S^3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads

Husheng Zhou, Soroush Bateni and Cong Liu
The University of Texas at Dallas
husheng.zhou@utdallas.edu, soroush@utdallas.edu, cong@utdallas.edu

Abstract. Deep Neural Networks (DNNs) are being widely applied in many advanced embedded systems that require autonomous decision making, e.g., autonomous driving and robotics. To handle resource-demanding DNN workloads, graphic processing units (GPUs) have been used as the main acceleration engine. Although much research has been conducted to algorithmically optimize the efficiency of applying DNN to applications such as object recognition, limited attention has been given to optimizing the execution of GPU-accelerated DNN workloads at the system level. In this paper, we propose S^3DNN, a system solution that optimizes the execution of DNN workloads on GPU in a real-time multi-tasking environment, which simultaneously optimizes the two (sometimes) conflicting goals of real-time correctness and throughput. S^3DNN contains a governor that selectively gathers system-wide DNN requests to perform smart data fusion, and a novel supervised streaming and scheduling framework that combines a deadline-aware scheduler with the concurrency-enabled CUDA stream technique. To simultaneously maximize concurrency-induced benefits and real-time performance, S^3DNN explores a rather interesting and unique characteristic of DNN workloads, where multiple layers of a DNN instance often exhibit a gradually decreased GPU resource utilization pattern. We have fully implemented S^3DNN in a GPU-accelerated system and have conducted extensive sets of experiments evaluating the efficacy of S^3DNN under a wide range of system and workload scenarios. The results show that S^3DNN significantly improves upon state-of-the-art GPU-accelerated DNN processing frameworks, e.g., up to 37% and over 40% improvements in real-time performance and throughput, respectively.

1 Introduction

Embedded systems are displaying a trend towards increased autonomy due to new application domains such as cyber-physical systems (CPS) and internet-of-things (IoT), which are emerging from the integration of embedded systems and the physical environment. Such systems are often equipped with multiple sensors to capture environmental information and process them in real-time to make correct autonomous control decisions. Examples include autonomous driving vehicles and smart robotics. To make intelligent decisions, deep neural networks (DNNs) are being widely adopted in many such systems. For example, for autonomous driving systems including several commercially available solutions, DNNs are used to map the raw pixels from on-vehicle cameras to the steering commands [1], [2]. This DNN-based approach is powerful because with limited training data from humans, the driving system can learn to drive by itself.

Work supported by NSF grants CNS 1527727 and CNS CAREER 1750263.

Adopting inherently computation-intensive DNNs in often resource-constrained embedded systems creates new challenges since adding sufficient DNN layers to guarantee high detection accuracy may easily explode the computation demand [3], [1], [4], [5]. Currently, embedded system designers mainly rely on using GPU-accelerated hardware platforms to satisfy the need, since GPUs enable orders of magnitude faster and more energy-efficient execution of many general-purpose workloads, hence the name GPGPU. For instance, Volvo has recently announced using the latest NVIDIA DRIVE PX2 computing engine to power a fleet of 100 Volvo XC90 SUVs starting to hit the road in 2017 [6], [7]. In addition to hardware acceleration, there are some research exploring the specific features of DNN workloads to improve the single-tasking throughput at the algorithmic level [8], [9], [10], [11]. However, to the best of our knowledge, there is a lack of research effort tackling these challenges from the critical system-level optimization perspective: how to optimize the execution of GPU-accelerated DNN workloads at the system level in a real-time multi-tasking environment. A critical objective is to guarantee real-time performance while maximizing system throughput and resource utilization to mitigate the inherent resource constraint imposed by most embedded hardware.

In this paper, we propose S^3DNN (Supervised Streaming and Scheduling for DNN)–a system solution that supports DNN-based real-time object detection workloads on GPU in a multi-tasking environment, while simultaneously improving real-time performance, throughput, and GPU resource utilization. To guarantee real-time performance, S^3DNN extends a classical deadline-aware real-time scheduler namely least-slack-first (LSF) to prioritize and schedule multiple DNN instances. To maximize throughput and GPU resource utilization, S^3DNN schedules workloads in the granularity of GPU kernels and dynamically aggregates kernels that under-utilize GPU resources to enable better concurrency. Moreover, in order to simultaneously maximize real-time and throughput performance, instead of simply combining LSF and kernel concurrency, S^3DNN develops a rather novel supervised streaming and scheduling framework that is motivated by our key observations on a unique characteristic of DNN workloads.

Specifically, we have conducted measurement-based case studies, from which we observe that DNN-based object detection workloads often exhibit a "staged" GPU resource utilization pattern. Intuitively, an object detection workload can be viewed as a set of dependent layers each of which takes a set of arrays as input and outputs a set of feature maps that will then be processed by the subsequent layers. Our observation is that such workloads often start with the most expensive layer in terms of GPU resource utilization, followed by subsequent layers exhibiting consistently gradual decrease

978-1-5386-5295-4/18/$31.00 ©2018 IEEE
DOI 10.1109/RTAS.2018.00028

in GPU resource utilization. Based on this unique workload characteristic, our insight is that instead of scheduling GPU kernel execution in a default unoptimized manner, it may be much more beneficial to perform supervised streaming and scheduling from a system-level point of view. S^3DNN is able to achieve this by employing a governor that is in charge of gathering system-wide DNN requests and judiciously fusing them together, and a scheduler that is in charge of maximizing concurrency benefits through considering varying resource requirements due to the staged utilization characteristic of DNN workloads.

We have fully implemented S^3DNN on top of a system with a multi-core CPU and multiple GPUs. S^3DNN is designed as a middleware between input videos and GPU hardware to optimize the execution of DNN-based object detection workloads on GPU. S^3DNN is implemented as a frontend-backend framework. Multiple frontends are in charge of forwarding all incoming video precessing requests to the backend, while the backend serves as a daemon process to aggregate requests and perform the actual computation. There are two major components in the S^3DNN backend. The first component is a governor that selectively fuses incoming video frames at the system level, consolidating frames into multiple DNN instances. The second component, namely the supervised streaming and scheduling module (S^3 scheduler), is developed to prioritize and schedule the assigned DNN instances to maximize both real-time and throughput performance through optimizing execution concurrency. Note that as a middleware solution, S^3DNN can be easily integrated with any existing DNN processing framework (e.g., Caffe [12]) with a small amount of code rewriting efforts.

We have conducted extensive sets of experiments evaluating the efficacy of S^3DNN under a wide range of system and workload scenarios. The results show that S^3DNN significantly improves upon state-of-the-art GPU-accelerated DNN processing frameworks in terms of both real-time and throughput performance. For example, compared to YOLO which is a state-of-the-art DNN object detection framework, S^3DNN improves system throughput by up to 18% in a single GPU system environment and up to 37% in a multi-GPU system environment. Regarding real-time performance, S^3DNN guarantees an almost 100% deadline meeting ratio under a reasonably light workload scenario. Under stressful scenarios with heavy workloads, S^3DNN outperforms the existing solutions by over 40% in terms of the number of frames that can be processed by their deadlines.

2 Background

In this paper, we focus on using GPGPU to accelerate DNN workloads in a system consisting of multiple discrete GPUs and a multi-core CPU. We use CUDA as the GPU programming model. In the following, we provide needed background on the CUDA programming model and DNN.

2.1 The CUDA programming model

GPGPU applications typically obey the following execution flow: (i) initializing the GPU device, (ii) allocating GPU device memory, (iii) transferring data from host memory to device memory, (iv) launching the computation work (kernel) on GPU, (v) copying results back to host memory, and (vi) freeing device memory and closing the device. We further highlight some terminologies that will be frequently used in the rest of this paper.

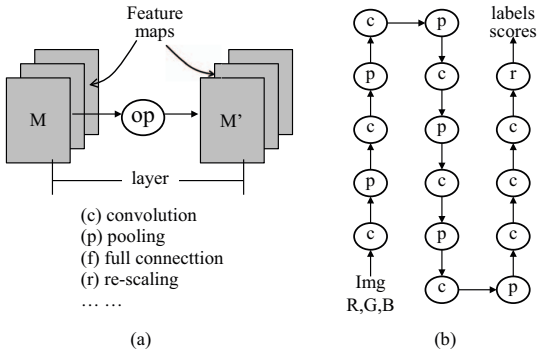


Fig. 1: (a) DNN layers are essentially array-based computations operated on lists of arrays often called feature maps. (b) A state-of-the-art DNN for object recognition, formed by connected layers.

