Re-thinking CNN Frameworks for Time-Sensitive Autonomous-Driving Applications: Addressing an Industrial Challenge*

Ming Yang[1], Shige Wang[2], Joshua Bakita[1], Thanh Vu[1], F. Donelson Smith[1], James H. Anderson[1], and Jan-Michael Frahm[1]
[1]Department of Computer Science, University of North Carolina at Chapel Hill [2]General Motors Research

***Abstract*—Vision-based perception systems are crucial for profitable autonomous-driving vehicle products. High accuracy in such perception systems is being enabled by rapidly evolving convolution neural networks (CNNs). To achieve a better understanding of its surrounding environment, a vehicle must be provided with full coverage via multiple cameras. However, when processing multiple video streams, existing CNN frameworks often fail to provide enough inference performance, particularly on embedded hardware constrained by size, weight, and power limits. This paper presents the results of an industrial case study that was conducted to re-think the design of CNN software to better utilize available hardware resources. In this study, techniques such as parallelism, pipelining, and the merging of per-camera images into a single composite image were considered in the context of a Drive PX2 embedded hardware platform. The study identifies a combination of techniques that can be applied to increase throughput (number of simultaneous camera streams) without significantly increasing per-frame latency (camera to CNN output) or reducing per-stream accuracy.**

I. Introduction

This paper reports on the results of an industrial automotive design study focusing on the application of convolutional neural networks (CNNs) in computer vision (CV) for autonomous or automated-driving cars. CNNs are essential building blocks for CV applications that process camera images in diverse uses including object detection and recognition (cars, people, bicycles, signs), scene segmentation (finding roads, sidewalks, buildings), localization (generating and following a map), etc. CV applications based on CNN technology are essential because cameras deployed on cars are much less costly than other sensors such as Lidar.

CV in a resource-constrained setting. While cameras are cheap, CV computational costs are not. Keeping overall monetary cost low is essential for consumer-market cars and requires effective management of computing hardware. Because state-of-the-art CV approaches have been shown to take over 94% of the computational capacity in a simulated vehicle [18], computer hardware costs can be significantly reduced by optimized implementations. Existing CNN frameworks have some amount of optimization with parallelism, but focus largely on processing a single stream of images in a parallel manner. The solutions considered in this study re-think the way CNN software is implemented so that effective resource management can be applied to CNN applications. Our study focuses on one of the most demanding CV applications in cars, the timely detection/recognition of objects in the surrounding environment.

Industrial challenge. The major challenge we seek to address is to resolve the inherent tensions among requirements for throughput, response time, and accuracy, which are difficult to satisfy simultaneously.

The sheer number and diversity of cameras necessary to provide adequate coverage and redundancy (for safety) drives throughput requirements. A typical configuration in today's experimental automated-driving vehicle includes ten or more cameras that cover different fields of view, orientations, and near/far ranges. Each camera generates a stream of images at rates ranging from 10 to 40 frames per second (FPS) depending on its function (e.g., lower rates for side-facing cameras, higher rates for forward-facing ones). All streams must be processed simultaneously by the object detection/recognition application.

For this application, response time (latency from image capture to recognition completion) is time-sensitive because of its position on the critical path to control and actuation of the vehicle. Every millisecond that elapses during object recognition is a millisecond not available for effecting safe operation. Ideally, the camera-to-recognition latency per frame should not substantially exceed the inter-frame time of the input images (e.g., 25 milliseconds for a 40 FPS camera).

Accuracy requirements for detection/recognition depend on a number of factors specific to the functions supported by a given camera. For example, accuracy may be much more important in streams from front-facing cameras than those from side-facing cameras. For most of the design alternatives we consider, our modifications have no effect on the accuracy of the CV CNN. However, we do explore one particular instance of a throughput vs. accuracy trade-off of relevance in systems with many cameras (see Sec. V).

To summarize, our challenge is to *re-think the design of CNN software to better utilize hardware resources and achieve increased throughput (number of simultaneous camera streams) without any appreciable increase in per-frame latency (camera to CNN output) or reduction of per-stream accuracy.*

Our hardware and software environment. Cars are manufactured by integrating components (including computing hardware

* Most of the study described in this paper was conducted during an internship by the first author at General Motors. Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors.

and software) from a supply chain of various vendors ("Tier-1" and "Tier-2" suppliers). To meet the computational demands of CV applications, supplier offerings are usually based on various application-specific integrated circuits (ASICs) such as field programmable gate arrays (FPGAs), or on digital signal processors (DSPs) or graphics processing units (GPUs). FPGAs and DSPs have advantages in power and density over GPUs, but require specialized knowledge for using the hardware and its tooling environment. Moreover, integrating CV applications supplied by different sources using proprietary black-box solutions built on FPGAs or DSPs can increase product cost. Because of these complicating factors, we consider GPUs as the primary solution for CV applications. GPUs have more familiar development tools and a large number of open-source software choices, TensorFlow [1] being a notable example.

In this study, we use the NVIDIA DRIVE PX Parker AutoChauffeur (previously called the 'DRIVE PX2' or just 'PX2') [20]. This embedded computing platform is marketed by NVIDIA as the "first artificial-intelligence (AI) supercomputer for autonomous and semi-autonomous vehicles." The PX2 incorporates multicore processors along with multiple GPUs to accelerate CV applications. With respect to such applications, we focus on a variant of the open-source CNN framework 'Darknet' configured to implement an object detection/recognition program called 'Tiny YOLO' (version 2, 'YOLO' for short) [22, 23, 24]. While our study targets specific hardware and CNN software, our hardware and software choices are representative of the multi-core/multi-GPU hardware and CNN designs that are used and will continue to be used for automated driving.

Our design study. As stated earlier, the fundamental challenge we consider is to design an object-detection application that can support independent streams of images from several cameras. Due to the constraining nature of our hardware, effective utilization of every part of the system is essential. Our preliminary measurements (provided in Sec. V) revealed that GPU execution time dominates the time spent processing each image in YOLO, yet one of the PX2's two GPUs[1] was left completely idle and the second was utilized at less than 40%. This result motivated us to explore system-level optimizations of YOLO as a path to higher throughputs.

Many CNN implementations similarly seem to have limited throughput and poor GPU utilization. As explained further in Sec. III, the root of the problem lies in viewing processing as a single run-through of all the processing layers comprising a CNN. However, in processing an image sequence from one camera, each layer's processing of image i is independent of its processing of image $i + 1$. If the intermediate per-image data created at each layer is buffered and sequenced, there is no reason why successive images from the same camera cannot be processed in different layers concurrently. This motivates us to consider the classic *pipelining* solution to sharing resources for increased throughput.[2] While pipelining offers the potential for greater parallelism, concurrent executions of the same layer on different images are still precluded. While allowing this may be of limited utility with one camera, it has the potential of further increasing throughput substantially when processing streams of images from multiple cameras (all of which are also independent). This motivates us to extend pipelining by enabling each layer to execute in *parallel* on multiple images.

We altered the design of Darknet to enable pipelined execution, with parallel layer execution as an option, in the YOLO CNN. We also implemented a technique whereby different camera image streams are combined into a single image stream by combining separate per-camera images into a single composite image. This compositing technique can greatly increase throughput at the expense of some accuracy loss. We also enabled the balancing of GPU work across the PX2's two GPUs, which have differing processing capabilities. The YOLO variants arising from these options and various other implementation details are discussed in Sec. IV. The options themselves are evaluated in Sec. V.

Contribution. This paper provides an effective solution for a challenging industry problem: designing CNN systems that can provide the throughput, timeliness, and accuracy for CV applications in automated and autonomous driving. Through re-thinking how the YOLO CNN is executed, we show that its throughput can be improved by more than twofold with only a minor latency increase and no change in accuracy. We also show that throughput can be further improved with proper GPU load balancing. Finally, we show that the compositing technique can enable very high throughputs of up to 250 FPS on the PX2. While this technique does incur some accuracy loss, we show that such loss can be mitigated by redoing the "training" CNNs require. We claim these results are sufficiently fundamental to guide CNN construction for similar automotive CV applications on other frameworks or hardware.

