A Close Look at Multi-tenant Parallel CNN Inference for Autonomous Driving

Yitong Huang, Yu Zhang, Boyuan Feng, Xing Guo, Yanyong Zhang, Yufei Ding

A Close Look at Multi-Tenant Parallel CNN Inference for Autonomous Driving

Yitong Huang[1], Yu Zhang[1(✉)], Boyuan Feng[2], Xing Guo[1], Yanyong Zhang[1], and Yufei Ding[2]

[1] University of Science and Technology of China, Hefei, China
{hyt,guoxing}@mail.ustc.edu.cn, {yuzhang,yanyongz}@ustc.edu.cn
[2] University of California, Santa Barbara, USA
{boyuan,yufeiding}@cs.ucsb.edu

Abstract. Convolutional neural networks (CNNs) are widely used in vision-based autonomous driving, i.e., detecting and localizing objects captured in live video streams. Although CNNs demonstrate the state-of-the-art detection accuracy, processing multiple video streams using such models in real-time imposes a serious challenge to the on-car computing systems. The lack of optimized system support, for example, could lead to a significant frame loss due to the high processing latency, which is unacceptable for safety-critical applications. To alleviate this problem, several optimization strategies such as batching, GPU parallelism, and data transfer modes between CPU/GPU have been proposed, in addition to a variety of deep learning frameworks and GPUs. It is, however, unclear how these techniques interact with each other, which particular combination performs better, and under what settings. In this paper, we set out to answer these questions. We design and develop a Multi-Tenant Parallel CNN Inference Framework, MP-Infer, to carefully evaluate the performance of various parallel execution modes with different data transfer modes between CPU/GPU and GPU platforms. We find that on more powerful GPUs such as GTX 1660, it achieves the best performance when we adopt parallelism across CUDA contexts enhanced by NVIDIA Multi-Process Service (MPS), with 147.06 FPS throughput and 14.50 ms latency. Meanwhile, on embedded GPUs such as Jetson AGX Xavier, pipelining is a better choice, with 46.63 FPS throughput and 35.09 ms latency.

Keywords: Multi-Tenant · Parallel strategy · Autonomous driving · CNN

1 Introduction

Cameras are increasingly deployed on self-driving cars because they offer a much more affordable solution than LIDARs [15]. A car can install a surround-view camera system consisting of up to 12 cameras to capture the panoramic view of the surrounding, which is critical for safe driving [5, 4]. For such a system, each camera continuously generates a video stream. Upon these video streams, convolutional neural networks (CNNs) such as Faster R-CNN [13] and YOLOv3 [12] are usually deployed to achieve accurate object detection. However, these computation-intensive CNN models usually lead to long latency and low throughput (measured by the number of simultaneous camera streams

that are supported) due to the limited processing power of on-car CPU/GPUs. For example, although the original implementation of YOLOv3-416 on the Darknet [11] claims to run within 29 ms on desktop GPU – Titan X (Table 1), its actual latency can go up to 108 ms on Jetson AGX Xavier, an embedded GPU. The problem deteriorates rapidly as more cameras are installed on self-driving cars for better sensing.

Table 1. Specifications and software versions of NVIDIA devices

Platform	**Titan X** (Desktop)	**DRIVE PX2** (Embedded)	**GTX 1660** (Desktop)	**Jetson AGX Xavier** (Embedded)
Price	$1200	$ 15000	$ 219	$ 699
CPU	–	8-core ARM	8-core Intel i7[†]	8-core ARM
GPU	3584 CUDA cores	1408 CUDA cores×2	1408 CUDA cores	512 CUDA cores
Power	250W	80W	120W	30W
FP32 OP/s	11 TFLOPS	8 TFLOPS	4.3 TFLOPS	5.5 TFLOPS
INT8 OP/s	44 TOPS	20-24 TOPS	-	22 TOPS
Software	[11]	[15]	MPInfer	MPInfer
CUDA	-	9.0	10.2	10.1
TensorRT	-	-	7.0.0	6.0.1
libtorch	-	-	1.3.1	-

[†] This Intel CPU does not belong to the GTX 1660 GPU card

Recently, this problem has aroused the attention from both the academic and industrial community. Firstly, the wide adoption of deep learning models has accelerated the upgrade of GPUs and the development of inference frameworks. For example, TensorRT[10] is a high performance inference framework that NVIDIA has introduced and updated frequently since 2017. Meanwhile, new GPU hardware includes desktop GPUs like GTX 1660, GTX 2080 and embedded GPUs like DRIVE PX2, Jetson AGX Xavier, etc. Secondly, several aspects of parallel execution modes have been explored and applied to deep learning inference. For example, in [15], YOLOv2 is altered to enable pipelined execution, with parallel layer execution as an option, in order to improve the throughput; in [3] and [6], the authors propose to boost the throughput via cloud-based CNN inference and dynamic batching; the effect of the CUDA context parallelism mode and the NVIDIA MPS mode is studied in [6], and several data transfer modes between CPU/GPU are evaluated in [1].