Context: Context conceptually represents separate virtual address spaces on the GPU hardware. A context is either transparently or explicitly created for a CUDA application at the GPU device initialization stage. NVIDIA has provided the MPS (Multi-Process Service) feature in newer versions of CUDA to transparently merge multi-process CUDA applications into one context. However, the usage of MPS is limited by operating system (only supports Linux-based system), applications (only supports 64-bit applications), GPU hardware (compute capability 3.5 or higher), and special configuration (set GPU to be exclusive to other processes). Moreover, MPS fails to consider DNN-specific characteristics since it is an application-oblivious approach and will blindly combine all of the processes assigned to it. In this paper, we assume a more common scenario, where different CUDA applications run on different contexts.

SM: Streaming multiprocessor (SM or MP) is an internal unit of NVIDIA GPU hardware that performs the actual computation. Each GPU consists of several SMs, the number of which is product-specific. Low-end GPUs typically have one SM, and high-end GPUs have up to 24 SMs. For example, NVIDIA Quadro 6000 has 14 SMs.

Kernel, Thread, Thread block, Grid: In the CUDA programming model, a kernel is a piece of code executed on GPU hardware consisted of multiple CUDA threads executed in parallel. Thread is the atomic executing unit in CUDA programming. An array of threads stitch together to form a thread block or block for short. These blocks are further combined to form a grid. A grid represents the thread organization of a kernel launch. The dimension of block (blockDim.x, blockDim.y, and blockDim.z) and grid (gridDim.x and gridDim.y) are explicitly controlled by the programmer. Once a kernel is launched, its dimensions cannot change. During kernel execution, each block is assigned to an SM and is executed in its entirety. We note that each SM can be assigned multiple blocks.

CUDA stream: CUDA stream is a technique introduced by a newer version of CUDA [13], [14], which aims to hide the latency of memory copy and kernel launch from different

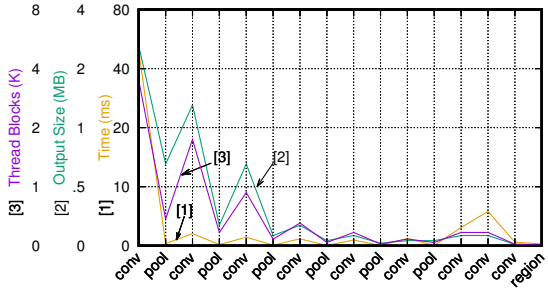


Fig. 2: Resource usage pattern of DNN workloads.

independent operations. Its additional effect is the capability of enabling concurrent kernel execution which allows multiple kernels to execute on the same GPU simultaneously when each kernel cannot fully utilize the entire GPU device. A CUDA stream can be explicitly created by the programmer and bound to a kernel launch or data copy operation. To avoid confusion with data stream/video stream, when we talk about this technique, we use the term CUDA stream.

2.2 Deep Neural Network (DNN)

A DNN can be viewed as a dataflow graph, in which its nodes, or layers, are essentially array-based computations (as shown in Fig. 1(a)) [8]. Each layer takes a set of arrays, called ***feature maps*** as input and outputs a set of feature maps that will in turn be processed by subsequent layers belonging to the same DNN instance. Fig. 1(b) shows an illustration of a set of DNN layers used in YOLO [8], which is a popular DNN framework aimed at real-time object detection. The letter within each cycle denotes the functionality performed by the corresponding layer: 'c' for convolution layer which convolves inputs by a convolution filter, 'p' for a max pooling layer which replaces each input array value by the maximum of its neighbors, 'f' for a fully connected layer which multiplies the feature maps by a weight matrix, and 'r' for a region layer, which is a layer specific to YOLO that is responsible for finding areas of interest in an image. In order to use a DNN for object detection, a pre-trained weight file is needed which is estimated from training data beforehand. Training an accurate weight file is an offline procedure that usually takes several days or weeks, done by "learning" features from large-scale image datasets. In this paper, we are not concerned with improving the training process, rather, we focus on efficient execution of DNNs for real-time object detection.

3 Motivation

In this section, we illustrate a set of measurements-based case studies to motivate our design. We use YOLO [8] as the target program, representing the state-of-the-art, real-time DNN-based object detection frameworks. YOLO is capable of taking continuous frames from a video file or a camera as input, and is able to output frames with labeled objects. We run multiple YOLO instances simultaneously on an NVIDIA Quadro 6000 GPU, each of which uses a road drive video from the KITTI vision benchmark suite [15] as input. We use average processing FPS (frames per second) of all processed frames to measure the throughput of the system. If the object recognition programs can process every frame of the video at a higher or equal FPS than the original processing time of the video (i.e., 40 ms per frame), we consider that the video can be processed in real-time. Besides average FPS, we also use deadline miss rate, denoted *pMiss*, to indicate the percentage of frames that miss the deadline. We note that, different from online cameras, pre-recorded videos can be processed at a higher FPS than the origin, since such videos are actually processed as a series of images – once the previous image has finished being processed, the next image is processed immediately afterwards without waiting for the release time. We also consider online cameras in the evaluation (Sec. 5).

3.1 GPU Usage Pattern For DNNs

In the first case study, we directly use YOLO to perform object detection on an input video. The neural network is configured to use the default setup (yolo.cfg), which is composed of 16 layers of three types depicted on the x-axis of Fig. 2 ("conv" for the convolution layer, "pool" for the max-pool layer, and "region" for the region layer). For each layer, the y-axis reports the size of the output which is temporarily stored in GPU global memory (in MB), number of thread blocks (in Kilo), and execution time (in ms) in processing each frame. Two key observations can be obtained from this figure: (1) The GPU resource utilization, including both the global memory usage and the thread block numbers, shows a "staged" pattern as the layers go deeper (i.e., consistent gradual decrease on resource utilization along with the increased depth of layers). (2) The final layers exhibit a small input data and light computation, thus may under-utilize the GPU resources. The intuition behind these observations is that DNN generates more fine-grained feature maps at earlier layers and prunes these details gradually along the depth increase to get a higher level of abstraction. In classic DNN models [16], [17], [18], they usually use fully connected layers at the last few layers to match the pattern generated from input images with all the possible classes stored in the model file. The input size of the fully connected layer is determined by the number of classes. In the case of real-time object detection in autonomous driving which is the focus of this paper, the class number is usually small since we are only interested in a small subset of objects (e.g., pedestrian, stop sign, vehicle, traffic signal). For example, the class number in our experiments is provided by vanilla YOLO an is 20 and 80, trained from the coco dataset [19] and voc [20] respectively. We note that we have observed similar trends on other popular DNN-based object detection tools [9], [10], [11] using their default configurations.

Insight 1: The GPU usage pattern in DNN shows a staged GPU resource utilization pattern, where the earlier layers involve more intensive computations and larger input sizes, and the later layers incur lighter computations and smaller input sizes, which may under-utilize GPU hardware.

3.2 Data Fusion

Fetching multiple images in a "batch" to process them in one pass is a common optimization used in DNNs, which can significantly improve the throughput with a proper batch size [16], [18], [17]. The improvement is due to the fact that using batch can reduce the time wasted in I/O operations and data communication round-trips, and thus can improve performance compared to processing images sequentially. Moreover, in some cases, it is able to improve GPU utilization.

3.2.1 Application Level Data Fusion: Many prominent DNN implementations such as Caffe [12] include a data fusion functionality. For a multi-tasking environment where multiple DNN instances are running simultaneously, it is possible to batch multiple images into one instance instead in order to improve overall throughput. This approach is particularly useful under the autonomous driving scenario, where a vehicle is often equipped with multiple cameras and sensors to cover front, sides, and rear view. In this case study, we illustrate the effect of data fusion on throughput and latency.

We modified YOLO to support batch-based processing of up to four videos in order to process them all at once. The baseline uses up to four vanilla YOLO programs, each of which processes one input video. These videos are of the same target FPS (25 FPS). We use average FPS to measure the throughput in two different configurations, baseline without fusion and our implementation with data fusion which fuses all input frames together. To measure the real-time performance, we use *pMiss* to record the percentage of frames that miss their deadline. The results are shown in Table I, where the first row indicates the number of incoming video streams, the second and third rows show the average FPS, and the fourth and fifth rows record the *pMiss*.

