Primitives Enhancing GPU Runtime Support for Improved DNN Performance

Aditya Dhakal
University of California, Riverside
adhak001@ucr.edu

Sameer G Kulkarni
Indian Institute of Technology, Gandhinagar
sameergk@iitgn.ac.in

K. K. Ramakrishnan
University of California, Riverside
kk@cs.ucr.edu

***Abstract*—Deep neural networks (DNNs) are increasingly used for real-time inference, requiring low latency, but require significant computational power as they continue to increase in complexity. Edge clouds promise to offer lower latency due to their proximity to end users and having powerful accelerators like GPUs to provide the computation power needed for DNNs. But it is also important to ensure that the edge-cloud resources are utilized well. For this, multiplexing several DNN models through spatial sharing of the GPU can substantially improve edge-cloud resource usage. Typical GPU runtime environments have significant interactions with the CPU, to transfer data to the GPU, for CPU-GPU synchronization on inference task completions, etc. These result in overheads. We present a DNN inference framework with a set of software primitives that reduce the overhead for DNN inference, increase GPU utilization and improve *performance*, with lower latency and higher throughput.**

Our first primitive uses the GPU DMA effectively, reducing the CPU cycles spent to transfer the data to the GPU. A second primitive uses asynchronous 'events' for faster task completion notification. GPU runtimes typically preclude fine-grained user control on GPU resources, causing long GPU downtimes when adjusting resources. Our third primitive supports overlapping of model-loading and execution, thus allowing GPU resource re-allocation with very little GPU idle time. Our other primitives increase inference throughput by improving scheduling and processing more requests. Overall, our primitives decrease inference latency by more than 35% and increase DNN throughput by 2-3×.

***Index Terms*—GPU, edge-cloud, DNN, inference**

I. Introduction

Deep Neural Networks (DNNs) are the machine learning (ML) algorithms of choice for image recognition, speech recognition used in applications such as autonomous driving, language translation *etc.* Accelerators such as GPUs, TPUs, and FPGAs often provide an order of magnitude speedup of DNN processing compared to multi-core CPUs, They are often indispensable for real-time DNN inference. End-user devices like smartphones, IoT devices, AR/VR headsets, *etc.* lack the compute power to provide the desired low latency inference. Compute-heavy DNN inference tasks are often offloaded to the cloud to take advantage of powerful accelerators in the cloud [1]. Edge clouds with GPUs can host ML frameworks and achieve the low latency suitable for real-time DNN inference [2]–[4].

GPUs are well-suited for the highly parallel computations needed for DNNs. However, there are challenges using GPUs for DNN processing, especially in resource constrained Edge cloud servers. In a typical DNN execution, the application transfers a large batch of streaming data (*e.g.,* images) from the network to the GPU memory. Then, DNN kernels (equivalent to function code running in CPU) are launched with the maximum number of GPU threads to infer on the batch of data. However, we observe that transferring data to the GPU uses substantial CPU cycles and results in a 40% increase in inference latency for certain image recognition DNN models. Other inefficiencies exist while utilizing GPUs for DNN processing. GPU runtimes facilitate DNN kernels to run asynchronously with respect to CPUs, allowing the CPUs to queue the tasks for the GPU. Notification of task completion to the CPU is usually performed by placing a synchronization barrier or a callback function. Similarly, launching DNN kernels in the GPU, and notifications of task completion require the CPU to interact with the GPU using the GPU's runtime APIs, adding to the inference latency. Further, since an edge cloud is likely to be relatively resource-constrained compared to a centrally-located large cloud site, it needs its resources (especially the more expensive and power-hungry components such as GPUs) to be efficiently utilized. We have observed that many DNNs cannot fully utilize the power of current GPUs. Multiplexing GPUs across several applications can be an effective method to improve GPU utilization. However, multiplexing the GPU can increase the latency of DNN execution, causing the DNNs to miss their execution time constraints. In this paper, we are motivated by the need to utilize the GPU efficiently for low latency DNN execution. For this, we have utilized a set of software primitives that reduces the overhead (and thus latency) created by inefficiencies in GPU runtimes. We facilitate multiplexing of the GPU, while increasing throughput and maintaining low latency for DNN executions.

