping the internal implementation simple in order to easily adapt to new situations and keep up with the constant progression of the AI field.

For this research, three models were examined utilizing the PyTorch framework; Faster R-CNN, SSD, and YOLO version 3. Each model is part of its own individual code and do not use the same dataset. The Faster R-CNN and YOLO models utilize the COCO 2017 [11] dataset while the SSD model utilizes the VOC 2007 and 2012 [2] datasets.

V. Models

There are three models utilized during this research: Faster Region-based Convolutional Neural Networks (Faster R-CNN), Single Shot MultiBox Detector (SSD), and You Only Look Once (YOLO). During this section, the architectures for each model will be explained.

A. Faster R-CNN

Region-based Convolutional Neural Networks (R-CNNs) have for many years been the influence in the advancement of object detection. At first the R-CNNs were computationally expensive in their original development which prompted the advancement of fast R-CNN, which used GPUs for faster computation power. Both R-CNN and fast R-CNN rely on a Selective Search architecture, which greedily merges superpixels based on engineered low-level features [6]. However, while fast R-CNN was an improvement from R-CNN, the Selective Search architecture became the bottleneck of the algorithm. Therefore, when faster R-CNN was developed, it introduced the use of a different type of architecture, Region Proposal Network (RPN). RPN generates potential bounding boxes in an image, classifies the proposed boxes, and then refines the bounding boxes by eliminating any duplicate detection, and finally re-scores the box based on the other objects in the scene [9]. The frameworks in this research that use the Faster R-CNN model are ChainerCV and PyTorch.

B. SSD

Often times for object detection, when speed is preferred then the accuracy of object detection suffers. The Single Shot MultiBox Detector (SSD) model maintains an accuracy of object detection and increases the speed by avoiding the re-sampling of pixels [8]. Some of the changes SSD has implemented that differs from other models, such as Faster R-CNN and YOLO, includes offsetting bounding box locations by using a small convolutional filter to predict object categories, detecting different aspect ratios with the use of separate predictors (filters), and then applying these filters to various feature maps from the later stages of a network in order to perform object detection at multiple scales [8]. The frameworks in this research that use the SSD model are ChainerCV and PyTorch.

C. YOLO

The You Only Look Once (YOLO) model reframes object detection as a single regression problem to predict what objects are in an image and where they are located within the image by only looking at the image a single time [9]. The simplicity of YOLO is one of the reasons it is considered on of the fastest models available; the image is resized, run through a convolutional network that simultaneously predicts multiple bounding boxes with potential classifications, and then output goes through a non-max suppression to filter the object detection predictions [9]. The frameworks in this research that use the YOLO model are Chainer, Darknet, and PyTorch.

VI. Characterization Results

For the evaluation of the different frameworks in this paper, first we will examine the differences in the output images that each model produced. Then we will examine the GPU Kernels and briefly discuss the time that the code spent on the GPU. A more comparative look of the time on the GPU will be discussed in the *Results* section. Each Framework-model combination utilize the same image [34] for a clearer comparison of the output images.

A. ChainerCV

The ChainerCV framework that was utilized in this research included the three models, Faster R-CNN, SSD, and YOLO, in a single code. Each have a similar structure to each other with varying speeds and detection accuracy that each produced. Figure 1 displays the output images of each model using the ChainerCV framework.

From the output images, we notice that the Faster R-CNN (left) and SSD (center) had a slightly more accurate detection rate on the bounding boxes compared to the YOLO (right) model. When paired with the ChainerCV framework, the YOLO model did not produce as many detected vehicles as Faster R-CNN or SSD models. The decrease in accuracy and detection rate could be the result of attempting to write a single code for multiple models as opposed to optimizing a code to a specific model.

The algorithmic pattern mapped out by the NSys program, figure 2 displays a scattered pattern. The reason for this is that the code is constantly switching between processing the data on the CPU and GPU. Where the GPU has gaps, the CPU takes over processing.

This switching method is not the most efficient since the CPU has an overall longer duration for processing than the GPU does. This switching prolongs the code into making ChainerCV the longest running framework of those tested in this research.

B. Darknet

The single Darknet code utilized during this experiment used both version 3 and 4 of the YOLO model. Therefore, this comparison can only be observed between the two different versions of the same model. Figure 3 displays the output images of both versions.