In order to understand these contributions in full, an understanding of relevant background information and related work is required. This we provide in the next two sections.

II. Background

In this section, we provide detailed descriptions of our hardware platform, the DRIVE PX2 [20], the Darknet CNN framework, and the object-detection application ("YOLO") that runs using this framework.

A. NVIDIA DRIVE PX2

The PX2 is embedded hardware specifically designed to provide high-performance computing resources for autonomous operation on a platform with small size, weight, and power characteristics.

[1]Technically, the PX2 provides four GPUs split between two systems on a chip (SoCs). We considered only one of these SoCs in our evaluation.

[2]For example, instruction pipelining in processors can fully utilize data-path elements and increase throughput.

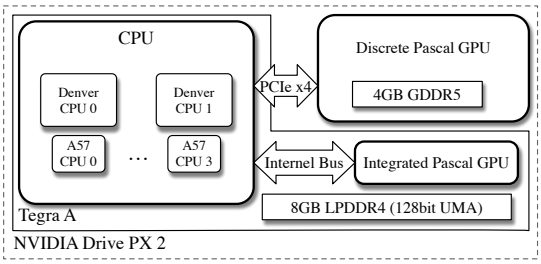

Fig. 1: NVIDIA DRIVE PX2 architecture (showing one of the PX2's two identical and independent systems).

	Integrated GPU (iGPU)	Discrete GPU (dGPU)
Computing units	256 CUDA Cores	1152 CUDA Cores
Accessible memory	6668 MBytes	3840 MBytes
Memory bandwidth [25]	≈50 GBytes/s	≈80 GBytes/s
Shared L2 cache size	512 KBytes	1,024 KBytes

TABLE I: Specifications of the PX2 iGPU and dGPU.

Architecture. The PX2 contains two identical and independent "Parker" systems on a chip (SoCs), called Tegra A and B, connected by a 1 Gbps Controller Area Network (CAN) bus. We show one of the identical halves of the PX2 in Fig. 1. Each SoC contains six CPUs (four ARMv8 Cortex A57 cores, two ARMv8 Denver 2.0 cores), and an integrated Pascal GPU (iGPU). Each Parker SoC also connects to a discrete Pascal GPU (dGPU) over its PCIe x4 bus. While the iGPU and CPU cores share main DRAM memory, the dGPU has its own DRAM memory. Memory transfers between CPU memory and dGPU memory use the PCIe bus. *The remainder of this paper, including the design descriptions and evaluation experiments, use only one of the PX2's two independent systems.*

iGPU vs. dGPU. As previously mentioned, the PX2 has two types of GPUs – the iGPU and the dGPU – using the same Pascal architecture. While they share an architecture, they share little else. Consider their specifications as shown in TABLE I. The dGPU has more cores, faster memory, and a larger L2 cache. The iGPU lags behind, advantaged only by its access to the larger, albeit shared, main memory DRAM banks.

The exact performance differences between the iGPU and the dGPU depend on the characteristics of GPU programs used in a specific application. We characterize these differences for YOLO in Sec. V.

B. YOLO: You Only Look Once

In this section, we describe Tiny YOLOv2 (simply "YOLO" in the remainder of this paper). We use YOLO as an exemplar for a wide class of CNN implementations that can be optimized using the designs presented here. We also use it for empirical evaluations of the new CNN framework.

Convolutional neural networks. YOLO is a CNN that *localizes* and *classifies* multiple objects in an image during one forward pass through the CNN. The localization operation predicts the position and dimensions (in pixel units) of a rectangle ("bounding box") in the image that contains an object of interest. The classification operation predicts with a probability what specific class (car, bicycle, sign, etc.) the object belongs to. Collectively, localization and classification are here referred to as *object detection*.

A neural network can "learn" to localize and classify objects in images via a process called *supervised training* in which a large number of images labeled with "ground-truth" bounding boxes and class identifications are input to the CNN running in training mode. Training typically results in computing a large set of parameters (or "weights") defining how the CNN will respond when given images not in the training set (e.g., images from cameras mounted on a car). For most of the experiments in Sec. V, we used the YOLO CNN as already trained on a widely used set of relevant training images (VOC 2007+12), so we did not modify the YOLO network architecture or training approach. This means that most of our work applies to any CNN constructed using Darknet, from YOLOv3 to AlexNet. This is extremely important as noted in [18] because the state of the art rapidly changes for autonomous vehicles. Only one optional enhancement we consider, namely *image compositing*, requires retraining.

Tiny YOLO implementation. Tiny YOLOv2 is a lower-accuracy, faster-speed derivative of YOLOv2 by the YOLOv2 authors [22, 24]. This version of YOLO runs roughly five times faster than the standard version of YOLO ($\geq$ 200 FPS vs. $\geq$ 40 FPS on a high-end GPU) [24]. The network architecture is shown in Fig. 2, as it would be configured by default for C cameras (denoted 0 through $C-1$). In this default system, each camera uses a separate "private" instantiation of the network. Each instantiation contains nine `convolution` or convolutional layers interleaved with six `maxpool` layers and a `region` layer at the end. All YOLO networks run on the *Darknet* [23] CNN framework. A *convolutional layer* convolves over the input by computing the dot product between input and filters to produce an activation map of features. A *maxpool layer* applies a non-linear transform to the input by picking maximum values out of non-overlapping regions. YOLO's *region layer* uses the activation maps of the last convolutional layer alongside precalculated "anchor" values to predict the output locations and bounding boxes of objects of interest. These anchors characterize the common shapes and sizes of bounding boxes in the training data and help improve the prediction accuracy. All of our modifications and benchmarks are based on a proprietary fork of Darknet, so frame rates in Sec. V are not directly comparable to those from the open-source version.[3] Unfortunately, we are unable to open source our fork due to intellectual property management issues.

The Darknet framework is programmed in C and CUDA. CUDA is an API library and set of C language extensions provided by NVIDIA for general-purpose programming of applications that use a GPU for acceleration. Programs in CUDA run on the CPU and can request execution of programs called "kernels" that are written to run on the GPU. Kernel execution requests are effectively placed in FIFO CUDA stream queues managed by GPU hardware and its device driver. The GPU scheduler executes kernels from multiple queues in a

[3]Specifically, our base version of Darknet pre-loads frames and performs frame resizing in the main thread whereas the open-source version uses a separate thread.

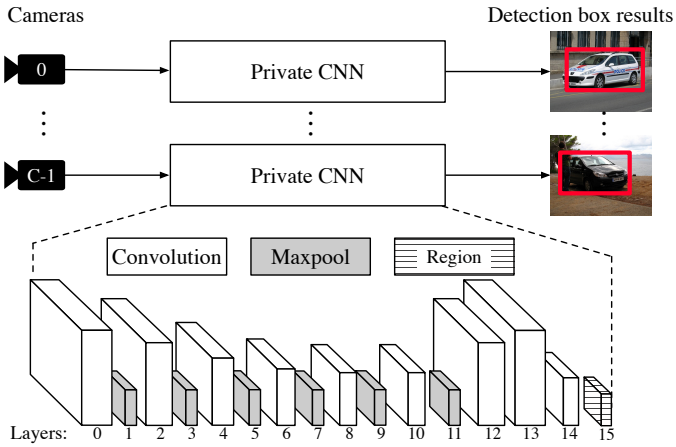

Fig. 2: Traditional camera processing setup and the Tiny YOLOv2 network architecture. Larger boxes indicate longer execution times. Frame data flows left to right.

complicated, semi-fair order but always respects the FIFO ordering within a stream queue. When the CPU requests the execution of a kernel on the GPU, control returns to the CPU shortly after the request is enqueued, but before the kernel is executed. The CPU program must use an explicit synchronization operation to determine when all its requested kernel executions have completed.

Darknet uses a common CUDA programming pattern in its layers. The pattern is (a), if necessary, copy data from CPU memory to GPU memory, (b) start one or more kernels, (c) synchronize on the GPU completion of the kernels, and (d), copy data from GPU memory to CPU memory. Requests for copy operations use specialized hardware on the GPU but are placed in the FIFO stream queues in order with the kernels. The number of kernels run by YOLO varies between layers, but in total may be hundreds of kernels per image.