With the emergence of these hardware/software techniques, it calls for a comprehensive study of the GPU system that can carefully evaluate the effect of these techniques, preferably with combinations of techniques at different levels. Such an evaluation study should bear in mind the characteristics of self-driving systems such as low latency, low power consumption and low computing resources, and should be able to answer: (1) which parallel execution mode yields the best performance, by how much; (2) how does this performance gain change with the number of concurrent video streams, the number of parallel threads and the data transfer modes; (3) how much do the inference framework and CNN models impact the multi-stream video inference performance.

In this paper, we set out to conduct such a comprehensive evaluation study to answer these questions. We develop a Multi-tenant Parallel CNN Inference framework, MPInfer, to support several proposed techniques and serve as a common platform for performance comparison. On MPInfer, we carry out detailed performance characteri-

zation that considers different parallel execution modes, data transfer modes between CPU/GPU (pageable, pinned, unified), and GPU devices. In particular, three aspects of parallel execution modes are carefully examined: 1) pipelining, 2) batching, and 3) GPU parallelism. Here, the GPU parallelism includes: 1) parallelism across CUDA streams in a single CUDA context; 2) parallelism across CUDA contexts and 3) parallelism across CUDA contexts enhanced by NVIDIA Multi-Process Service (MPS mode for short below) [9]. Targeting at optimizing CNN inference for self-driving systems, we include embedded GPUs in our study and choose TensorRT as the baseline learning framework because of its high performance. We choose one-stage YOLOv3 and two-stage Faster R-CNN for evaluation.

We conduct the performance evaluation on GTX 1660 and Jetson AGX Xavier. Table 1 lists the specifications and software versions of these two architectures as well as Titan X and PX2 which were used by [11] and [15], respectively. Our results show that, when using YOLOv3, the MPS mode achieves the best performance on GTX 1660, reaching 147.06 FPS throughput and 14.50 ms latency. While on Jetson AGX Xavier, pipelining is a better choice, reaching 46.63 FPS throughput and 35.09 ms latency. In addition to the overall trend, we also have the following detailed observations. Firstly, when the workload (*e.g.,* 5 cameras) is close to the system capacity, the latency of MPS mode does not increase with the parallel number. Instead, when the workload (*e.g.,* 6 cameras) is much higher than the capacity, the latency of MPS mode increases approximately linearly with the parallel number. Secondly, pageable/pinned memory have similar performance impact on MPS while the unified memory performs the worst. Thirdly, compared to LibTorch, TensorRT reduces the latency by 61.7% and improves throughput by $3.19\times$. Fourthly, both CUDA stream and pipelining modes achieve the best energy efficiency (0.147 J of GPU per frame) on Jetson AGX Xavier. Finally, using YOLOv3 can support 5 cameras while using Faster R-CNN can only support 2 cameras.

2 Background and Related Work

GPU Programming Models. The CUDA programming model provides the CUDA stream and CUDA context for GPU parallelism. A sequence of operations in a CUDA stream execute in issue-order on the GPU. Operations in different streams can execute in parallel. A CUDA context groups CUDA streams. Different CUDA contexts execute in a time slicing style. The key difference is that different CUDA streams have shared address space while CUDA contexts do not, and CUDA context also introduces overhead during GPU context switch. Moreover, NVIDIA provides MPS for transparently enabling co-operative multi-process CUDA applications. Despite these potential benefits, CUDA programming model and optimization techniques also exist non-obvious pitfalls [16], and are largely unrecognized among the deep learning community, which hurdles the exploitation of these techniques in accelerating CNN inference.

Object Detection and Autonomous Driving. Auto-driving continuously captures images of surrounding and detects objects in these images to follow the road signs and avoid collision. These targets naturally align with object detection, leading to the wide deployment of CNNs in self-driving cars for achieving the high detection accuracy and high safety. Among these CNNs, YOLOv3 has become the most prevalent CNN due

to the low computation overhead and the high accuracy. However, a tension exists between the high resource consumption of CNNs and the limited budget on self-driving cars. This tension exacerbates when multiple cameras are deployed on the self-driving cars and image streams keep coming from these cameras. This tension motivates the design of MPInfer, a multi-tenant parallel CNN inference framework for auto-driving.