The amount of vehicles detected for the Darknet framework far exceeded the amount of cars that were detected for the ChainerCV framework. There are slight differences between the YOLO version 3 (left) and version 4 (right) codes. The most noticeable being that version 4 detects the humans driving the vehicles in the front of the image while version 3 does not. Version 3, however, detects more vehicles further along the street in the image than version 4 does.

The algorithmic pattern mapped out by the NSys program, figure 4, displays a constant stream of GPU commands. While

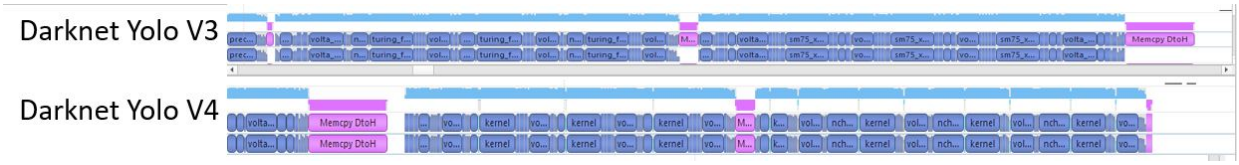

Fig. 4. Darknet GPU Nsys Results

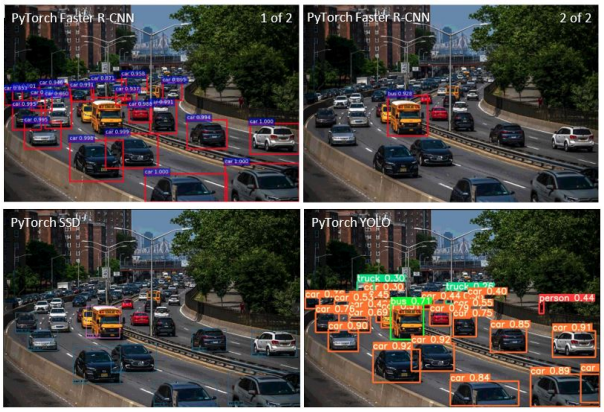

Fig. 5. PyTorch Output Images

Darknet does utilize the CPU before and after the GPU, the GPU computation is relatively consistent.

The Darknet framework has consistently the fastest duration for object detection of the frameworks selected for the research. The claim of being optimized for consumer grade equipment [31] appears to be accurate.

C. PyTorch

For PyTorch, there are three different codes that were used for the three different models, Faster R-CNN, SSD, and YOLO version 3. As such, the resulting output images differ not only in speed and accuracy but also how the bounding boxes are displayed. Figure 5 displays the output images of all three individual codes.

For the Faster R-CNN (top), two output images were created. The image on the top left displays the bounding boxes around all cars that the code was able to detect. The top right image displays the bounding boxes around all buses that the code was able to detect. The resulting output image of the SSD model (bottom left) has the least amount of vehicles detected of all the different code variances. The bounding boxes are also less noticeable and more difficult to pick out. The YOLO version 3 model (bottom right) had the most objects detected and even included the variance of truck and persons on the street detection. However, the accuracy of the bounding boxes were less than the Faster R-CNN model.

The algorithmic pattern mapped out by the Nsys program, figure 6, displays a continuous stream of GPU commands for Faster R-CNN and SSD models. The YOLO model displays a more sparse pattern of GPU commands.

One of the interesting things about the way the PyTorch framework is structured, all of the memory the GPU needs is allocated prior to any detection calculation. By taking the time in the beginning to allocate the memory usage, the detection computation time is significantly reduced.

VII. Key Findings

Our profiling results demonstrate the general trend that frameworks that support multiple models are slower than those that are designed around a specific model. Specifically, of the three models that achieved real-time inference, two were Darknet YOLO implementations. In contrast, ChainerCV SSD and Faster-RCNN were extremely slow, running in around five seconds, catastrophic in a live traffic environment. Another YOLO implementation on ChainerCV nearly met our benchmark, completing inference in 44 ms, ensuring that YOLO was overwhelmingly the fastest model on average.

GPU utilization was also a primary research interest targeted in profiling. We find that we can loosely stratify utilization by framework: PyTorch most heavily relies on CUDA kernels, ChainerCV relatively evenly splits workload between kernels and memory, and Darknet is highly dependent on memory. Given these patterns, we conjecture that Darknet's high inference speed can be largely attributed to its high memory utilization percentage.