III. Related Work

YOLO comes from a family of "single-shot" detection approaches. These architectures, pioneered by Redmon and Liu [19, 21], perform all the steps of detection in one run of a single CNN. This allows for higher throughput while competing with the state-of-the-art on accuracy. This builds off several foundational CV CNN works [10, 11, 17].

CNNs inherently consist largely of parallel math, and thus GPUs serve as the ideal execution platform. Unfortunately, the leading GPU manufacturer (NVIDIA) presently holds their architecture and software stacks extremely closely. This secrecy has even left some publicly documented synchronization behaviors of their hardware to be incorrect [27]. These factors have previously made it very challenging to build CV CNNs that consistently or properly utilize the GPU. (We show how inadequate the best existing efforts have been in this paper.)

Fortunately, recent work has begun to address this problem. While AMD's GPUOpen effort [2] helped unveil aspects of GPU design, Amert et al. [3] and Capodieci et al. [5] recently presented more relevant findings. These works uncover details on how CUDA threads are scheduled inside the same process and dispatched to the GPU cores on the NVIDIA Parker SoC, highlighting a set of timesliced FIFO queues at the core. While some other works [16] have critiqued Amert et al.'s efforts for ambiguous applicability to other NVIDIA GPUs, our platform (the PX 2) shares an SoC (the NVIDIA Parker chip) with their work. This, and some limited validation work by Bakita et al. in [4], allows us to safely apply the findings of [3] here.

In the area of parallel CNN scheduling, frameworks such at TensorFlow [1] already support *intra-model* CNN graphs. These graphs serve to enable "parallelism, distributed execution, compilation, and portability" in their applications (per their website). However, neither the original paper [1] nor their survey of CV CNNs [12] seems to consider anything besides parallelism inside one, serialized execution of a model on static data. Their graphs seem mostly to serve to formalize the implicit dataflow graph present in any CNN. In contrast, our work deals with synthesizing an extra-model *shared CNN* by which to collapse multiple CNN instances into a single shared program. It is worth emphasizing that our efforts are designed to largely work alongside the vast body of existing CNN optimization and compression efforts such as DeepMon [14].

One other work similar to ours, S^3*DNN*, applied techniques such as kernel-level deadline-aware scheduling and batch processing (data fusion) that require additional middleware [28]. We focus on higher-level parallelism by applying a graph-based architecture. Such a graph-based architecture was also applied in [26], which focuses on separating CPU and GPU workloads at the kernel level to guarantee response-time bounds.

Some of our work here also considers ways to optimally partition YOLO between GPUs of different capabilities. The partitioning concept is not new, used by Kang et al. to dynamically balance CNN layers between a compute-impoverished system and a networked server with Neurosurgeon [15]. We, however, use the technique to optimize solely for performance without the networking constraints that dominate that work.

IV. System Design

In this section, we discuss some of our ideas for how to implement this shared CNN. First, we describe our overarching approach, which involves generalizing the concept of a CV layer by combining groups of consecutive layers into "stages," examining opportunities for parallelism across stages ("pipelining"), and considering opportunities for parallelism inside each stage ("parallel-pipelining"). Second, our hardware platform provides the somewhat unique opportunity to examine GPU-work-assignment strategies across asymmetric processors, so we briefly consider that issue. Third, we propose a novel technique for realizing additional throughput gains in high-camera-count environments. Along the way, we also briefly touch on several other issues that naturally arise with the concept of a shared CNN. These include data handling, communication issues, and implicit synchronization. Before delving too deeply into the details, however, we first motivate why we are interested in this shared approach in the first place.

The obvious solution to support multiple cameras is simple: run multiple instances of the application – one process for each

camera – and have them share the GPU(s), as shown in Fig. 2. We reject this approach for two reasons. First, the high memory requirements of each network instance heavily limits the total number of instances (a maximum of six on our hardware platform). Second, running the models independently does not easily allow for fast synchronization across multiple cameras. Ideally, we would like each network and camera to process frames at the same rate, prioritizing work for networks that are lagging behind. However, this proves extremely challenging in practice as it requires modifying GPU scheduling behavior – a task still not satisfactorily solved by any research that we are aware of.

A. Enabling Parallelism

As noted in Sec. I, CNN layers and models are inherently independent and thus sharable – the only major complexities for us stem from Darknet's limited "bookkeeping" side-effects and from the need to correctly pipe intermediate data between layers. In this section, we explore ways of enabling parallelism in such a shared computation structure.

Decomposition into stages. As a precursor to introducing parallelism, we decompose a shared CNN into a succession of computational *stages*. Each stage is simply a subsequence of layers from the YOLO CNN shown in Fig. 2, as illustrated in Fig. 3. To correctly manage both frame data and the bookkeeping data required by YOLO, we buffer indices to this data using queues between stages as shown in Fig. 3. Note that these queues are not necessarily index-ordered. Conceptually, we can view the data being indexed as being stored in infinite arrays, but in reality, these arrays are "wrapped" to form ring buffers. The size of the frame ring buffer is large enough to prevent any blocking when adding another frame to it. However, memory constraints limit the ring buffer for bookkeeping data to have only ten entries. This implies that only ten frames can exist concurrently in the shared CNN at any time. Note that Darknet ensures that the original layers execute correctly without further modification by us when provided with the appropriate bookkeeping data. Data buffering and synchronization in our pipelined implementation are provided by PGM$^{\mathrm{RT}}$, a library created as part of prior work at UNC [7].

Serialized execution. Before enabling parallelism, we first look at the scheduling of the default serialized execution of the unmodified YOLO through an example in Fig. 4(a). We will explain Fig. 4(b) and (c) shortly. Fig. 4(a) shows a single three-stage CNN that processes frames serially. At time t_0, the first stage (Stage 0) starts the initial preparation work on the CPU for frame f. This initial preparation work (for example, resizing the frame) requires Stage 0 to execute longer on the CPU than the following stages. Stage 0 launches its kernel, indicated by S0, at time t_1 into the single CUDA stream queue belonging to this CNN instance. Stage 1 launches another kernel, S1, at time t_1, and S2 is launched at time t_2. These kernels S0, S1, and S2 are executed in FIFO order and are completed at times t_4, t_5, and t_6 respectively. Since the intermediate data produced by the first kernels S0 and S1 are not needed for CPU computations, only the last stage suspends to wait for all

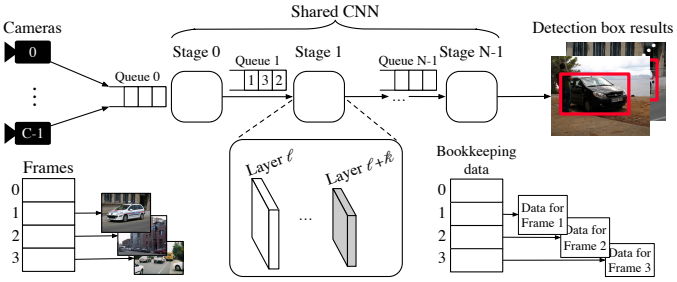

Fig. 3: Sharing one CNN for multiple cameras.

kernels to complete (at time t_6). The use of intermediate data only by kernels running on the GPU is common to other CNN implementations. Because the processing of multiple frames is serialized by the single CNN, the GPU is often left idle when no kernel requests are in the stream queue.