Efficient CNN Inference Framework. Several works have been proposed to accelerate CNN inference for autonomous driving. NVIDIA TensorRT Inference Server [3] is a containerized server to deploy models from different frameworks in data centers and it improves utilization of both GPUs and CPUs. [15] introduces the pipelined execution of CNN models for autonomous-driving applications. LibTorch is the C++ version of the popular framework PyTorch. Darknet is a C framework introduced by the author of YOLO. TensorRT [10] provides a SDK for the high-performance deep learning inference. While the inference of individual CNN is accelerated, it is unclear how these frameworks impact the multi-stream video inference performance.

3 Design of MPInfer

Below, we present the overview, design goals and issuesof MPInfer, which is a common platform to conduct comprehensive evaluation of various parallel execution modes.

3.1 Overview of MPInfer

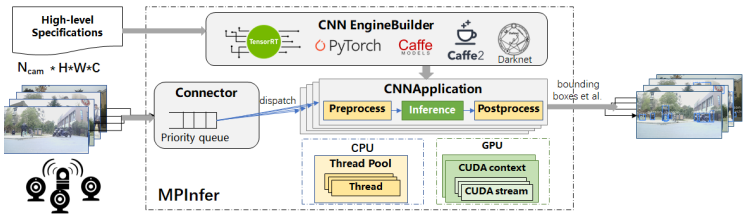

Fig. 1. Overview architecture of MPInfer

We show the overview of MPInfer design in Fig. 1. MPInfer takes a CNN model's high-level specifications and tunable parameters (*e.g.,* batch size and data type) as input. *CNNEngineBuilder* then utilizes these specifications to generate a set of optimized CNN models for efficient execution. We design *CNNEngineBuilder* to be general enough for handle a large variety of CNN inference models (*e.g.,* Faster R-CNN and YOLO), considering the available resource budget and application requirements (such as inference accuracy). During the execution, MPInfer takes image streams from multiple cameras and queues these images in the connector with a *priority queue*. The *priority queue* orders the received images based on a heuristic task priority algorithm, and then dispatches them to the subsequent *CNNApplication*. *CNNApplication* holds the user-specified CNN model generated by *CNNEngineBuilder* along with the necessary pre-

and post-processing, and maintains a pool of workers to execute each task and enables different parallel strategies across image streams. Finally, MPInfer outputs results such as bounding boxes to the next module (*e.g.,* planning) in the self-driving system.

3.2 Design Goals

Configurablility. Configurability is one of the key design goals because MPInfer is designed to support several proposed techniques. Our framework is equipped with the hyper-parameters on the number of cameras or the number of video streams N_c , enabling the full characterization of the workload. Upon this characterization, MPInfer maintains tunable parameters on the number of threads N_t, parallel strategies S, and interactive CPU/GPU memory management techniques M for fully unleashing the parallel computing ability in different GPU acceleration hardware. Although MPInfer is built on the high performance framework TensorRT, MPInfer is also equipped with conditional compilation to support libTorch and Darknet.

Efficiency. Our framework is also designed to choose the best parallel execution mode in a specific configuration which achieves high throughput, low latency and small frame loss rate (FLR). Throughput is measured by the number of images processed in a fixed period. A typical configuration in today's experimental auto-driving vehicle includes a surround-view camera system with up to 12 cameras. Each camera generates a stream of images at rates ranging from 10 to 40 frames per second (FPS) depending on its functionality (*e.g.,* lower rates for side-facing cameras, higher rates for forward-facing ones) [15]. All streams must be processed simultaneously by the object detection/recognition application. A minimum rate of 30 FPS was identified as a real-time boundary [14]. Latency (from image capture to recognition completion) is another relevant critical performance factor and is directly relative to safety for autonomous-driving. The camera-to-recognition latency per frame for real-time autonomous driving should not exceed the inter-frame time of the input images (*e.g.,* 33 ms for a 30 FPS camera). Since there are multiple video streams at a certain frequency, frame loss rate (FLR for short below) is considered as another important metric. FLR can be measured by $FLR = (N_{input} - N_{processed})/N_{input}$. The FLR here is used to show the impact of throughput because the input rate is usually not the same as the processing throughput.