The quickest framework-model combination tested was the Darknet YOLO version 3 code. The accuracy and number of objects detected was also stellar in comparison to the others, visibly detecting many more cars at farther distances. Overall, for the hardware setup utilized for this research, the Darknet YOLO version 3 code had an increase of speed without much sacrifice to accuracy. Darknet YOLO version 4, while also maintaining a good accuracy, took almost double the processing time that version 3 did. Contrasting YOLOv3, while PyTorch-SSD does have the second fastest processing time, the number of objects that were detected was the lowest among all of the framework-model combinations.

VIII. Discussion

Each framework-model combination showed varying results for the kernel outlay. Through examination of the kernels, we are able to see that the more efficient codes were those that relied more on the GPU as opposed to the CPU. Though, it is possible that due to the ChainerCV being a single code written for all three models, that the code might not be optimized for each model. Therefore, a way to further this research is to also test frameworks that are optimized for a specific model, in the

Fig. 6. PyTorch GPU Nsys Results

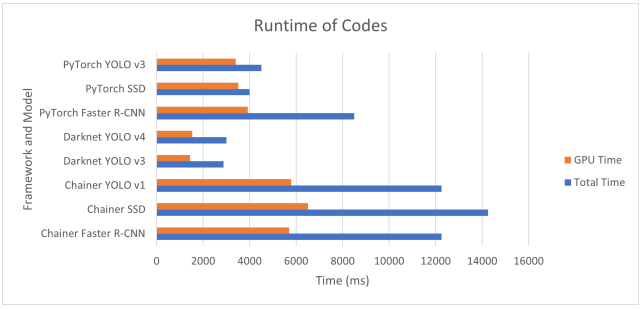

Fig. 7. Runtime of Codes

Model	GPU Inference Time (ms)
Darknet YOLOv3	16.2
PyTorch SSD	23.9
Darknet YOLOv4	33.6
Chainer YOLOv1	44.1
PyTorch Faster-RCNN	163.9
PyTorch YOLOv3	447
Chainer Faster-RCNN	4727.6
Chainer SSD	5685.2

Fig. 8. GPU Inference Time

vein of Darknet, which also utilizes both the CPU and GPU, in order to gain a more accurate perspective.

When looking at accuracy, Darknet has produced the overall fastest inference speed with no drastic decrease in accuracy for object detection utilizing the YOLO framework. It is also worth mentioning that three of the four YOLO framework-model combinations were among the top four fastest GPU inference times as shown by figure 8 in the *Results* section. Therefore, we must conclude that the YOLO model, specifically paired with the Darknet framework, is the most efficient algorithm for consumer grade computing equipment.

Note that all of these results were conducted using consumer grade computing equipment and as such, may vary with other consumer grade or industrial grade computing equipment. With further funding of this type of research, we can gain a better understanding of the full impact that these programs have on other consumer grade computing equipment as it could be tested on a variance of other consumer computing equipment.

IX. Conclusion

One of the most significant takeaways from our resource characterization was that the four models with the highest GPU memory utilization percentage were also the four models with the quickest inference times. In future roadside edge devices, we may therefore be interested in prioritizing memory rather than CUDA cores to optimize speed. We agree that further exploration is necessary to better understand our results and offer clarity on edge requirements. Possible extensions include profiling these models on a variety of GPUs, testing on video input, and performing detection on a large data set with varying pixel count, number of objects, and weather conditions.

References

[1] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[2] Mark Everingham et al. *PASCAL Visual Object Classes (VOC)*. 2007 and 2012. URL: http://host.robots.ox.ac.uk/pascal/VOC/.

[3] Andreas Geiger, Philip Lenz, and Raquel Urtasun. “Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[5] Daniel J Fagnant and Kara Kockelman. “Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations”. In: *Transportation Research Part A: Policy and Practice* 77 (2015), pp. 167–181.

[6] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *Advances in Neural Information Processing Systems (NIPS)*. 2015.

[7] Marius Cordts et al. “The cityscapes dataset for semantic urban scene understanding”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3213–3223.

[8] Wei Liu et al. “Ssd: Single shot multibox detector”. In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[9] Joseph Redmon et al. “You only look once: Unified, real-time object detection”. In: *Proceedings of IEEE conference on computer vision and pattern recognition*. 2016.

[10] Xinlei Chen and Abhinav Gupta. “An Implementation of Faster RCNN with Study for Region Sampling”. In: *arXiv preprint arXiv:1702.02138* (2017).

[11] Tsung-Yi Lin et al. *COCO: Common Objects in Context*. 2017. URL: https://cocodataset.org/#home.