Pipelined execution. We now explore different methods for enabling the parallel processing of stages. The first method we consider is to enable *pipelined execution*, i.e., we allow successive stages to execute in parallel on different frames. We implement pipelined execution by defining a thread per stage, which processes one frame by performing the computational steps illustrated in Fig. 5 for a stage comprised of one convolution layer followed by one maxpool layer. Each stage's thread starts by waiting on the arrival of an index from the previous stage to signal that stage's completion. Once this index becomes available, the thread starts processing by calling the original CNN layer functions with the appropriate bookkeeping data. After these layer functions return, the thread notifies the next stage of completion by pushing the index it just finished into the next queue. It then returns to waiting for input data again. *This approach requires no changes to the original YOLO layer functions.*

The resulting parallelism is illustrated in Fig. 4(b). For conciseness, we henceforth refer to this pipelined-execution variant of YOLO as simply "PIPELINE" and refer to the original unmodified YOLO as "SERIAL." In PIPELINE, multiple per-frame queues are created and pipeline stages can execute concurrently on CPU cores. This enables kernels from different frames to be queued and executed concurrently on the GPU when enough of its resources are available. One example of the concurrent GPU execution of kernels is noted in Fig. 4(b). The threads for stages are scheduled by the Linux scheduler `SCHED_NORMAL`. Threads may migrate between CPU cores or block when cores are not available. The concurrent executions of threads on the CPUs and kernels on the GPU should enable PIPELINE to provide higher throughput or lower latency. We evaluate this in Sec. V. While the longer first stage causes some overall pipeline imbalance, we believe this is largely confined to the CPU. We further mitigate the influence of a pipeline imbalance by enabling thread parallelism within each stage as described next.

Parallel-pipeline execution. As seen in Fig. 4, PIPELINE enables parallelism across stages but not within a stage. Given the independence properties mentioned in Sec. I, there is no reason why the same stage cannot be executed on different

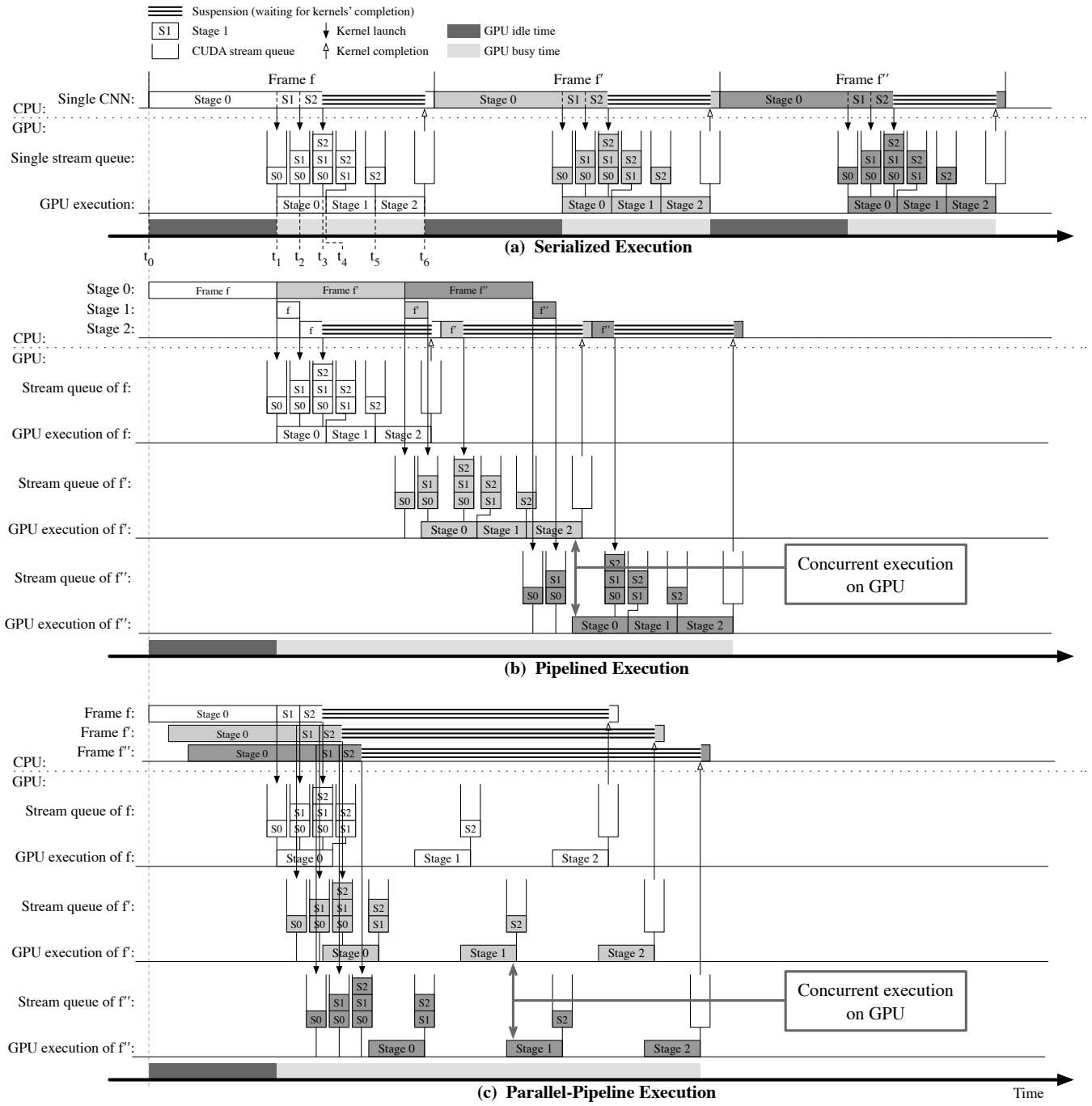

Fig. 4: Scheduling of SERIAL, PIPELINE, and PARALLEL.

frames at the same time. To enable this, we consider a second parallel YOLO variant, called *parallel-pipeline execution*, or simply "PARALLEL" for brevity, that extends PIPELINE by allowing parallel executions of the same stage. The PARALLEL YOLO variant is realized by allocating a configurable number of "worker" threads in each stage, and a "manager" thread that is responsible for dispatching worker threads. The worker threads can process different frames concurrently. This is illustrated in Fig. 6 (once again) for a stage comprised of one convolution layer and one maxpool layer. The difference between PIPELINE and PARALLEL is illustrated in Fig. 4(c). In PARALLEL, more parallelism is enabled on both CPU cores and the GPU through allowing multiple threads of the same stage to execute in parallel. Recalling the scheduling rules revealed in [3], kernels are scheduled in order of their arrival times to the heads of their stream queues. In Fig. 4(c), these arrival times are such that kernel executions from different frames are interleaved, as shown.

With the scheduling shown in Fig. 4, we can see that PIPELINE and PARALLEL achieve the purpose of fully utilizing the GPU resources (much less GPU idle time than SERIAL). We quantify the performance in Sec. V.

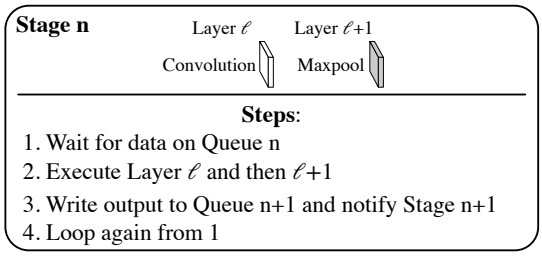

Fig. 5: Stage representation for pipelined execution model.

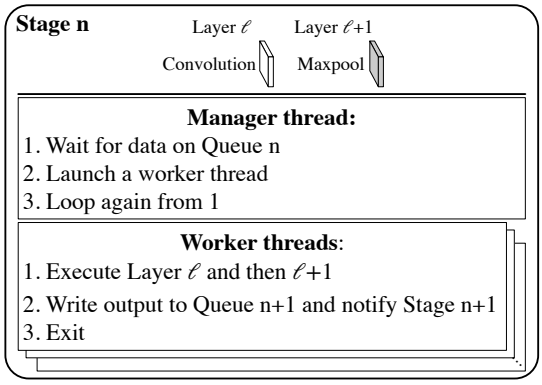

Fig. 6: Stage representation for parallel-pipeline execution model.

B. Multi-GPU Execution

Recall from Sec. II that the PX2 has an iGPU and a dGPU with different capabilities. As seen in Fig. 12, which we cover in detail later, the iGPU tends to execute layers substantially slower than the dGPU. However, notable exceptions (maxpool layers and the region layer) do exist, so the iGPU is still a useful computational resource. Given this observation, we enhanced our PIPELINE and PARALLEL implementations to allow a configurable distribution of stages across GPUs. While utilizing both GPUs may be beneficial, additional data-transfer and synchronization overheads can cause dual-GPU configurations to sometimes under-perform single-GPU ones. We analyze this trade-off experimentally in Sec. V-B.