We observe that, with more incoming videos, data fusion becomes more efficient in improving throughput, as indicated by the average FPS. Another observation is that data fusion causes a 100% pMiss ratio when fusing four video streams together, yet achieving a high FPS (21.2×4). This is because, with data fusion, all the fused images will start and complete at the same time with a longer processing time compared to the individual execution scenario. In this case study, when fusing four video streams together, the resulting processing time exceeds the deadline of each frame, thus incurring a 100% pMiss ratio. In practice, multiple cameras on an autonomous driving vehicle may be of different FPS. For example, cameras on the dashboard may have higher FPS than cameras at the rear end. In such scenarios, fusing data from cameras with different frequency may lead to more frequent deadline misses for cameras with higher FPS.

3.2.2 System Level Solution: While existing application level data fusion has proven effective in increasing throughput, it falls short on multiple accounts. First and foremost, for a multitasking environment consisting of multiple DNNs (such as in autonomous driving), application level data fusion will not be effective since it would not be aware of other DNN processes in the system. Moreover, in such systems the number of executing DNN instances varies from time to time. Since application level batching needs to be pre-configured, it would not be able to adjust. For example, in Table I, the data fusion FPS can vary significantly depending on the number of videos and can cause many variations that a real-time system needs to deal with. Finally, an application level solution would not be able to recognize data priority and take scheduling decisions into consideration, which might result in lower-priority tasks to be batched with higher-priority tasks, causing an unfair and potentially dangerous situation. This is evident in the 100% miss rate visible in the 4 video data fusion as shown in Table I.

Insight 2: To overcome the under-utilization issue incurred by individual DNN workloads, fusing data from multiple videos may achieve significant throughput gain yet may harm the per-frame response time for each individual video.

TABLE I: APT and pMiss of data fusion and base line

# of videos	1	2	3	4
Baseline FPS per video	52.6	29.2	18.5	15.6
Data fusion FPS per video	52.6	33.3	26.7	21.2
Baseline *pMiss*	0%	5%	83%	92%
Data fusion *pMiss*	0%	0%	0%	100%

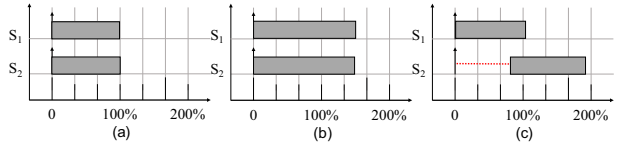


Fig. 3: Execution time of two streamed concurrent Kernels with different numbers of thread blocks: (a) small number (b) medium number (c) large number.

3.3 Kernel Scheduling and Concurrency

Besides data fusion, as mentioned in Sec. 2, scheduling tasks in an orderly fashion while utilizing the CUDA stream technique may also improve throughput by enabling concurrent execution of multiple kernels bound to different streams while ensuring deadlines, as illustrated by the following case study.

3.3.1 Enabling Concurrency using CUDA Streams: To understand how GPU performs concurrency for kernels in different streams, we conduct three case studies, each of which performs matrix multiplication-based computation that is frequently used in DNN. In all case studies, we concurrently launch two kernels placed in two different CUDA streams. The only difference among three case studies is the total number of thread blocks contained in each kernel. The execution traces are shown in Fig. 3, where case studies (a), (b), and (c) have 7, 28, and 500 thread blocks in each kernel, respectively. The x-axis represents the normalized execution time defined by the actual execution time divided by the execution time of executing only one of the two kernels under each scenario.

We observe in Fig. 3(a) that the response time of completing two kernels is the same as only one kernel being executed, thus enabling a perfect concurrency for this scenario. This is because the total number of thread blocks contained in these two kernels equal the number of SMs in GPU hardware (i.e., 14 thread blocks or SMs). Thus, each thread block is assigned to a dedicated SM. In Fig. 3(b), the observation is that the response time becomes longer than the standalone execution time of one kernel. In this case, although the total number of thread blocks (56 in this case) is more than the number of SMs, concurrency can still be enabled because NVIDIA compute capability 2.x devices can support up to 8 blocks per SM. However, due to the large number of thread blocks, these thread blocks from two kernels will be executed in an interleaving manner, causing a performance better than serialized execution but worse than the perfect concurrent execution case (e.g., Fig. 3(a)). For the third case study where we aggressively increase the total number of thread blocks to 1000, as seen in Fig. 3(c), we observe that these two kernels can only partially overlap to a rather limited degree of concurrent execution. The reason is because with a

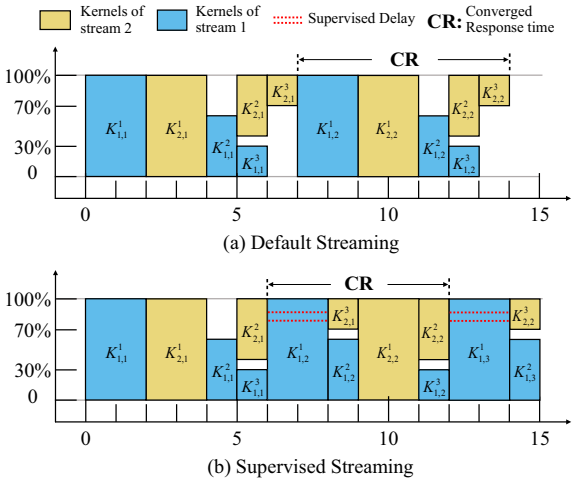


Fig. 4: Concurrency under CUDA stream without supervised streaming (inset (a)) and with supervised streaming (inset (b)).

large number of thread blocks per kernel, the first kernel will fully occupy all GPU resources when it starts execution. Near the end of the first kernel's execution, there are a remaining 52 thread blocks (i.e., 52 = 500 thread blocks % 14x8 GPU capacity) to be executed on GPU, the second kernel can start execution concurrently using the available SMs on GPU.

Insight 3: Concurrent kernel execution through putting each kernel in a different CUDA stream may improve overall system throughput performance, particularly when individual kernels cannot fully utilize GPU resources. However, adopting CUDA stream to improve concurrency may harm the timing predictability of individual kernels due to interference.

3.3.2 Supervised CUDA Streams with Scheduling: Based upon Insight 3, we perform another case study to understand how DNN workloads may further benefit from stream-enabled concurrency. We run two identical periodic DNN tasks, each consisting of three layers. To enable concurrency, we bind each task to a separate CUDA stream. We configure the first, second and third layer of each DNN task to occupy 100%, 60% and 30% of the GPU resource, i.e., 224, 67 and 34 respectively, thread blocks according to the used GPU hardware. Fig. 4(a) shows the execution schedule constructed from the measurements data when we run two streams concurrently. Let $K^k_{i,j}$ denote the k^{th} layer/kernel of the j^{th} job (i.e., the j^{th} video frame) released by DNN task K_i. As seen in Fig. 4(a), streaming indeed improves performance by enabling concurrent execution. For example, $K^2_{2,1}$ is concurrently executed with $K^3_{1,1}$ since the total number of thread blocks of these two kernels is smaller than the available thread blocks supported by the GPU hardware.

While we are seeking to further improve concurrency, a very interesting observation appears. As seen in Fig. 4(a), $K^3_{2,1}$ is executed alone without other concurrent kernels. However, performance may be improved by concurrently executing $K^3_{2,1}$ with a later released kernel $K^2_{1,2}$. To achieve this, we have to let $K^1_{1,2}$ execute first and preempt $K^3_{2,1}$, thus releasing $K^2_{1,2}$ upon its completion. The resulting execution schedule is shown in Fig. 4(b), where clearly both response time performance and concurrency are improved.

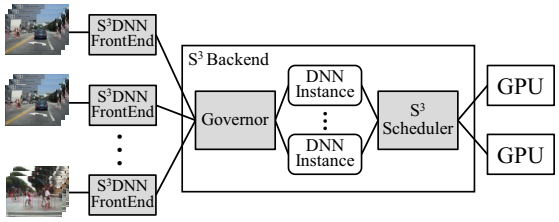


Fig. 5: Design overview of S^3DNN.

Insight 4: To optimize concurrency for DNN workloads exhibiting staged computation demand, instead of performing default CUDA streaming, it may be much more beneficial to perform "supervised streaming", which judiciously schedules kernels for maximum concurrency benefits through considering varying resource requirements of different layers of DNN tasks.