[12] Tsung-Yi Lin et al. “Feature pyramid networks for object detection”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017.

[13] Yusuke Niitani et al. “ChainerCV: a Library for Deep Learning in Computer Vision”. In: *ACM Multimedia*. 2017.

[14] Joseph Redmon and Ali Farhadi. “YOLO9000: better, faster, stronger”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.

[15] J. Ren et al. “Accurate Single Stage Detector Using Recurrent Rolling Convolution”. In: *ArXiv e-prints* (Apr. 2017). arXiv: 1704.05776 [cs.CV].

[16] Shaoqing Ren et al. “Object detection networks on convolutional feature maps”. In: *IEEE transactions on pattern analysis and machine intelligence* 39.7 (2017), pp. 1476–1481.

[17] Mahadev Satyanarayanan. “Edge computing for situational awareness”. In: *IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 2017.

[18] Y. Zhou and O. Tuzel. “VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection”. In: *ArXiv e-prints* (Nov. 2017). arXiv: 1711.06396 [cs.CV].

[19] Toru Baji. “Evolution of the GPU Device widely used in AI and Massive Parallel Processing”. In: *2018 IEEE 2nd Electron Devices Technology and Manufacturing Conference (EDTM)*. IEEE. 2018, pp. 7–9.

[20] Xinyu Huang et al. “The ApolloScape Dataset for Autonomous Driving”. In: *arXiv preprint arXiv:1803.06184* (2018).

[21] Congcong Li. *High quality, fast, modular reference implementation of SSD in PyTorch*. https://github.com/lufficc/SSD. 2018.

[22] Wenjie Luo, Bin Yang, and Raquel Urtasun. “Fast and furious: Real time end-to-end 3d detection, tracking and motion forecasting with a single convolutional net”. In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2018, pp. 3569–3577.

[23] Hang Qiu et al. “AVR: Augmented vehicular reality”. In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM. 2018, pp. 81–95.

[24] Joseph Redmon and Ali Farhadi. “Yolov3: An incremental improvement”. In: *arXiv preprint arXiv:1804.02767* (2018).

[25] Yan Yan, Yuxing Mao, and Bo Li. “Second: Sparsely embedded convolutional detection”. In: *Sensors* 18.10 (2018), p. 3337.

[26] Qi Chen et al. “Cooper: Cooperative Perception for Connected Autonomous Vehicles Based on 3D Point Clouds”. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 514–524. DOI: 10.1109/ICDCS.2019.00058.

[27] Qi Chen et al. “F-cooper: Feature based cooperative perception for autonomous vehicle edge computing system using 3D point clouds”. In: *Proceedings of ACM/IEEE Symposium on Edge Computing (SEC)*. 2019.

[28] Sidi Lu, Yongtao Yao, and Weisong Shi. “Collaborative Learning on the Edges: A Case Study on Connected Vehicles”. In: *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*. 2019.

[29] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[30] Seiya Tokui et al. *Chainer: A Deep Learning Framework for Accelerating the Research Cycle*. Sept. 2019. URL: https://export.arxiv.org/pdf/1908.00213.

[31] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. Apr. 2020. URL: https://arxiv.org/pdf/2004.10934.pdf.

[32] Asif Faisal et al. “Mapping two decades of autonomous vehicle research: A systematic scientometric analysis”. In: *Journal of Urban Technology* 28.3-4 (2021), pp. 45–74.

[33] Glenn Jocher et al. *ultralytics/yolov3: v9.5.0 - YOLOv5 v5.0 release compatibility update for YOLOv3 (Version v9.5.0)*. 2021.

[34] Sarah Maslin Nir. *Traffic in New York City has returned to nearly prepandemic heights, mirroring a trend replicated nationwide*. May 2021.

[35] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. “Scaled-YOLOv4: Scaling Cross Stage Partial Network”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2021, pp. 13029–13038.

[36] Qing Yang et al. “Machine-learning-enabled cooperative perception for connected autonomous vehicles: Challenges and opportunities”. In: *IEEE Network* 35.3 (2021), pp. 96–101.

[37] *Autopilot – Tesla*. https://www.tesla.com/autopilot.

[38] *GeForce – Nvidia*. https://www.nvidia.com/.

[39] *Intel Editorial: For Self-Driving Cars, Theres Big Meaning Behind One Big Number: 4 Terabytes — Business Wire*.
[40] *Velodyne – LiDAR*. https://velodynelidar.com/.