C. Accelerating Multiple Streams with Multi-Camera Composite Images

What if we could combine several physical cameras into a single "virtual" one, as illustrated in Fig. 7, by combining independent per-camera images into a single composite image as illustrated? Such an approach could have a tremendous impact on throughput. For example, if all physical cameras were combined in groups of four, throughput could potentially be increased by fourfold. Whether this is acceptable would depend on how latency and accuracy are impacted.

To enable this approach to be evaluated, we added flexible support for compositing physical cameras to the YOLO variants already described. Note that these variants themselves are not impacted by this approach: the fact that the images to be processed are obtained by compositing does not change the processing to be done. Because of this, it is possible to have both composite and full images supported in the same system, as illustrated in Fig. 7.

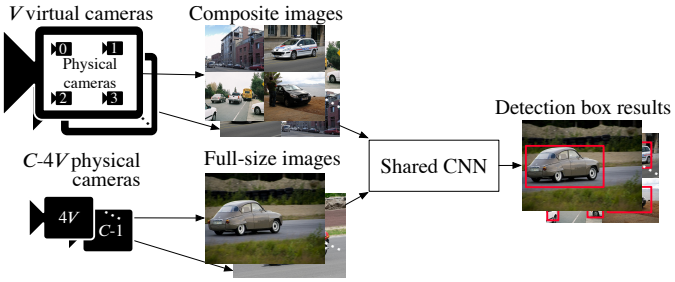

Fig. 7: CNN extended for multi-camera composite images.

The compositing approach works by exploiting the flexibility of YOLO's neural network to process multiple frames as a single frame. This requires taking k^2 frames for some k, tiling them in a grid, and then passing that combined image through YOLO as a single input, as shown in Fig. 7. *For best accuracy, this approach requires re-training the YOLO network on a set of suitably tiled images to produce new model weights.*

While compositing should yield a throughput improvement that is linear in the number of images in the grid, there are several additional factors to consider. First, frame arrival times from different cameras must be synchronized or the combining process will block waiting for all the images required to fill the grid. This could increase latency to the time between frames on the camera with the smallest frame rate. Second, this approach will reduce accuracy. Because our configuration of YOLO only accepts frames of 416 by 416 pixels, each frame in the grid must be downscaled by a factor proportional to the grid size. For detecting objects that are large relative to the frame size (e.g., close vehicles), this compromise may be permissible, however that will not be true in all cases.

Additionally, we must stress that this approach only works for CNNs that reason locally. Using this approach in global-classification systems such as those used in scene understanding *will yield incorrect results*. While the YOLO authors believe their network learned to use global features, we did not observe this behavior in our analysis.

D. Avoiding Implicit Synchronization

The original Darknet framework uses the single default FIFO queue in CUDA for kernel requests. This works flawlessly for serialized execution by a single-threaded program, but falls apart when we want concurrent parallel-pipeline instances. Specifically, use of only the default queue causes destructive implicit synchronization on the GPU, which can significantly degrade performance. As a prior work suggested [27], our approach mitigates that by using CUDA functions to create a unique FIFO queue for the kernel executions and memory transfers of each parallel-pipeline instance.

V. EVALUATION

Our evaluation experiments were designed to answer a number of questions related to our challenge: How many cameras of differing frame rates (FPS) can be supported with the PX2 running a shared CNN? Can acceptable latency be maintained for each camera supported? How can we use the

configuration options in PIPELINE and PARALLEL to increase the number of cameras supported? Are multi-camera composite images a viable option for supporting significant numbers of cameras?

The evaluation of multi-camera composite images must explicitly consider the effects on object detection/recognition accuracy. In all other evaluations of PIPELINE and PARALLEL using single images, we verified that accuracy remained unchanged (as expected, since the CNN layer processing programs were unchanged).

We next describe aspects of the experimental methodology common to all experiments and then proceed to answer our questions. We first evaluate the frame rates enabled by the three basic execution models on single and multiple cameras. Then, we explore the effects of configurable options – stage granularity, the number of threads per stage, and the assignment of stages to GPUs. Finally, we explore the trade-off between throughput and accuracy with multi-camera composite images.

General experiment set-up. In each experiment, the CNN processes an aggregate of at least 10,000 frames. Latency is measured as the time difference between the arrival time of a frame at the first CNN layer and the time when the object detection result for that image is ready. Because the system runs Linux and all its services, worst-case latency outliers can occur. Occasionally dropping a small number of frames when their processing time exceeds a latency threshold is acceptable for our automated-driving platform. Therefore, latencies are reported only for those in the 95^{th}-percentile of all measured values.

Unless otherwise stated, the default CNN configuration has one CNN layer per pipeline stage for both PIPELINE and PARALLEL and ten threads per stage for PARALLEL. In these default configurations, only the PX2's dGPU is used. The choice of these default configurations is explained as part of configuration option evaluation. All the experiments were conducted on the NVIDIA Drive PX 2 with Linux kernel 4.9.38-rt25-tegra with the PREEMPT_RT patch, NVIDIA Driver 390.00, and CUDA 9.0, V9.0.225. CPU threads were scheduled under `SCHED_NORMAL`.

A. How Many Cameras can be Supported?

To answer this question we first consider processing one stream of images by the shared CNN. Cameras are characterized by their frame rates (FPS), and we emulate camera inputs to the CNN at rates common to cameras used in automated driving (10, 15, 30, 40 FPS). We also consider streams with much higher frame rates (up to 100 FPS) to represent aggregate frame rates from multiple cameras sharing the CNN. Our emulated camera is a program running on a separate thread that generates input frames to the shared CNN at a defined inter-frame interval corresponding to a given FPS (e.g., 25 milliseconds for 40 FPS). Because latency is a critical metric, we present results as a plot of latency for different input camera rates and compare execution under SERIAL, PIPELINE, and PARALLEL.

PIPELINE supports 63 FPS, PARALLEL may support 71 FPS. Fig. 8 shows the latency results. Our main criterion for supporting a frame rate is that latency does not become unbounded at that rate. Using this criterion, SERIAL (essentially the original YOLO) can support rates up to 28 FPS and PIPELINE can support up to 63 FPS. PARALLEL can also support 63 FPS with negligible latency reduction and up to 71 FPS before latency becomes unbounded. Note that the latency at rates lower than 63 FPS is essentially constant indicating that 36 milliseconds is the minimum latency for YOLO (as seen on the y-axis of Fig. 8). Latency becomes unbounded in any of these scenarios when either the GPU becomes overloaded or the CNN execution model cannot further utilize the available GPU capacity. The latency increase of PARALLEL at 71 FPS over PIPELINE at 63 FPS is 20 milliseconds. This may be acceptable in scenarios such as maneuvering in a parking lot. However, running multiple SERIAL instances (at most six due to memory constraints) can support at most 53 FPS with latency of 114 milliseconds before getting unbounded latency.

We attribute these results to the inter-stage GPU dispatch parallelism of PIPELINE and the additional intra-stage GPU dispatch parallelism of PARALLEL. These simultaneous dispatches allow us to more fully utilize the GPU and achieve a higher frame rate. Although some dispatch parallelism can be achieved by running multiple SERIAL instances, GPU resource usage is still limited because kernels from different process contexts cannot run concurrently in CUDA. Concurrent kernel execution from different processes can be enabled with NVIDIA's MPS (Multi-Process Service), however, NVIDIA has not implemented this on the PX2.

These results are perhaps most useful as ways to estimate how many cameras with differing rates can share the CNN. As long as the aggregate rate from all cameras is less than the supported total rate, latency should not be compromised. For example, the results show that a single, shared SERIAL YOLO could not support a single 30 FPS camera on the PX2 hardware while PIPELINE and PARALLEL should easily support two 30 FPS cameras. Similarly, PIPELINE can support up to four 15 FPS cameras, or two at 15 FPS and one at 30 FPS, etc.

Note that all these results are from using only one of the two independent computing systems on the PX2 board. Using both systems to run two independent CNN configurations should double the number of cameras supported by one PX2.