4 Design and Implementation of S^3DNN

4.1 Design Overview

S^3DNN is designed to serve as a middle-ware between input videos and GPU hardware to optimize the execution of DNN-based object detection workloads on GPU. S^3DNN is implemented as a frontend-backend framework, where multiple S^3DNN frontends take videos as input and forward all data and DNN processing requests to the S^3DNN backend for actual computation, as shown in Fig. 5.

The S^3DNN backend consists of two major components, a *Governor* (Sec. 4.2) and an S^3 *scheduler* (Sec. 4.3), motivated by the four insights discussed in Sec. 3. Whenever multiple video frame processing requests are forwarded to the backend, the governor will decide to selectively fuse them together so that they can be processed in fewer *DNN instances*. The governor also assigns the fused DNN instances to one or several virtual contexts, each of which is associated with a GPU device according to the computing capability (i.e., GFlops) and memory capacity (i.e., GPU global memory size). Since GPU memory is a major constrained resource, S^3DNN leverages a classical bin-packing algorithm to dispatch videos by considering GPU memory capacity as the bin size. The S^3 scheduler is then in charge of scheduling the DNNs (virtual contexts) at the granularity of GPU kernel. In order to meet real-time constraints, S^3 scheduler uses a kernel-level least-slack-first (LSF) scheduling policy, which prioritizes kernels considering both their deadlines and subsequent kernels belonging to the same DNN instance. Note that in our current implementation, S^3DNN uses YOLO [8] as the DNN engine which represents the fundamental DNN processing framework. However, as a general middleware solution, S^3DNN can be easily plugged into other existing DNN frameworks, requiring minor rewriting work.

Before diving into the detailed design of each component, we present an abstract definition for the system. Namely, our system is constructed from a set of frame processing requests, $\tau = \tau_1, ..., \tau_n$, in which τ_i is eventually assigned to a DNN instance to be processed (this DNN instance can be shared by multiple τ_i). For the sake of simplicity, we assume each layer

contains only a single GPU kernel. If multiple kernels are used in a layer, we can combine them into a single GPU execution to achieve the same goal. We use this model throughout the paper as a platform for any real-time analysis. In the case of data fusion, the DNN structure and algorithms function the same way, albeit with a larger data input.

4.2 System-level Data Fusion

As suggested by Insight 2, performing system-level data fusion upon DNN workloads may effectively improve performance in terms of throughput. Thus, S^3DNN implements a governor in the frontend-backend framework to selectively fuse incoming video frames at a system-wide level, by considering both throughput improvements and potential response time increases as executing fused frames may lengthen the execution time of individual frames. The goal is to conduct data fusion in a way such that real-time constraint is satisfied while throughput can be maximized.

Data fusion is possible due to an exploitation of matrix operations inside a DNN. For a normal DNN, most operations are matrix-based, following a format similar to the following:

$$R_{M*N} * F_{N*K} = M_{M*K}, \tag{1}$$

in which F is called filter, while R is the input for that layer and M is the output. Fusing α matrices together will turn Eq.1 into:

$$R_{(\alpha*M)*N} * F_{N*K} = M_{(\alpha*M)*K}. \tag{2}$$

We consider the problem of efficiently fusing n videos with at most m different FPS targets into q DNN instances, each of which corresponds to a CUDA stream, where $m \leq q \leq n$. Each video is composed of a series of continuous frames that are processed sequentially. We define (e_k, d_k) to characterize each job in a DNN instance S_k, where e_k denotes the job's execution time on a GPU and d_i denotes the job's deadline (i.e., $1/f_i$ of the corresponding video where f_i denotes its target FPS). The design goal is to map τ to S. Before describing our developed data fusion algorithm, there are several constraints we will apply. First of all, data fusion shall be performed only when the system observes multiple videos simultaneously. In other words, we never wait for more video streams to arrive in order to enable data fusion. Clearly, waiting for multiple frames to arrive and then fusing them together will significantly cause real-time requirements of an already arrived video to be violated. Moreover, in order to simplify the design complexity, only videos with the same DNN configuration (*i.e.*, layers, weights, thus with the same e_i) will be fused into one DNN instance since they exhibit the same intensity of computation.

In addition to performing data fusion, the governor also needs to ensure that the total utilization of each DNN instance after fusion, denoted by U_k, is no greater than 1, where $U_k = \sum_{i=1}^{n_k} (\frac{e_i(S_k)}{d_i(S_k)})$ and $e_i(S_k)$ ($d_i(S_k)$) denotes the execution time and deadline, respectively, of the i^{th} fused video in the k^{th} DNN instance; Otherwise, frame deadlines may be frequently missed which negatively impacts the real-time correctness. The pseudo-code of this algorithm is given in Algorithm 1.

A very interesting observation drawn while designing Algorithm 1 is that the total utilization of the system will not necessarily equal to the sum of the individual utilizations $\sum_{\tau_1}^{\tau_n}(U_k)$.

Algorithm 1 Data Fusion Algorithm

```
Require: τ[] of all n inputs
Require: Instances[] of all DNN instances
1: function FUSE(InputList,Instances)
2:     S[]=∅
3:     for i in τ do
4:         for j in S do
5:             if i.d == j[0].d then
6:                 if LookUpHistory(j,i) < 1 then
7:                     j.insert(i); EXIT()
8:     Instances = CreateDNNInstance(S.length, FuseData(S))
```

This phenomenon is due to the inherent massive parallelism in GPUs, which can lead to the total execution time of two combined kernels to be less than the sum of the execution times of each kernel due to better resource utilization. In order to mitigate this problem, Algorithm 1 uses a history-based approach to figure out real-world combined execution times instead of relying solely on individually measured execution times.

Algorithm description. As seen in Algorithm 1, our data fusion algorithm takes the task set τ as input in the form of a list of m deadlines of n frames. It will then go through all combinations of τ (input frames) and S (the current DNN instances) in lines 3 and 4. As was mentioned earlier, two tests need to pass. First and foremost, the configuration should match. Thus, Algorithm 1 will check if the deadline of the current task matches the deadline of the first task in a DNN instance (line 5). After that, the algorithm checks to see if adding this new task will violate the utilization test (line 6). This is done through the $LookUpHistory()$ function. This function takes the existing tasks in a DNN instance along with a new task as input. It will then probe the history table to see if any history regarding this combination of tasks exists. If not, it will simply add up the individual utilizations to calculate the upper bound. If no such DNN instance is found, the algorithm will create a new one (line 8).

Memory-constrained dispatching. After the governor is finished fusing incoming videos, it will dispatch the fused video streams to different GPUs. Since GPU memory capacity is the major resource bottleneck herein, we transform this problem into a classical bin-packing problem and apply existing methods for dispatching. Specifically, the GPU memory size can be viewed as the bin size, while the memory required by each fused video stream is viewed as the item size. The problem is to pack all the fused video streams into GPUs such that the number of required GPUs is minimized. We thus leverage the classical first-fit algorithm to resolve this problem, which has been proven effective in many cases [21].

Optimizations in the governor. S^3DNN uses history-based prediction for each kernel launch by storing its execution time information together with system information (input size, other kernels, etc), as well as a hash value representing this kernel, in a lookup table. In addition to our previously mentioned observation, this approach is also based on a realistic assumption that the kernels used in a given DNN are fixed, and the critical variable that impacts execution time is the input data size as well as other kernels. This lookup table is saved to a file and loaded into memory when S^3DNN boots up.

Our implementation of S^3DNN optimizes memory usage. Specifically, instead of constructing a copy of a separate DNN

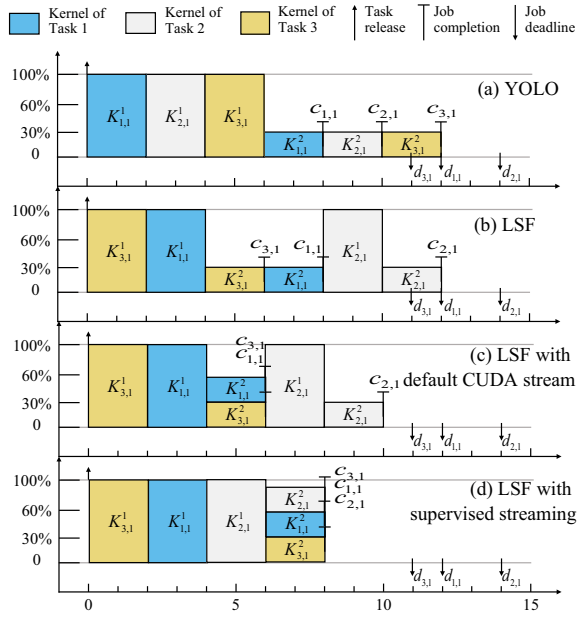


Fig. 6: Comparison of four scheduling policies.

instance (which includes the model and input/output data) for each input video, S^3DNN optimizes memory usage by sharing model data among multiple DNN instances in GPU global memory. In practice, since all input video streams may use the same DNN configuration for specific purposes (e.g., object recognition with the same accuracy goal), S^3DNN would only need to store one copy of the model data in each GPU's device memory, shared by multiple DNN instances.