3.3 Design Issues

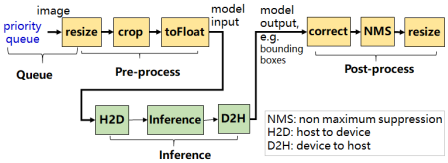

Fig. 2. Process of three-stage CNN inference

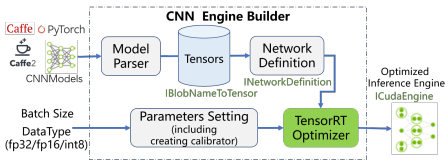

Fig. 3. CNN Engine Builder

Detailed Process of Three-Stage CNN Inference. We specify the detailed process of three stages in MPInfer, as illustrated in Fig. 2. The first stage is the *Pre-process* stage on the input images, which resizes the images and crops the input images to a target resolution, and then converts the data type of the image to be float for further processing. The second stage is the *Inference* stage, which first transports the data from the host (*e.g.,* mobile platforms or desktop servers) to the GPU device, namely H2D. The GPU device will conduct inference with CNNs on the received images and generate predictions. These predictions will be further transported to the host for the next stage, namely D2H. The third stage is *Post-process*, which corrects the generated predictions and conducts non-maximal suppression (NMS)[2] for improving the prediction accuracy. Finally, the predictions are resized to fit original image size and applied to the input images as the final results.

CNN Engine Builder. MPInfer introduces a CNN engine builder to build an optimized CNN inference model by calling TensorRT, as shown in Fig. 3. The engine builder consumes two user inputs of the CNNs and the high-level specifications. One input is the CNN model written in Caffe [7] for describing the CNN layer types, layer parameters, and the topological CNN structures. The other input is the user-defined high-level specifications on the batch size and the data type. Batch size N_b decides the trade-off between the latency and the throughput, where a larger batch size usually leads to higher throughput at the cost of higher latency. The data type of CNN weights and tensors $T_{op} \in \{\texttt{fp32}, \texttt{fp16}, \texttt{int8}\}$ guides the optimization of CNN models and shows a significant impact on the inference speed. In particular, `int8` data type consumes $4\times$ less GPU memory and shows proportional speedup compared to `fp32`, while maintaining a comparable inference accuracy [8]. The CNN engine builder feeds these two inputs to the TensorRT optimizer for optimizing individual CNN inference and generating an optimized inference engine with significantly decreased inference latency. TensorRT is chosen to be part of the infrastructure in MPInfer because it is an efficient framework that supports graph optimization, auto-tuning, and `int8` quantization. We reuse it to achieve better performance when using different parallel strategies.

Execution Strategies. One solution to support multiple camera streams is the serial execution. However, it fails to benefit from the parallel execution of heterogeneous equipments(CPU and GPU). To avoid this, we can do the *Pipelining* between CPU and GPU. As jobs on CPU and GPU can run asynchronously, the pre-process job for the current input on CPU can perform in parallel with the inference job of the previous input on GPU. However, due to the complex CUDA kernel design of operators in CNNs, the current operator implementations cannot fully utilize the GPU resource. To further utilize GPU, we can perform tasks in parallel on GPU. Fig. 4 shows three execution strategies discussed previously. Although parallel execution in this figure only shows three workers executing in parallel, MPInfer can support more or fewer workers as needed. Longer inference latency of *Parallel* execution is due to the parallel GPU. Post-process is left with inference due to its small latency.

Resource Management of Different GPU Parallel Modes. As mentioned in Section 2, modern GPU programming model provides CUDA stream, CUDA context and MPS for specifying fine-grained parallelism levels. We apply these techniques in MPInfer to fully utilize GPUs and design two resource management modes to support these

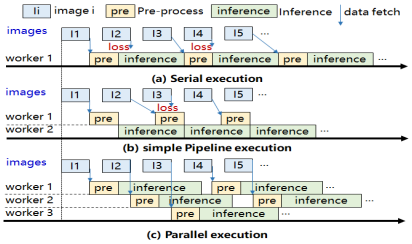

Fig. 4. Different execution strategies. The box width indicates the execution latency