Multiple cameras are supported for aggregate frame rates. We verified that estimations based on the FPS of a single image stream are valid for multiple. This experiment investigated whether the additional competition from multiple input streams to the shared CNN changed latencies. The results are shown in Fig. 9 for one, two, and four cameras generating input to PARALLEL YOLO. The supported total camera frame rate is determined by the hardware capacity and the CNN framework execution model, thus it is constant with multiple cameras instead of one. That is, this data corroborates the estimate given above that supporting one 60 FPS camera is the equivalent of supporting two 30 FPS cameras, four 15 FPS cameras, etc. The increased latencies for two and four cameras seen on the y-axis

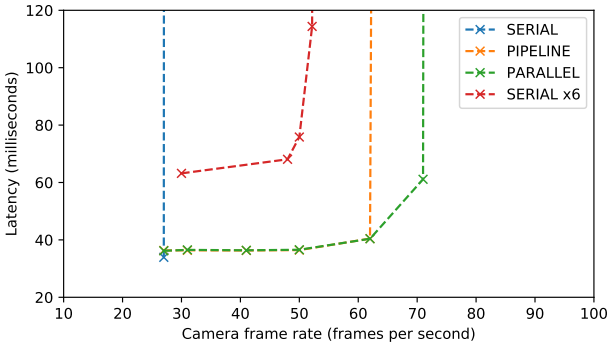

Fig. 8: Latency of SERIAL, PIPELINE, and PARALLEL YOLO.

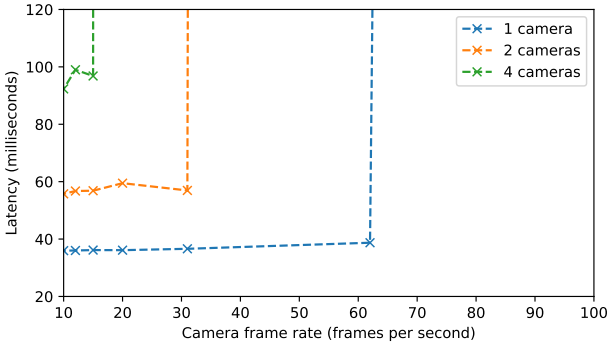

Fig. 9: PARALLEL YOLO latency with different numbers of cameras.

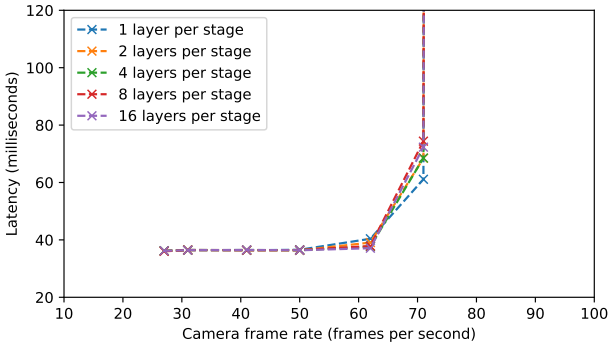

Fig. 10: PARALLEL YOLO latency with different granularities (number of layers per stage).

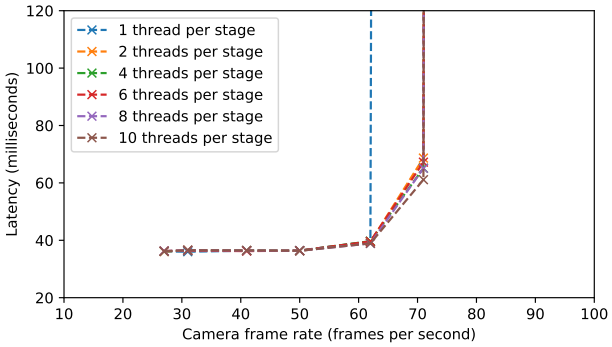

Fig. 11: PARALLEL YOLO latency with different numbers of threads.

in Fig. 9 are an artifact of the experiment. The input images are processed in sequence, so longer latency measurements will be incurred when a frame starts being processed after frames that arrive at the same time from other cameras. These in-phase frame arrivals can be avoided in a real system by setting a different phasing for each camera (a topic for future work).

B. Can Our CNN Implementations Be Better Configured?

PIPELINE can be configured with multiple CNN layers aggregated into the pipeline stages, i.e., configuring the granularity of each stage. PARALLEL can be configured in the same way, but also adds the option to adjust the number of threads assigned per stage, i.e., configure the degree of parallelism within each stage. In addition, sequences of stages can be assigned to either the iGPU or the dGPU on the PX2 to avoid wasting any available GPU capacity. We evaluate these configuration options next.

Granularity of stages improves latency under heavy loads for PARALLEL. In Fig. 10, we compare the latency results at frame rates from a single camera stream for PARALLEL with different stage granularities. We did not observe any impact of granularities on PIPELINE, so those results are not shown here.

Note that pipeline is effectively disabled by configuring a single stage containing all 16 layers. This approximates multiple instances of SERIAL that are running in one process context, a configuration not supported in the original framework.

As shown in Fig. 10, latencies of different granularities up to a frame rate at most 63 FPS (the maximum load before latencies increase significantly, e.g., the max steady state) are very close. While all the granularities can support a heavier processing load up to 71 FPS with an increase in latency, the finer-granularity stages have smaller latencies in this region. Because each stage is implemented by one manager thread and multiple worker threads, more stages with finer granularity enable more parallelism among stages on the CPU. Our design for data communication between stages introduces small overheads, which is reflected in the very small changes in the performance of different granularities when the frame rate is low. *A granularity of one layer per stage is a good configuration choice.*

More threads for PARALLEL improves latency under heavy loads. In Fig. 11, we compare the latency results at frame rates from a single camera stream for PARALLEL with different numbers of threads. More parallelism is enabled with more threads per stage, so more frames can be processed in parallel by a stage. Having only one thread per stage is essentially the same as pipelined execution, so latency becomes unbounded at 63 FPS as shown previously. Between 63 FPS and 71 FPS the latencies increase substantially but higher numbers of threads have somewhat lower latencies and a slower rate of latency increase with load. *A larger number of threads, 10 or so, is a good configuration choice.*

Overhead analysis and performance bottleneck. Enabling PIPELINE and PARALLEL certainly causes additional overheads. We evaluate the computational and spatial overheads with different configurations by comparing the CPU and main memory usage between SERIAL and PIPELINE or PARALLEL. In each experiment, process or thread instances share the six active CPU cores on the PX2. The maximum CPU

	CPUs (%)	Memory (MB)
SERIAL	92	774
SERIAL x6	536	4,644
PIPELINE (single thread)	219	1,132
PARALLEL (10 threads)	239	1,136

TABLE II: CPU and memory usage.

capacity is 600%. As shown in TABLE II, a single instance of SERIAL uses 92% of the CPU capacity, essentially one core's worth. Six instances of SERIAL use 536%, most of the available CPU resources. PIPELINE with one thread per stage uses 219% of the CPU capacity and 1,132 MB of memory. Compared to one SERIAL instance, these figures represent an increase of 127% in CPU capacity and 358 MB with respect to memory. These increases reflect added overheads for synchronization, scheduling, and data buffering. Moving to a PARALLEL implementation with ten threads per stage causes only a slight increase in CPU usage, from 219% to 239%, and a negligible increase in memory usage. We have shown in Sec. V-A that a significant increase in the supported frame rate is achieved with PARALLEL and PIPELINE configurations compared to both one and six SERIAL instances. Under SERIAL, the GPU is around 60% utilized, while under PARALLEL and PIPELINE, it is nearly 100% utilized, shown from CUDA profiling results that are similar to Fig. 4. This demonstrates that using more CPU resources in order to fully utilize the scarce resources of the GPU is a reasonable strategy.

Multi-GPU execution improves latency under heavy loads We evaluate the latency performance of different GPU assignment strategies next to understand the impact of assigning some layers to the iGPU.