4.3 Supervised Streaming and Scheduling

As motivated by Insights 3 and 4 discussed in Sec. 3.3.2, S^3DNN seeks to optimize the execution of fused video streams assigned to each GPU through supervised streaming and scheduling, with the goal of maximizing concurrency (thus throughput and GPU utilization) and real-time performance. With the help of the "per-thread" streaming option in CUDA 7 or newer versions, streaming-enabled concurrency becomes even easier to achieve, because the hardware scheduler inside the GPU computing engine takes care of the throughput optimization. Unfortunately, when kernels are submitted to GPU hardware by the driver, they do not have any priorities, causing the kernels to be executed in a FIFO order. This may jeopardize another important real-time performance indicator for DNN-based real-time object recognition workloads: deadline meeting ratio. Motivated by this, S^3DNN develops a GPU scheduling algorithm incorporating several novel ideas to simultaneously achieve these two (sometimes) conflicting goals. Specifically, this scheduler extends a classical real-time scheduler least-slack-first (LSF) to improve real-time performance, combined with supervised streaming via a lookahead approach for maximizing concurrency benefits. We choose to use kernel-level LSF because LSF, as a dynamic priority scheduler, can make kernels from different DNN instances be assigned different priorities and execute in an interleaving manner. The slack of a kernel at time t is defined to be the deadline of the corresponding job of the DNN task minus t, and then minus the total remaining amount of execution of

Algorithm 2 Supervised streaming and scheduling algorithm

```
Require: Q   ▷ Queue of kernels to be scheduled
Require: G   ▷ Group of kernels that can execute concurrently
Require: C   ▷ Critical queue that will be submitted to GPU
function ENQUEUE(Q[],k)
    for q in Q do
        UpdateSlacks(Q)
        if Slack(k) > Slack(q) then
            Q.InsertBefore(k, q)
function DEQUEUE(Q[])
    UpdateSlacks(Q)
    if G ≠ ∅ then
        go to assign
    h ← HeadOf(Q)
    G.insert(h)
    if tbRatio(h) < 1 then
        for t in Q-h do
            if tbRatio(t) < 1 then G.insert(t)
            if tbRatio(G) > 1 then break
    else go to assign
    if tbRatio(G) < 1 then
        h' ← HeadOf(Q)
        p ← LookAhead(h')
        if tbRatio(p) < 1 then
            G.reserve(p); C.insert(h')
            go to submit
        else go to assign
    C ← G; G ← ∅
    Submit(C); Q.Remove(C)
```

this task's kernels in the current job period. Static priority schedulers such as RM and other dynamic priority schedulers such as EDF will always assign the same priority to kernels belonging to the same DNN instance, preventing concurrency to be implemented efficiently. Regarding the concern of runtime overhead, LSF is similar to EDF in this case. This is because each kernel execution on GPU is non-preemptive, implying that the slack value of a DNN task (used to decide the priority of a pending kernel) only needs to be updated when any of its kernels complete execution on GPU. We first use the following example to illustrate the fundamental ideas behind S^3DNN's scheduler.

Consider three DNN tasks K_1, K_2, K_3, each of which contains two layers. The first layer utilizes 100% of the thread block resource in a GPU, and the second layer utilizes 30% of the thread block resource (following the same pattern observed in Sec. 3). K_1, K_2, and K_3 have a deadline of 12, 14, and 11 time units respectively. Fig. 6 illustrates four possible schedules under four different scheduling and streaming methods, where the y-axis represents the percentage of the thread block resource required by a kernel.

Fig. 6(a) shows the FIFO schedule for isolated processes which is the default behavior used in the YOLO framework, which causes a deadline miss for K_3 due to serialized execution and deadline-oblivious prioritization under FIFO. Fig. 6(b) shows the schedule under a kernel-level non-preemptive LSF scheduling algorithm, which prioritizes kernels according to least-slack-first. A kernel $K^k_{i,j}$'s slack is defined to be $r^1_{i,j} + d_{i,j} - r^k_{i,j} - F(r^k_{i,j})$, where $r^k_{i,j}$ denotes the release time of the k^{th} kernel of the j^{th} job of the i^{th} DNN instance, and $F(r^k_{i,j})$ denotes the amount of computation completed by kernel $K^k_{i,j}$ at time $r^k_{i,j}$. Intuitively, the slack denotes the number of time units

a DNN task can use to complete the remaining computation of the corresponding released kernel. Note that $K_{i,j}^{k}$ is released at the time when $K_{i,j}^{k-1}$ (if any) completes. Thus, under the kernel-level LSF, a kernel's priority is defined using LSF when it is released. As seen in Fig. 6(b), the kernel-level LSF is able to meet all tasks' deadlines, yielding an end-to-end response time of 12 time units. Note that LSF will not incur much overhead at runtime since priority definition is determined only at kernel boundaries, and the maximum number of kernels released in the scheduling queue equals to the number of CUDA streams which is often small, for example, less than 6 in our evaluation.

Although applying kernel-level LSF improves real-time performance, it does not benefit from concurrency. Thus, we illustrate in Fig. 6(c) a schedule under the kernel-level LSF scheduler combined with default CUDA streaming (i.e., put each DNN task into a separate stream and run the three streams concurrently). As seen in the figure, since K_1^2 and K_3^2 only use a total amount of 60% of the thread block resource, these two kernels can execute concurrently at time 4. Doing so further improves the end-to-end response time to be 10 time units.

An interesting observation obtained from Fig. 6(c) is that if K_1^2 and K_3^2 can be supervised to wait for the release of K_2^2, then these three kernels can actually be executed concurrently, which results in a further response time performance improvement as well as a better overall GPU utilization and throughput, as illustrated in Fig. 6(d). As seen in this figure, a supervised delay happens at time 4, where the scheduler chooses to run a lower-priority task K_2^1 first, thus allowing those three kernels to run together. This key observation motivates our design of S^3DNN's GPU scheduler. Since DNN tasks fundamentally exhibit a staged computation pattern where later layers often require fewer resources (i.e., Insight 1), it is more likely that later kernels belonging to different DNN tasks can be executed concurrently. However, since a kernel is released and pushed into the scheduling queue only if its predecessor kernel completes, at each scheduling instance (i.e., when a current running kernel completes on GPU), our scheduler takes a "lookahead" approach to find whether the highest-priority kernel in the scheduling queue can run concurrently with any later released kernels, as well as any kernels already waiting in the scheduling queue. This idea is illustrated by the example shown in Fig. 6(d), where at time 4, the scheduler lookaheads and finds out that the highest-priority kernel in the queue at time 6, which is K_3^2, can concurrently run with a later released kernel K_2^2 and an already released kernel K_1^2.