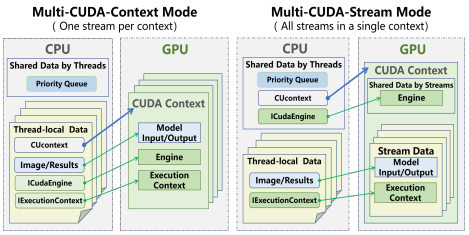

Fig. 5. Memory consumption in different CUDA context/stream modes

parallelism levels, as illustrated in Fig. 5. The first mode is *Multi-CUDA-Context*, where one CUDA context contains only one CUDA stream. In this mode, each thread maintains a `CUcontext` for the metadata on the CUDA context execution, an `ICudaEngine` for the optimized CNN model (network structure and weights), and `IExecutionContext` for intermediate activation values on the CNN execution. Because the resource isolation between CUDA contexts, each thread needs to instantiate the CNN model (`ICudaEngine` in TensorRT) alone, which causes the replica of CNN models and weights. The second mode is *Multi-CUDA-Stream*, where all CUDA streams exist in a single CUDA context and share the same engine for the optimized CNN model. This mode significantly reduces the memory overhead in repeatedly storing the same CNN model. Since MPS is a transparently server enabling co-operative multi-process CUDA applications, the resource management between MPS and CUDA context is the same. For short, CUDA context is similar to the process in the operating system, CUDA stream is similar to thread, and MPS is a server to simulate multiple processes as multiple threads.

Data Transfer Modes Between CPU/GPU. As illustrated in Fig. 2, the host input data (images, CPU side) of CNN network should be transferred to device (GPU) memory and the device output of network also needs to transfer to host side. There are three different data transfer modes between CPU/GPU for such operations. The first one is default *pageable memory allocation* of the host data (*e.g.,* memory allocated by `malloc`). When a data transfer of such memory occurs, GPU needs to first allocate a temporary page-locked (or pinned) memory, then transfers pageable memory to pinned memory and finally transfers pinned memory to device memory. The second one is *pinned memory allocation* of host data (*e.g.,* memory allocated by `cudaHostAlloc`). This does not need to allocate a temporary pinned memory and do the related transfer as the first one. While the allocation of pinned memory will reduce the available physical memory for OS and program, too much allocation will reduce system performance. The third one is the *unified memory allocation* of transfer data. Such memory is a single memory address space accessible from CPU and GPU, so the explicit memory transfer is not needed and the internal software/hardware will do this for the program. MPInfer separately manages the input and the output of the CNN network in each thread to support these three data transfer modes between CPU/GPU, as illustrated in Fig. 5.

Effects of Batching. Batching the input images can reduce the number of CUDA kernel launches. However, the number of cameras used for autonomous-driving is only

up to 12 and the wait time to get enough batch size is unacceptable. When the batch size is smaller than 8, latency of individual image increases approximately linearly according to batch size.

4 Evaluation

4.1 Experimental Settings

The measurements are performed on a desktop server GPU (GTX 1660 + 8-core Intel i7) and an embedded GPU system (Jetson AGX Xavier), as detailed in Table 1. We conduct object detection experiments using the officially trained YOLOv3 model in Darknet format and Faster R-CNN in caffe format. Since the GTX1660 platform can support PyTorch, we use the version generated by PyTorch from Darknet as the baseline, denoted as `LibTorch`. For the embedded Jetson AGX Xavier, we directly use Darknet version as a baseline. The raw image size is 768×576, the input size of YOLOv3 is 416×416 and the input size of Faster R-CNN is 375×500. The length of the priority queue is set to 1, and the latest frame has the highest priority. We use int8 quantization of TensorRT to accelerate the CNN inference.

Since the common frequency of camera is 30Hz, we emulate camera inputs to the CNN at rates 30FPS or its multiples. We measure the latency, throughput and FLR over 10,000 input images in different frame rates. Latency is measured from the arrival time of the image in the priority queue to the time we get the bounding boxes, so the latency includes all three stages in the CNN inference. Throughput is measured by the number of processed images in this time period. Frame loss rate (FLR) is measured by $FLR = (N_{input} - N_{processed})/N_{input}$, where N_{input} is the total number of input images and $N_{processed}$ is the total number of processed images.

We measure the energy consumption on the embedded Xavier platform, which has two INA3221 monitors providing the current power consumption of GPU, CPU, SOC, DDR, and system 5V. We write a program to read `/sys/bus/i2c/drivers/ina3221x` system file once per second and accumulate the current power consumption values to obtain the energy consumption. Based on practical experience, the power profile is set to 30W to get the best performance.

4.2 Experimental Results

Finding the Best Parallel Execution Strategy. Table 2 (a) and (b) show the latency, the throughput, and the FLR of different strategies under different number of cameras each at 30 FPS on two GPU platforms when using YOLOv3. Columns "Queue" and "Total" indicate frame wait time in queue and the end-to-end latency including frame wait time, pre-process time, inference time of YOLOv3 and post-process time. The two tables just show the best setting of different strategies. CUDA stream mode in these two tables uses pinned memory to allocate the transfer data and all others use pageable memory. The number of threads has shown in these two tables.