Layer performance on both GPUs. We first measured the performance of each layer on both GPUs. We executed SERIAL using each GPU individually and measured per-layer response times and per-layer CPU computation times. Results are shown in Fig. 12, which consists of two parts that use a common x-axis; the top part depicts per-layer dGPU-iGPU performance ratios (obtained by dividing a layer's iGPU response time by its dGPU response time); the bottom part depicts the per-layer data just mentioned. As seen, convolution layers typically take longer on the iGPU, some two to three times longer. Convolution layers contain heavy matrix computations that are completed much faster with more CUDA cores. The maxpool and region layers have lower performance ratios because they have lighter computation loads. *These layers are good choices to be moved to the iGPU.*

GPU assignment strategy. To introduce as little latency degradation as possible while using both GPUs, our strategy is to move layers with low dGPU-iGPU performance ratios to the iGPU. Given the performance ratios at the top of Fig. 12, we evaluate the GPU assignment strategies shown in TABLE III. Each strategy is characterized by the sequence of layer numbers assigned to each GPU. For example, strategy S1 assigns layers 0 to 13 to the dGPU and layers 14 and 15 to the iGPU.

Strategy S1 substantially reduces latency for PARALLEL under heavy loads. Fig. 13 shows that, for PARALLEL, moving the last two CNN layers to the iGPU under strategy S1 improves

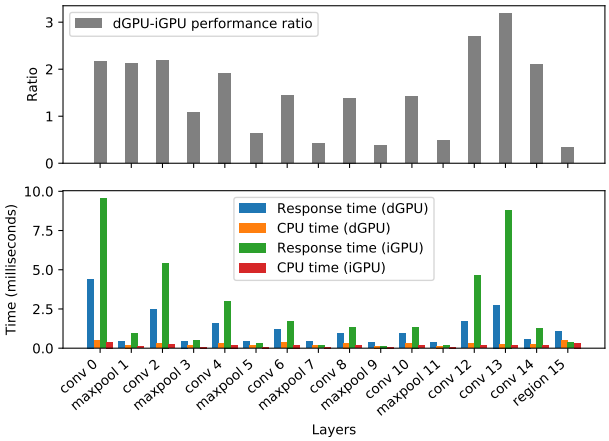

Fig. 12: Per-layer execution times.

Strategy ID	dGPU	iGPU
S0	All	$\emptyset$
S1	$\{0, 1, ..., 13\}$	$\{14, 15\}$
S2	$\{0, 1, ..., 4, 12, 13, 14, 15\}$	$\{5, 6, ..., 11\}$
S3	$\{0, 1, ..., 4, 12, 13\}$	$\{5, 6, ..., 11, 14, 15\}$
S4	$\{0, 1, ..., 7\}$	$\{8, 9, ..., 15\}$

TABLE III: GPU assignment strategies.

latency. S1 provides support for a frame rate of 71 FPS with latency only slightly greater than a frame rate of 63 FPS when using only the dGPU. At a frame rate of 80 FPS, the latency is approximately the same as a frame rate of 71 FPS using only the dGPU. All other strategies considered either increase latencies or have negligible effect.

Strategies S1 and S3 provide some latency reduction for PIPELINE under heavy loads. In Fig. 14, S1 provides support for a frame rate of 71 FPS in PIPELINE with latency only slightly greater than a frame rate of 63 FPS using only the dGPU. S3 for PIPELINE supports a frame rate of 71 FPS with reduced latency comparable to that for PARALLEL using only the dGPU. All other strategies considered either increase latencies or have negligible effect.

C. Are Multi-Camera Composite Images Effective?

Any combination of the features and execution models above fails to provide enough throughput at desired latencies for a vehicle equipped with more than four cameras at a frame rate of 30 FPS using both systems on the PX2 board. We propose to tackle this by leveraging the technique of compositing frames from multiple cameras into one image for processing. Although compositing might introduce a small loss of accuracy as the image resolution will be reduced, this can be acceptable for low-criticality cameras in certain circumstances. We devote this section to evaluate the effectiveness of the technique.

With a multi-camera composite, the throughput is multiplied by the number of images in the composite image. However, the frame concatenation introduces overhead. Also, because the frames are down-sampled as explained in Sec. IV-C, the object detection accuracy could be reduced. We evaluate the throughput, latency, and accuracy of the multi-camera composite image technique.

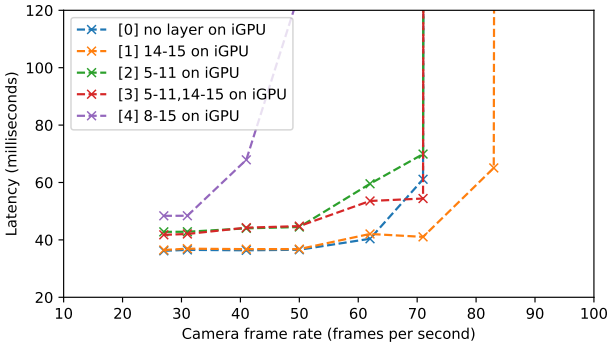

Fig. 13: PARALLEL YOLO latency with different GPU assignments.

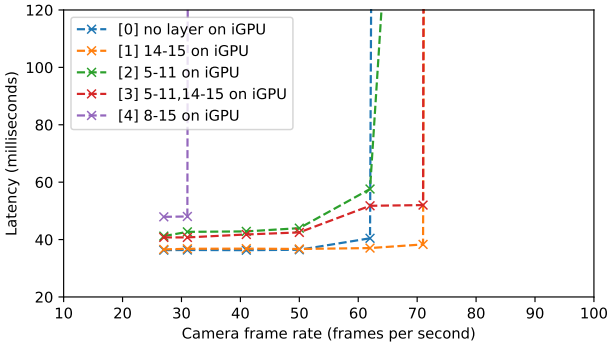

Fig. 14: PIPELINE YOLO latency with different GPU assignments.

Experimental set-up. We focus on comparing YOLO's performance on full-size images and 2x2 composite images. The network is evaluated on a standard object-detection dataset called PASCAL VOC [8], using two types of tests: one of full-size images and one of composite images. For the full-size test, the original VOC test data is used. For the composite, we generate test images by randomly choosing four full-size VOC test images and combining them into a composite image with a 2x2 grid. Each VOC test image is used exactly once as a component of a composite image, hence, the size of the composite test set is four times smaller than the full-size one. This ensures that the total numbers of objects to be detected are the same in both tests. At runtime, full-size and composite images are both resized by YOLO to be 416x416 during processing.

Evaluation of detection results. Evaluation of the network's output is done using *mean average precision* (mAP) – a metric commonly used to evaluate object-detection algorithms [9]. In loose terms, mAP characterizes a model's ability to have high recall and high precision. In other words, higher mAP means more objects are detected and the detections and localizations are more accurate. A detailed explanation of mAP is provided in Appendix A.

Similar to other state-of-the-art detection networks, YOLO is designed to be scale-invariant, meaning it can detect objects at different scales and resolutions. However, when image resolutions decrease significantly (in our case, by a factor of four), the network's performance can start to degrade. As seen in TABLE IV, with a 2x2 grid setup, YOLO trained on the original VOC data can only perform at 36.46% mAP when inferring composite images, an almost 20% accuracy drop.

mAP	Full-size Test	Composite Test
YOLO	55.63	36.46
Composite YOLO	57.64	48.29

TABLE IV: mAPs for full-size and composite tests with PASCAL VOC dataset.

mAP	Full-size Test	Composite Test
YOLO	63.66	44.91
Composite YOLO	66.20	56.21

TABLE V: mAPs of object classes relevant to autonomous driving.[4]

To mitigate this, we retrained the network on the VOC dataset using the same number of training images for both full-size and composite data. The resulting mAPs are shown in TABLE IV. Moreover, since our primary applications are to autonomous driving, we also show in TABLE V the detection results for relevant object classes.[4] Overall, our results consistently indicate that with YOLO retrained on both composite and full-size data, the detection accuracy of composite images is improved without reducing that of full-size images. With the retrained composite network, we can provide a trade-off between a 10% accuracy drop and a fourfold increase of throughput.