Algorithm description. Motivated by the above-discussed ideas, we now present our developed GPU scheduler for S^3DNN. We use a metric, *tbRatio*, to measure the proportion of the demanded thread blocks by a kernel to the total number of thread blocks provided by the GPU hardware (often constrained by either the hardware architecture or register/shared memory size). For example, GPUs with compute capability 2.x can support up to eight blocks per SM, if register/shared memory size is not the bottleneck. The pseudo-code of the algorithm is given in Algorithm 2. There are two major functions defined in Algorithm 2: Enqueue (Line 1) and Dequeue (Line 6). The algorithm needs three input data structures, including a scheduling queue (denoted by Q), any subset of kernels that can execute concurrently (denoted by G), and the subset of kernels with the highest priority in the scheduling queue (denoted by C). Function Enqueue is invoked at kernel arrivals and completions, which sorts the released kernels in Q using LSF (Lines 2-5). It first updates the remaining slacks for each kernel (line 3), and then inserts each newly arrived kernel k into Q according to LSF (Lines 4-5). Instead of invoking Dequeue immediately after a kernel's (e.g. k_0's) completion, S^3DNN delays this invocation a little bit until k_0's successor kernel is pushed into the queue. It first updates slacks for kernels in the scheduling queue (Line 7), and then checks if G is empty (Line 8). If G is not empty, then the scheduler directly submits kernels within G for execution (Line 9). Next, the kernel h at the head of the scheduling queue is pushed into G (Lines 10-11). Then the scheduler checks whether h can fully occupy the GPU by calculating its *tbRatio*. If *tbRatio* of h is smaller than 1 (Line 12), indicating it may be concurrently executed with other kernels, then the scheduler will seek to put more kernels in Q whose *tbRatio* is also less than 1 (Lines 13-15); else h is directly submitted to GPU device for execution (Line 16). After all potential small kernels in Q are merged into G, the scheduler checks whether the *tbRatio* of G is still less than 1 (Line 17). If so, the scheduler looks ahead the successor kernels of the ones residing in Q in the order of priorities, in order to identify any such kernels that have not been released but with $tbRatio < 1$ (Lines 18-22). The looking ahead operation can be achieved because S^3DNN builds dependency graphs (linear in YOLO) for all kernels at DNN instance construction. The purpose of doing this lookahead method is to check whether it is possible to have a maximum set of kernels that can run concurrently. If such a successor kernel exists, then the scheduler will supervise the set of kernels in G to wait for the release of this successor kernel. Considering the potential overhead caused by this lookahead method in practice, we limit the lookahead degree to one, which implies that the scheduler will only check the subsequent kernel released by the next highest-priority kernel in the scheduling queue (not counting the kernels already placed in G since they need to complete first in order to release subsequent kernels). Finally, kernels placed in G will be sent to C, which will be further submitted to GPU for execution (Lines 24-25).

TABLE II: Configuration of video numbers and FPS

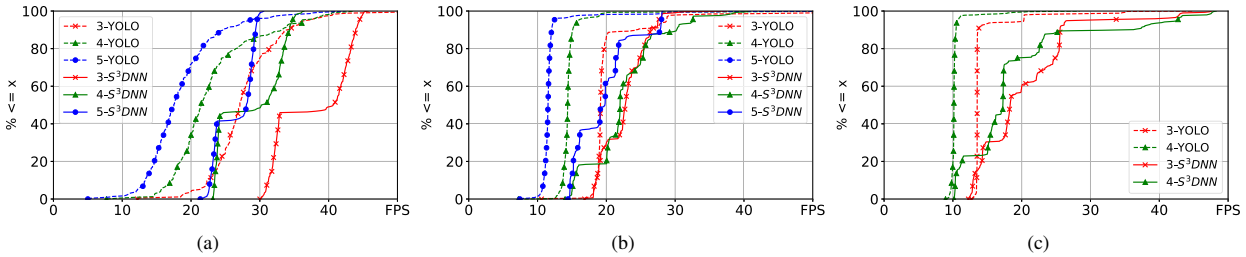
Config	light			medium			heavy	
FPS	3	4	5	3	4	5	3	4
10 FPS	n/a	n/a	n/a	n/a	n/a	n/a	1	2
15 FPS	n/a	n/a	n/a	1	2	2	2	2
20 FPS	1	2	2	2	2	3	n/a	n/a
25 FPS	2	2	3	n/a	n/a	n/a	n/a	n/a

In summary, S^3DNN's scheduler is designed to improve real-time performance through adopting a deadline-aware LSF algorithm while simultaneously maximizing concurrency through supervised streaming. We have included a rather intuitive example illustrating this algorithm in an appendix.

5 Evaluation

5.1 Experiment Setup

We conduct our experiments in a system consisted of an NVIDIA Quadro 6000 GPU, which is based on the Fermi micro-architecture and features 480 cores and 6 GB of GDDR5 memory, and an Intel Core i7-4790k CPU and 16 GB of



(a) (b) (c)

Fig. 7: CDF of FPS under (a) light (b) medium (c) customized DNN configurations. "3-yolo" ("3-S^3DNN") represents the FPS performance under YOLO (S^3DNN) when there are three input vidoes.

RAM. We use Ubuntu 14.04 based on Linux 3.5.7 as the underlying operating system. We compare S^3DNN to the YOLO framework as the baseline which is seeing widespread use in both academia and industrial systems for processing DNN workloads [8]. In this section, we will evaluate both real-time and throughput performance in terms of end-to-end response time and FPS under multi-tasking scenarios. We first conduct extensive experiments using video streams stored on the disk drive, and then a case study using live video streams captured in real-time by using on-dash mounted cameras. The frame frequency of a video as well as the deadline of each frame is determined by the videos' FPS. We use videos with 4 different FPS: 10, 15, 20, and 25. We evaluate three DNN configurations: one is configured with a shallow network, small weight file, and fast kernels, representing a light configuration; the second one is configured with a deep network, large weight file, and fast kernels, representing a medium configuration; the third one is configured with a deep network, large weight file, and slow kernels, representing a heavy configuration. These three configurations cover the two major aspects that affect the execution: execution time of each layer (fast or slow kernels) and number of layers (shallow or deep DNN layout).

5.2 Real-time performance

We first evaluate the real-time performance of S^3DNN by measuring the cumulative distribution function (CDF) of FPS under nine scenarios combining three DNN configurations with three cases of different input videos (3, 4, and 5 videos, respectively). For each experiment, we use a mixed set of video streams with different FPS, as shown in Table II. The second row indicates three workloads configurations. The first column uses FPS to indicate the deadlines of videos. Columns 2 to 9 show the number of concurrent videos and their composition. For example, the second column shows that there are three concurrent incoming videos in total, in which one is 20 FPS and two are 25 FPS.

Fig. 7 shows the evaluation results, where the x-axis represents FPS and the y-axis represents CDF. We observe that when the number of videos is more than one, S^3DNN outperforms YOLO by a significant margin. For example, with three input videos and light DNN configuration, S^3DNN is able to achieve an FPS higher than 30 in almost all cases, while 80% of the frames have an FPS lower than 30 under YOLO. Moreover, in many cases, S^3DNN is able to provide a higher FPS than the original FPS of the input videos. This implies that under S^3DNN, most of the video frames can be completed before their deadlines (note again that a frame's deadline is defined as its release time plus 1/FPS), thus meeting the real-time processing constraint. On the other hand, YOLO yields rather low FPS performance in many cases, particularly when the workloads are heavy and/or DNN configuration becomes heavy. Also note that as seen in Fig. 7(c), when the DNN configuration is heavy, we cannot get results from either YOLO or S^3DNN because the workload significantly over-utilizes the available hardware resource. Generally speaking, as is seen in the three insets of Fig. 7, with increased number of videos, and/or heavier DNN configurations, the overall performance under both YOLO and S^3DNN decreases. This is intuitive because more workloads will cause heavier contention on GPU. However, even under the scenario with the heaviest workload (i.e., heavy DNN configuration with 4 input videos as seen in Fig. 7(c)), S^3DNN is still able to provide reasonably good performance, where CDF of 15 FPS or above is greater than 70%; while YOLO yields a performance below 10 FPS for almost every frame in this case. Thus, our design of S^3DNN proves to be much more effective in delivering real-time performance through performing data fusion and supervised streaming and scheduling.

5.3 Overall Throughput

In this set of experiments, we mainly evaluate the throughput under S^3DNN compared to YOLO. For the three DNN configurations, we use multiple 25-FPS one-minute-long videos as input. Since we use offline videos as inputs, once one frame is processed, the next frame can be immediately processed afterwards. The throughput is shown in the left sub-figure of Fig. 8, where the x-axis is the number of videos processed simultaneously, the y-axis is the normalized throughput (i.e., the ratio of the throughput under S^3DNN divided by the throughput under YOLO). The four histograms shown under each case represent YOLO and S^3DNN under three different DNN configurations. We observe that S^3DNN outperforms YOLO when there are multiple input videos, fundamentally due to the fact that S^3DNN enables concurrency through supervised streaming. When there is only one video in the system, YOLO actually yields a slightly better performance than S^3DNN. This is because of the runtime overhead introduced by S^3DNN, which is reasonably small (less than 4% for all three DNN configurations). Another interesting observation is that S^3DNN yields the biggest performance improvement when there are two or three videos in the system with a heavy DNN configuration. This is because (*i*) the heavy DNN configuration implies heavier workloads that may benefit more from concurrency due to supervised streaming and scheduling (as discussed earlier), and (*ii*) data fusion becomes particularly effective in such cases because it is possible to fuse two or

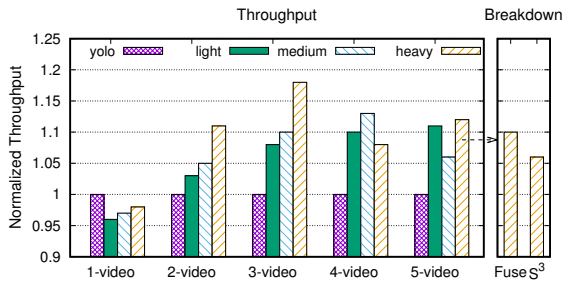


Fig. 8: Normalized throughput under light, medium, and heavy workloads using a variant number of videos.