On GTX 1660, MPInfer can support 5 cameras while maintaining extremely low FLR, and the best parallel mode is achieved using "`MPS+3 CUDA Contexts`", where the

Table 2. Comparison of MPInfer with advanced non-multi-tenant frameworks

(a) GTX 1660

Framework	N_c	Latency(ms) Queue	Latency(ms) Total	Throughput (FPS)	FLR (%)
LibTorch (serial)	4	4.17	35.30	31.85	73.16
	5	3.15	34.38	31.74	78.54
	6	2.73	34.00	31.68	82.10
TensorRT (serial)	4	3.71	13.52	101.62	14.06
	5	3.22	13.01	101.84	30.89
	6	2.71	12.87	97.99	44.40
MPInfer (pipelining)	4	0.08	9.56	118.65	0.02
	5	3.20	12.89	129.23	12.62
	6	2.73	12.54	127.28	28.08
MPInfer (3 CUDA Streams)	4	0.04	9.48	118.69	0.04
	5	0.06	18.83	147.78	11.17
	6	1.30	21.04	145.03	25.85
MPInfer (2 CUDA Contexts)	4	0.04	10.11	118.03	0.04
	5	2.47	17.60	130.39	11.17
	6	2.45	17.67	130.37	25.85
MPInfer (MPS+3 CUDA Contexts)	4	0.03	9.89	118.17	0.00
	5	0.04	**14.50**	**147.06**	**0.01**
	6	1.30	20.40	148.02	17.51

(b) Jetson AGX Xavier

Framework	N_c	Latency(ms) Queue	Latency(ms) Total	Throughput (FPS)	FLR (%)
Darknet (serial)	1	16.77	114.42	10.24	65.74
	2	8.43	106.16	10.23	82.79
	3	5.48	103.73	10.17	88.53
TensorRT (serial)	1	0.18	27.11	29.57	0.05
	2	8.30	35.49	36.72	36.82
	3	5.03	32.76	36.01	58.55
MPInfer (pipelining)	1	0.13	28.18	29.70	0.02
	2	8.85	**35.09**	**46.63**	**20.18**
	3	6.10	31.43	46.68	45.83
MPInfer (2 Streams)	1	0.13	27.40	29.72	0.00
	2	8.73	49.85	48.59	16.73
	3	6.01	47.15	48.57	43.94
MPInfer (2 Contexts)	1	0.70	32.26	29.52	0.49
	2	7.96	57.22	39.78	31.54
	3	3.34	56.50	36.54	57.73

Note: MPS does not work on Jetson AGX Xavier

end-to-end latency is 14.5 ms, throughput is 147.06 FPS and FLR is only 0.01%. Although "`3 CUDA Streams`" achieves similar throughput to "`MPS+3 CUDA Contexts`", the latency of "`MPS+3 CUDA Contexts`" is 23.0% shorter than "`3 CUDA Streams`". This is partially because the resources of each thread in the MPS parallel mode are relatively independent, thus resulting in a lower overhead in GPU resource scheduling.

On Jetson AGX Xavier, MPInfer can fully support 1 camera and can roughly support 1.5 cameras. Because the best throughput under 2 cameras is much higher than 1 camera, we compare the performance under 2 cameras. The best parallel mode under 2 cameras is achieved using "`pipelining`", where the end-to-end latency is 35.09 ms, throughput is 46.63 FPS and FLR is 20.18%. Although "`2 CUDA streams`" achieves the highest throughput, the throughput of "`pipelining`" is only 4.0% worse than"`2 CUDA streams`", while the latency of "`pipelining`" is 29.6% shorter than that of "`2 CUDA streams`". Compared to GTX 1660, Xavier has much less computing power and therefore can't support too many parallel threads.

Finding the Best Inference Framework: TensorRT vs. LibTorch vs. Darknet. On GTX 1660 under 4 video streams, `TensorRT` achieves throughput of 84.50 FPS. Compared to `LibTorch`, `TensorRT` reduces latency by 61.7% and improves throughput by $3.19\times$. On Jetson AGX Xavier under 2 video streams, `TensorRT` significantly reduces latency by 66.6% and improves throughput by $3.59\times$ compared to `Darknet`. Most of the benefit here comes from `int8` quantization. When we conduct serial inference on GTX 1660, we can achieve $2.53\times$ speedup with the 7.57ms inference latency, as opposed to the 19.17ms latency in the fp32 mode.

Finding the Best System Configuration: Camera Number, Frame Wait Time and FLR. The throughput of the system is limited by the aggregated camera frame rate when it is low, and increases as the number of camera increases until reaching its limit. Comparing across N_c from 4 to 6 on GTX 1660, "`MPS+3 CUDA contexts`" significantly

increases the throughput from 118.17 FPS to 148.02 FPS and the latency also increases from 9.89 ms to 20.40 ms, due to the increased number of images in each minute.