Although the mAP decreases with composite images, it is important to remember the following. First, the focus of our work is on providing the option to trade-off between higher throughput and acceptably lower accuracy, and not on the magnitude of the accuracy itself. Second, YOLO represents a state-of-the-art detection network, but we are tackling a very challenging CV problem with strictly limited resources. Third, real-life autonomous driving systems usually deploy tracking algorithms, which can use information of multiple frames to improve detection accuracy. Finally, as shown in [22], YOLO tends to perform better for bigger objects than for smaller ones. Thus, we expect the mAPs to be higher for nearby objects, which are usually prioritized in the case of automated driving.

A four-camera composite image supports 250 FPS (!) and a latency of 48 milliseconds. This is shown in Fig. 15. Examining the y-axis Fig. 15, observe that for lower respective frame rates, latency is higher in the four-camera scenario than in the one-camera scenario (about 45 vs. 36 milliseconds). This is due to the additional overhead incurred to combine multiple frames.

VI. CONCLUSIONS

In this paper, we presented the results of an industrial design study that addresses the challenge of re-designing CNN software to support multiple cameras with better throughput for automated-driving systems. We showed how to transform a traditional CNN framework to obtain pipelined and parallel-pipelined variants to better utilize hardware resources. We also examined the impact of properly load balancing GPU work

[4]Among 20 object classes, we deem the following eight to be irrelevant to the task of autonomous driving: airplane, boat, bottle, chair, dining table, potted plant, sofa, and TV monitor.

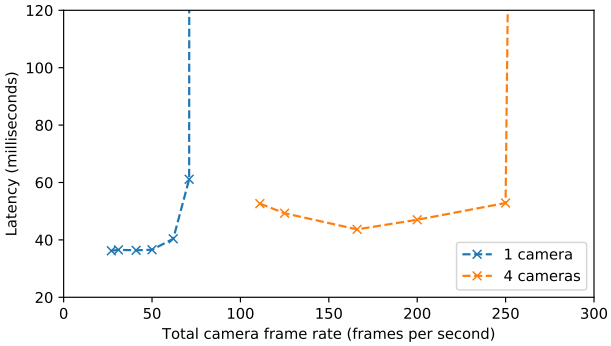

Fig. 15: PARALLEL YOLO latency with multi-camera-image processing and different numbers of cameras.

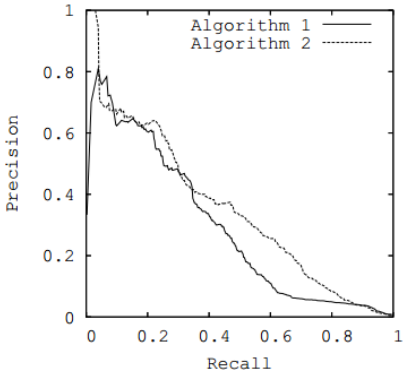

Fig. 16: Example Precision/Recall curves [6].

across the GPUs of a multi-GPU hardware platform. These basic techniques resulted in an improvement in throughput of over twofold with only minor latency increases and no accuracy loss. In order to accommodate more cameras that these basic techniques allow, we proposed the technique of compositing multiple image streams from different cameras into a single stream. We showed that this technique can greatly boost throughput (up to 250 FPS) at the expense of some accuracy loss. We demonstrated that this accuracy loss can be partially mitigated through network retraining.

The results of this study open up many interesting avenues for future work. For example, we would like to determine whether the compositing technique can be dynamically applied at runtime for input streams that may change in criticality. For example, when driving forward, side-facing cameras are less critical and thus could be reasonable to composite. However, when turning, these cameras become more critical, so disabling compositing for them might be appropriate. In multi-mode systems, compositing might also be mode-dependent, in which case it becomes an issue mode-change protocols must consider. In addition to these issues, we would like to explore whether the CNN model can be further optimized to increase detection precision at the center boundaries of a composite image. Embedded hardware platforms other than the PX2 and CV applications other than YOLO also certainly warrant attention.

APPENDIX A: MEAN AVERAGE PRECISION (MAP)

This appendix provides an intuitive understanding of the Mean Average Precision (mAP) metric, which is a popular metric for evaluating object detection accuracy. For a more detailed introduction, we refer the reader to [13] and Section 4.2 in [9] for further details on the precision-recall curve and mAP.

1) IoU, Precision, and Recall

In order to determine the precision of object detection we first need to measure the quality of object detections, which is done through the Intersection over Union (IoU) measure. It is defined as the proportion of overlapping area between the predicted and expected area of an object over their union:

$$IoU = \frac{PredictedArea \cap ExpectedArea}{PredictedArea \cup ExpectedArea} .$$

If the IoU of a prediction is greater than a chosen threshold (e.g., 0.5, as specified in PASCAL VOC dataset), we consider it to be a "correct" detection or a true positive.

Being able to measure correct detections for a given class (e.g., car), enables us to measure the precision and recall for the class, which are defined as follows.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

The precision measures the proportion of correctly detected objects over all predictions, while recall measures the proportion of correctly detected objects over all test objects. A good object detector should be able to detect most of the objects in the scene (high recall/completeness) and do so accurately (high precision/correctness). Please note, there are typically trade-offs to be made, i.e., in order to improve precision usually the recall suffers and vice versa.

2) Precision-Recall curve and Mean Average Precision

A precision-recall curve (see Fig. 16) is used to evaluate these trade-offs between precision and recall or false positives and false negatives [1,3]. For a given class, Average Precision (AP) is a numerical value that summarizes the shape of the precision/recall curve. Specifically, it is calculated by taking the average precision of 11 equally spaced recall levels $\{0, 0.1, , 1\}$ [9].

$$AP = \frac{1}{10} \sum_{r \in 0, 0.1, ..1} p_{interp}(r)$$

$$\text{with } p_{interp}(r) = \max_{r* > r} p(r*)$$

A method with high AP has relatively high precision at all levels of recall and thus, can perform accurate detection for the given class. Mean Average Precision (mAP) is the mean AP over all classes (e.g., car, people, dog, etc.).

ACKNOWLEDGEMENT

We thank Cheng-Yang Fu and Akash Bapat for their assistance with neural network retraining.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI '16*.

[2] AMD, "GPUOpen," Online at https://gpuopen.com.

[3] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *RTSS '17*.

[4] J. Bakita, N. Otterness, J. H. Anderson, and F. D. Smith, "Scaling up: The validation of empirically derived scheduling rules on NVIDIA GPUs," in *OSPERT '18*.

[5] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *RTSS '18*.

[6] J. Davis and M. Goadrich, "The relationship between precision-recall and ROC curves," in *ICML '06*.

[7] G. Elliott, N. Kim, J. Erickson, C. Liu, and J. H. Anderson, "Minimizing response times of automotive dataflows on multicore," in *RTCSA '14*.

[8] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal visual object classes challenge: A retrospective," in *IJCV '15*.

[9] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal visual object classes (VOC) challenge," in *IJCV '10*.

[10] R. Girshick, "Fast R-CNN," in *ICCV '15*.

[11] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *CVPR '14*.

[12] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama *et al.*, "Speed/accuracy trade-offs for modern convolutional object detectors," in *CVPR '17*.

[13] J. Hui, "mAP (mean Average Precision) for object detection," Online at https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173.

[14] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile GPU-based deep learning framework for continuous vision applications," in *MobiSys '17*.

[15] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *ASPLOS '17*.

[16] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar, "A server-based approach for predictable GPU access with improved analysis," in *JSA '18*.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *NIPS '12*.

[18] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *ASPLOS '18*.

[19] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *ECCV '16*.

[20] NVIDIA, "Get under the hood of Parker, our newest SoC for autonomous vehicles," Online at https://blogs.nvidia.com/blog/2016/08/22/parker-for-self-driving-cars/.

[21] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *CVPR '16*.

[22] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *CVPR '17*.

[23] J. Redmon, "Darknet: Open source neural networks in C," Online at http://pjreddie.com/darknet/.

[24] ——, "YOLO: Real-time object detection," Online at https://pjreddie.com/darknet/yolov2/.

[25] S. Sundaram, "DRIVE PX 2: Self driving car computer," in *GTC '16*.

[26] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, "Making OpenVX really 'real time'," in *RTSS '18*.

[27] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems," in *ECRTS '18*.

[28] H. Zhou, S. Bateni, and C. Liu, "S^{3}DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads," in *RTAS '18*.