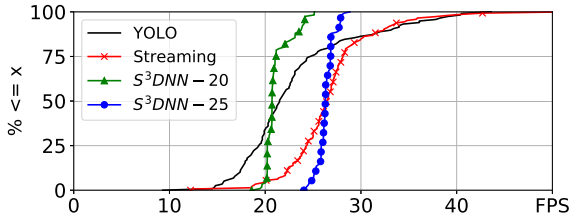


Fig. 9: Efficacy w.r.t. FPS under S^3 compared to isolated run and default CUDA streaming

three videos into a single DNN instance. Processing a single large matrix-based computation can be more efficient than separately processing several smaller matrix-based computations. For cases with four or five video streams, the throughput under S^3DNN decreases because four videos cannot be fused into a single DNN instance due to the fact that such data fusion would cause the processing time of the resulting fused frame to be definitely longer than the deadline of each individual frame.

The right sub-figure of Fig. 8 demonstrates the breakdown of the histogram representing five heavy workloads scenario. The histogram labeled "Fuse" ("S^3") indicates the throughput when S^3DNN only enables the functionality of data fusion (S^3 scheduler). We observe that data fusion brings the majority of throughput improvement (10%). S^3 scheduler still brings 6% throughput improvement though it is designed for meeting real-time constraints. The combination of both brings 12% improvement (as shown in the left sub-figure) which is not linearly added-up of two components' improvements. This is because when we use each standalone technique, the optimizations realized in the frontend-backend framework (i.e., model sharing, GPU context consolidation) benefit both scenarios.

5.4 Assessing the Supervised Streaming and Scheduling Module

As we believe the supervised streaming and scheduling module contributes to both real-time performance and throughput, we conducted a set of experiments specifically evaluating the efficacy of this module. As discussed in Sec. 4.3, we compare this module against two other methods, including YOLO and concurrency-enabled CUDA streaming (i.e., binding multiple DNN instances to separate CUDA streams and running them concurrently by the GPU hardware scheduler). In these experiments, we use the light DNN configuration and four input videos, two 25-FPS videos and two 20-FPS videos. We use the CDF of FPS as the evaluation metric.

Fig. 9 shows the experimental results, where the performance under S^3DNN is separated into two curves, corresponding to the two 20-FPS videos and the two 25-FPS videos, respectively. For the other two approaches, we do not separate the results according to FPS. This is because both these approaches are application-oblivious. Thus, the resulting performance for each of the four videos is almost identical, even though they have different FPS. For depiction clarity, we just draw one curve that represents the overall performance for processing all four videos under both approaches. Note that since S^3DNN schedules tasks considering their deadlines (thus FPS), the resulting performance under each FPS category becomes distinguishable. We observe in Fig. 9 that both CUDA streaming and our supervised streaming methods improve the performance compared to YOLO. Compared to CUDA streaming, the FPS performance under our supervised approach is clearly more predictable and consistent. For example, under CUDA streaming, over 30% frames of the two 25-FPS videos miss their deadlines (i.e., with a $<$ 25 FPS); while under S^3DNN, only 6% frames of the two 25-FPS videos miss their deadlines. Note that S^3DNN yields a slightly worse overall performance than CUDA streaming, because prioritizing kernels in the scheduling queue and performing supervised synchronization introduces runtime overheads, particularly when the queue has a large number of kernels.

5.5 Evaluating Multi-GPU Scenarios and an Online Webcam Case Study

We have also assessed S^3DNN's efficacy under multi-GPU scenarios and using an online webcam case study, where S^3DNN is proven to bring significant improvements compared to YOLO. Due to space constraints, we put the detailed results and discussions in an appendix.

6 Related Work

GPU resource management. Many recent research has been conducted to optimize the performance of GPU-enabled heterogeneous platforms, providing system solutions [22], [23], [24], [25], [21], [26], [27], [28], [29], [30], [31] and theoretical guarantees [32], [33], [34]. The non-preemptive feature of GPU significantly harms its predictability [35]. To tackle this issue, a set of works focus on how to make GPU execution preemptive [36], [37], [38], [39], while some other works target at providing accurate timing estimation [40], [41].

Real-time DNN-based object recognition. DNNs have been extensively adopted in object detection for their impressive improvements in detection accuracy [16], [12], which is the core function of many image/video processing applications. With GPU-accelerated platforms, DNN-based object recognition is now capable of processing vision workloads in real-time, either through algorithmic optimization [8], [9], [10], [11], or trading throughput with accuracy [42], [3].

7 Conclusion

In this paper, we present S^3DNN–a systemic solution that optimizes the execution of DNN workloads on GPU in a real-time multi-tasking environment. Experimental results show that S^3DNN significantly outperforms state-of-the-art GPU-accelerated DNN processing frameworks in a real-time multi-tasking environment.

Appendix

Illustration of Algorithm 2

Algorithm illustration. We use an abstracted workflow diagram shown in Fig. 10 to illustrate how our proposed scheduler schedules the example DNN task set given in Fig. 6(d). At time 0, three DNN tasks are released simultaneously. According to their slack times, the priority queue is sorted in the order of $K^1_{3,1}$, $K^1_{1,1}$ and $K^1_{2,1}$. The scheduler will thus dequeue and schedule $K^1_{3,1}$ first, and then $K^1_{1,1}$ since there is no chance to run either kernel concurrently with another kernel. At time 6, three kernels wait in the scheduling queue in the order of $\{K^2_{3,1}, K^1_{2,1}$, and $K^2_{1,1}\}$ (Fig. 10(a)). Since the highest-priority kernel $K^2_{3,1}$ has a thread block utilization of 30%, the scheduler will scan the other kernels in the scheduling queue to increase concurrency. In this case, it finds that $K^2_{1,1}$ can be grouped with $K^2_{3,1}$ to execute concurrently (Fig. 10(b)). Then, since the thread block utilization of these two grouped kernels is 60%, still less than 1, the scheduler seeks to lookahead to check whether there is any later-released kernel that can execute with this group together. With a lookahead degree of 1, the scheduler will only check once whether the successor kernel of the the second-highest-priority kernel in the queue, which is $K^1_{2,1}$, can be grouped together. In this case, the total thread block utilization of the three kernels $K^2_{3,1}$, $K^2_{1,1}$, and $K^2_{2,1}$ is 90%, thus eligible to run concurrently. Thus, the scheduler will group these three kernels together and will reverse the priority ordering of this group of kernels with $K^1_{2,1}$ which is originally the second-highest-priority kernel waiting in the queue, to maximize concurrency. The scheduler has to schedule $K^1_{2,1}$ first because $K^2_{2,1}$ will not be released untile $K^1_{2,1}$ completes.

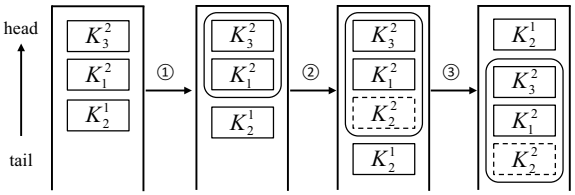


Fig. 10: Intuitive illustration of Algorithm 2 using the example given in Fig. 6 (d).