When frames arrive faster than the process rate, frames stay in the queue until a worker is free. For serial structures like LibTorch, Darknet, and TensorRT, if the arrival time interval becomes shorter than the time required to process one image, the time that a frame stays in the queue will suddenly rise. A frame loss happens when the queue is full, the oldest frame is ejected to give way to the new frame. Note that the high FLR indicates that a large portion of images are not processed, which dramatically obstructs utilizing multiple cameras for the improving driving safety.

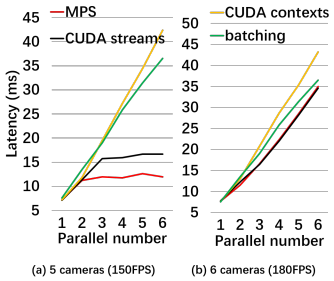

Fig. 6. Inference latency of Different Parallel Strategies on GTX 1660

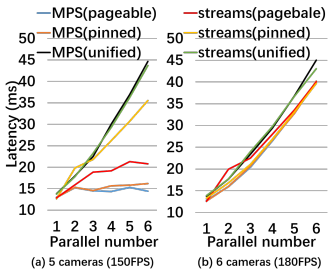

Fig. 7. Impact of Different Data Transfer Modes between CPU/GPU on GTX 1660

Impact of Different Parallel Strategies on Inference Latency. Under 6 video streams at 180 FPS, Fig. 6 shows the impact of different parallel strategies on the inference latency on GTX 1660 when using YOLOv3. Inference latency here does not include pre-process and post-process. Parallel number indicates the number of CUDA streams /contexts or the batch size of batching mode. Batching runs serially here. Fig. 6 shows that latency of CUDA context and batching increases approximately linearly according to the parallel number and CUDA context is the worst one. Fig. 6 (a) shows that when the workload is similar to the system capacity, the latency of MPS and CUDA stream will first increase and then become smooth as the parallel number increases. Meanwhile, MPS is the best one. Fig. 6 (b) shows that when the workload is much higher than the system capacity, all these strategies increase approximately linearly according to the parallel number. At such case, MPS and CUDA stream show similar performance and are better than the other two approaches. This is because the execution of MPS and CUDA stream is almost the same under a heavy workload.

Impact of Different Data Transfer Modes Between CPU/GPU. Fig. 7 shows the impact of different data transfer modes between CPU/GPU on GTX 1660. From the figure, we find that the unified memory is always the worst, because it needs to maintain memory consistency between CPU and GPU. Fig. 7(a) also shows that pinned memory is the best choice for CUDA stream mode and pageable/pinned memory have similar performance impact on MPS. This is because each thread in MPS has a separate memory space while each thread of CUDA stream shares the memory space of the CUDA con-

text. Fig. 7(b) shows that when the input workload is high, pageable/pinned memory have similar performance impact on both MPS and CUDA streams.

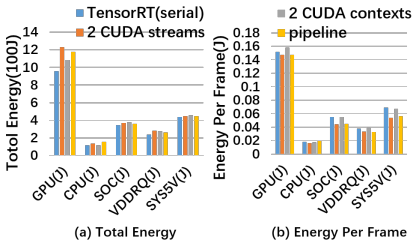

Fig. 8. Energy consumption on Jetson AGX Xavier under 2 video streams (60FPS)

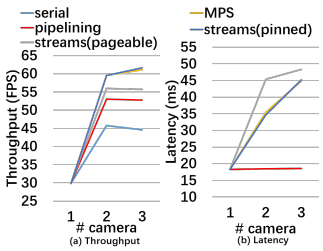

Fig. 9. Throughput and latency of Faster R-CNN

Energy Consumption. Fig. 8 (a) gives the total energy consumption on Jetson AGX Xavier with 2 video streams sending 10000 images when using YOLOv3. Fig. 8 (b) gives the average energy consumption per processed frame. Fig. 8 shows that `2 CUDA streams` and `pipelining` consume similar energy. From Fig. 8(a), we observe that `2 CUDA streams` and `pipelining` consume the most energy. This is because the high throughput of these two versions. Indeed, if we compare the energy consumption per frame, both `2 CUDA streams` and `pipelining` achieve the best energy efficiency, that is totally because they have lower FLR and handle more frames than others.

Faster R-CNN. Fig. 9 gives the throughput and latency of Faster R-CNN under different parallel strategies and camera numbers. `MPS` and `CUDA stream(pinned)` overlap in Fig. 9 (a) and (b). `Pipelining` and `serial` overlap in Fig. 9 (b). Like YOLOv3, `pageable/pinned` memory have similar performance impact on `MPS` and `pinned` memory is a better choice for `CUDA Stream` mode. `MPS` and `CUDA Stream(pinned)` parallel modes have similar performance. When using `MPS` and under 2 cameras, MPInfer can achieve 59.54 FPS throughput and 35.45 ms. MPInfer can only support 2 cameras when using Faster R-CNN, but it can support 5 cameras when using YOLOv3. This is due to the greater amount of computation in Faster R-CNN.

5 Conclusion

In this paper, we design and develop a multi-tenant parallel CNN inference framework, MPInfer, to support several proposed techniques and serve as a common platform for performance comparison. Our results show that `MPS` mode is both the better choice for YOLOv3 and Faster R-CNN. The future work is to explore more optimization strategies to support more cameras.

Acknowledgments. This work was partially funded by the National Major Program for Technological Innovation 2030—New Generation Artificial Intelligence (No. 2018AAA0100500) and the National Natural Science Foundation of China (No. 61772487).

References

1. Bateni, S., Wang, Z., Zhu, Y., Hu, Y., Liu, C.: Co-optimizing performance and memory footprint via integrated CPU/GPU memory management, an implementation on autonomous driving platform. In: RTAS'20. pp. 310–323 (2020)
2. Bodla, N., Singh, B., Chellappa, R., Davis, L.S.: Soft-NMS — improving object detection with one line of code. In: ICCV'17. pp. 5562–5570 (2017)
3. Goodwin, D.: NVIDIA TensorRT Inference Server boosts deep learning inference. https://devblogs.nvidia.com/nvidia-serves-deep-learning-inference/ (Sep 2018)
4. Hawkins, A.J.: Watch Mobileye's self-driving car drive through Jerusalem using only cameras. Online at https://www.theverge.com/2020/1/7/21055450/mobileye-self-driving-car-watch-camera-only-intel-jerusalem (Jan 2020)
5. Heng, L., Choi, B., Cui, Z., Geppert, M., Hu, S., Kuan, B., Liu, P., Nguyen, R., Yeo, Y.C., Geiger, A., Lee, G.H., Pollefeys, M., Sattler, T.: Project AutoVision: Localization and 3D scene perception for an autonomous vehicle with a multi-camera system. In: ICRA. pp. 4695–4702 (2019)
6. Jain, P., Mo, X., Jain, A., Subbaraj, H., Durrani, R.S., Tumanov, A., Gonzalez, J., Stoica, I.: Dynamic space-time scheduling for GPU inference. CoRR **abs/1901.00041** (2019), http://arxiv.org/abs/1901.00041
7. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: 22nd MM. pp. 675–678. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2647868.2654889, http://doi.acm.org/10.1145/2647868.2654889
8. Migacz, S.: 8-bit inference with TensorRT. In: GPU technology conference. vol. 2, p. 7 (2017)
9. NVIDIA: Multi-Process Service (vR440). Online at https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf (Nov 2019)
10. NVIDIA: TensorRT developer's guide (v7.0). Online at https://docs.nvidia.com/deeplearning/sdk/pdf/TensorRT-Developer-Guide.pdf (Dec 2019)
11. Redmon, J.: Darknet: Open source neural networks in C. Online at http://pjreddie.com/darknet/ (2013-2016)
12. Redmon, J., Farhadi, A.: YOLOv3: An incremental improvement arXiv:1804.02767 (2018)
13. Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: Towards real-time object detection with region proposal networks. In: 28th NeurIPS. pp. 91–99 (2015)
14. da Silva Carvalho, M.D., Koark, F., Rheinländer, C., Wehn, N.: Real-time image recognition system based on an embedded heterogeneous computer and deep convolutional neural networks for deployment in constrained environments. In: WCX SAE World Congress Experience. SAE International (Apr 2019). https://doi.org/https://doi.org/10.4271/2019-01-1045
15. Yang, M., Wang, S., Bakita, J., Vu, T., Smith, F.D., Anderson, J.H., Frahm, J.: Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In: RTAS'19. pp. 305–317 (April 2019). https://doi.org/10.1109/RTAS.2019.00033
16. Yang, M., Otterness, N., Amert, T., Bakita, J., Anderson, J.H., Smith, F.D.: Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In: Altmeyer, S. (ed.) 30th ECRTS. Leibniz International Proceedings in Informatics (LIPIcs), vol. 106, pp. 20:1–20:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.ECRTS.2018.20