5.5 Evaluating Multi-GPU Scenarios and an Online Webcam Case Study

Multi-GPU scenarios. Since S^3DNN can also be applied in a multi-GPU environment, we have conducted experiments to evaluate its performance in such scenarios. The evaluation platform is a heterogeneous multi-GPU system consisting of a NVIDIA GTX 480 device and a NVIDIA Quadro 6000 device, and an Intel i7 multi-core CPU. The NVIDIA GTX480 device features higher GFLOPS but less memory capacity compared to NVIDIA Quadro 6000. As briefly discussed in Sec. 4.2, for multi-GPU systems, the Governor in S^3DNN applies an efficient bin-packing heuristic to assign fused video streams onto GPUs according to each GPU's memory constraints. The baseline compared in this experiment is the best performance among various possible partitioning of YOLO instances onto two GPU devices. For example, when the number of videos is

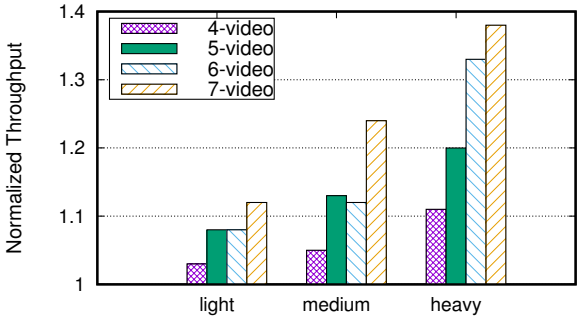


Fig. 11: Performance under multi-GPU scenarios.

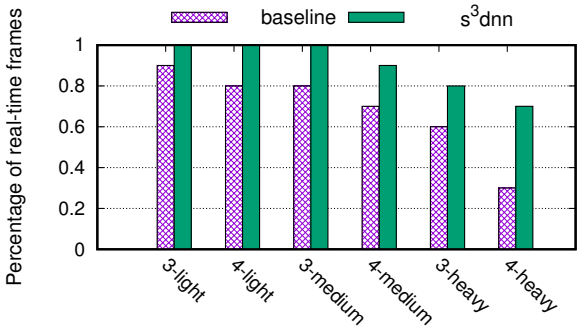


Fig. 12: Percentage of frames that meet their deadlines.

five and light DNN configuration is used, the baseline approach will assign three videos to GTX 480 and another two to Quadro 6000, which yields the best throughput among all partitioning possibilities.

Fig. 11 shows the throughput performance of S^3DNN compared to YOLO under different DNN configurations when processing different number of input videos. As seen in the figure, S^3DNN significantly improves the throughput performance under all scenarios except where there is only one video in the system (due to $<4\%$ runtime overheads which is reasonably small). The largest improvement (almost 40% improvement) occurs under the scenario with heavy DNN configuration and seven input videos. This is because in this case, the baseline solution assigns two videos on fast (high GFLOPS) but small (GPU memory size) GPU (i.e., GTX 480) due to memory constrains, leaving five video on slow but large GPU (i.e., Quadro 6000). This unbalanced assignment causes throughput loss. On the other hand, due to data fusion, S^3DNN can achieve a much more balanced partitioning, thus yielding a better throughput performance.

Case study: Online Webcam-based Object Recognition. Besides evaluating S^3DNN for offline video-based real-time object recognition, we conducted a case study evaluating the efficacy of S^3DNN under an online scenario, where online webcams continuously capture real-time video frames that require real-time object recognition processing. The major difference between using online webcams and offline videos is the following: for offline videos, the release time of each frame is rather flexible, i.e., whenever a frame completes, the next frame can be immediately released and fed to the system; while for online webcam-captured video streams, the release time of the frame is fixed, as defined by the FPS of

the webcam. Moreover, webcams often use a buffer to cache captured frames, with the advantage of hiding I/O latency. However, if the processing speed is slow, then the buffer will overflow and harm the real-time performance.

In this set of experiments, we use up to 4 online webcams, each of which is configured to be 15 or 20 FPS. We compare against YOLO, where each webcam is bound to a YOLO instance. Fig. 12 shows the results in terms of the percentage of the frames that meet their deadlines. The x-axis shows the evaluated scenarios, e.g., "3-light" denotes the scenario with three webcams and light DNN configuration. We observe that S^3DNN clearly outperforms YOLO in all scenarios, particularly when the system needs to process more workloads due to an increased number of webcams and/or heavier DNN configurations. For example, in the "4-heavy" case, baseline can merely process 30% of the frames in real-time while S^3DNN can achieve a schedulability of nearly 70%. Thus, S^3DNN is also effective in processing online video streams with enhanced real-time performance.

References

[1] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *ICCV*, 2015.

[2] NVIDIA, "Drive PX 2," http://www.nvidia.com/object/drive-px.html, 2016.

[3] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *MobiSys*, 2016.

[4] H. Bagherinezhad, M. Rastegari, and A. Farhadi, "Lcnn: Lookup-based convolutional neural network," *arXiv preprint arXiv:1611.06473*, 2016.

[5] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks," in *PDP*, 2010.

[6] NVIDIA, "Volvo to deploy drive px 2 in self-driving suvs," *https://blogs.nvidia.com/blog/2016/01/04/drive-px-ces-recap*, 2016.

[7] Autonews, "Carmakers tap nvidia's supercomputer to make leap toward self-driving cars," *http://europe.autonews.com/article/20160607/ANE/160609921/carmakers-tap-nvidias-supercomputer-to-make-leap-toward-self-driving*, 2016.

[8] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *CVPR*, 2016.

[9] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *CVPR*, 2014.

[10] R. Girshick, "Fast r-cnn," in *ICCV*, 2015.

[11] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *NIPS*, 2015.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *ACM MM*, 2014.

[13] NVIDIA, "Cuda 7 streams simplify concurrency," *https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/*, 2015.

[14] NVIDIA, "Kepler GK110," https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2014.

[15] J. Fritsch, T. Kuehnl, and A. Geiger, "A new performance measure and evaluation benchmark for road detection algorithms," in *International Conference on Intelligent Transportation Systems (ITSC)*, 2013.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[17] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[19] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014. [Online]. Available: http://arxiv.org/abs/1405.0312

[20] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, Jun. 2010.

[21] C. Augonnet, S. Thibault, and R. Namyst, "Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines," Ph.D. dissertation, INRIA, 2010.

[22] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-Class GPU Resource Management in the Operating System," in *USENIX ATC*, 2012.

[23] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *USENIX ATC*, 2011.

[24] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: operating system abstractions to manage gpus as compute devices," in *SOSP*, 2011.

[25] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *RTSS*, 2013.

[26] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO*, 2009.

[27] H. Zhou and C. Liu, "Task mapping in heterogeneous embedded systems for fast completion time," in *EMSOFT*, 2014.

[28] G. A. Elliott and J. H. Anderson, "Robust real-time multiprocessor interrupt handling motivated by gpus," in *ECRTS*, 2012.

[29] G. A. Elliott and J. H. Anderson, "An optimal k-exclusion real-time locking protocol motivated by multi-gpu systems," in *Real-Time Systems*, 2013.

[30] G. A. Elliott and J. H. Anderson, "Exploring the multitude of real-time multi-gpu configurations," in *RTSS*, 2014.

[31] Z. Dong, Y. Liu, H. Zhou, X. Xiao, Y. Gu, L. Zhang, and C. Liu, "An energy-efficient offloading framework with predictable temporal correctness," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 2017, p. 19.

[32] Z. Dong, C. Liu, A. Gatherer, L. McFearin, P. Yan, and J. H. Anderson, "Optimal dataflow scheduling on a heterogeneous multiprocessor with reduced response time bounds," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[33] Z. Dong and C. Liu, "Analysis techniques for supporting hard real-time sporadic gang task systems," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 128–138.

[34] W.-H. Huang, J.-J. Chen, H. Zhou, and C. Liu, "Pass: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling," in *DAC*, 2015.

[35] G. A. Elliott and J. H. Anderson, "Real-world constraints of gpus in real-time systems," in *RTCSA*, 2011.

[36] H. Zhou, G. Tong, and C. Liu, "Gpes: a preemptive execution system for gpgpu computing," in *RTAS*, 2015.

[37] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "Rgem: A responsive gpgpu execution model for runtime engines," in *RTSS*, 2011.

[38] C. Basaran and K.-D. Kang, "Supporting preemptive task executions and memory copies in gpgpus," in *ECRTS*, 2012.

[39] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *ISCA*, 2014.

[40] K. Berezovskyi, K. Bletsas, and B. Andersson, "Makespan computation for gpu threads running on a single streaming multiprocessor," in *ECRTS*, 2012.

[41] C. Gerum, O. Bringmann, and W. Rosenstiel, "Source level performance simulation of gpu cores," in *DATE*, 2015.

[42] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *SenSys*, 2015.