The first primitive, `GPU-DMA`, utilizes the DMA engine on the GPU to perform the primary task of efficiently transferring the data to the GPU while freeing up the CPU from performing a memory copy. Second, we use a primitive, `Sync-lite` that utilizes event-based APIs and a lightweight query mechanism to determine when a task in the GPU has completed, enabling us to perform rapid, low-overhead synchronization between the CPU and GPU. We utilize the concept Controlled Spatial Sharing (CSS) of the GPU, based on our previous work GSLICE [5], to multiplex GPU resources spatially among multiple concurrent DNN applications. However, CSS is limited to providing static allocation of GPU resources (Henceforth called percentage of GPU resources *i.e.,* GPU%). Therefore, we create a software primitive, `Overlapped-Execution`, that facilitates reconfiguring GPU resources to multiplex several DNN models

without any downtime. We have observed that running multiple instances of some DNN models with a smaller amount of GPU resources, rather than one instance being allocated the entire GPU, achieves higher inference throughput. We utilize this finding to devise another software primitive, `Multi-Model`, that dynamically changes the number of instances of a DNN model running, to meet throughput and latency requirements of DNN applications. To reduce GPU memory usage, thereby allowing us to run more applications on the GPU, we use a software primitive, `Param-Share`, that shares common data (*e.g.,* DNN parameters) among multiple instances of the same model. Further, batching is required for getting higher throughput during DNN inference, however, batching also increases latency. Moreover, the latency of batch size varies with variation in GPU resources that comes with spatial-sharing of the GPU. Therefore, we require a software primitive that can provide information on the runtime of a DNN based on batch size and GPU%. We name this primitive `Batch-Latency`.We briefly describe the context and motivation for these software primitives next.

By exploiting the the `Sync-lite` primitive in our testbed framework for DNN inference (as in a small edge cloud), we see a 20-30% reduction in inference latency. Using our two primitives `GPU-DMA` and `Multi-Model`, the overall system throughput is doubled. We are able to fit 20-25% more DNN models in the GPU through spatial sharing and `Param-Share`. Moreover, we are able to increase system throughput by 3× by supporting both spatial and temporal scheduling of DNN inferences with deadlines. We evaluate our complete framework with all the software primitives in a testbed with a single NVIDIA V100 GPU and also a cluster of 8 NVIDIA T4 GPUs and demonstrate that our framework multiplexing multiple DNNs improves GPU utilization and provides 3× higher throughput compared to state-of-the-art inference platform such as NVIDIA Triton server [6]. In summary, our contributions in this paper are:

- We describe our software primitives, which include enhancements to data movement, synchronization, improved multiplexing through overlapped execution, parameter sharing, and GPU resource management and scheduling.
- We evaluate each software primitive on a single NVIDIA V100 GPU and a cluster of NVIDIA T4 GPUs.

II. Related Work

A. Related Work

Various studies [6]–[11] have used temporal multiplexing of a GPU. However, works such as [12]–[14] have shown that spatial sharing can further increase GPU utilization and provide higher throughput compared to temporal sharing. Therefore, we also take advantage of spatial sharing of the GPU in our platform to multiplex applications. While spatial sharing of the GPU is beneficial, several studies [5], [15], [16] report that changing spatial resources (GPU%) is time-consuming due to the time it takes to load the DNN model to GPU. In this paper, we consider these challenges and create an inference framework with software primitives that spatially share the GPU providing higher throughput than temporal multiplexing solutions. Our primitives also aid in changing GPU% with negligible downtime.

Data transfer from streaming network packets to GPU adds significant delay for latency-critical systems [17], [18]. Solutions to data transfer often include creating large batches [19], which require a lot of CPU cycles or modification of device drivers [17], [20]. In contrast to those approaches, we utilize `GPU-DMA` software primitive to utilize GPU's DMA engine to do the primary task of data transfer from the network, thus, reducing the CPU load and data transfer latency without device driver modification or using the GPU runtime software. Many studies have pointed to different factors contributing to latency while multiplexing the GPU. Studies such as [21], [22] highlight that synchronization between GPU-CPU also adds significant latency to the processing. Other works, such as [23]–[26], observed that multiple applications running concurrently could interfere with each other's processing, resulting in higher latency for kernel execution. We learn the lessons from those studies and present our `Sync-lite` primitives to cut the synchronization time, and utilize CSS supported with `Overlapped-Execution` for dynamic allocation to isolate the applications' GPU resources and avoid interference that can increase processing latency.

Different approaches for scheduling applications in GPU has been studied by Nexus [27], Gandiva [7] and others [9], [28]–[30] present deadline aware tasks-scheduling in the GPU. These work only focus on time-sharing the entire GPU, which, might lead to underutilization of GPU. In contrast to these work, our platform can schedule the DNN execution spatially (with varying GPU%) and temporally with help of `Batch-Latency` primitive which predicts the latency for DNN inference at certain batch size and GPU%.

III. Motivation & Overview of Primitives

1) Spatial Multiplexing of the GPU: CUDA Streams and CUDA Multi-Process service (MPS) are two default methods of spatially sharing the GPU in NVIDIA GPUs. However, we observed that the streams and default-MPS lead to some DNN models interfering in each-other's execution due to applications vying for same GPU resources, resulting in increased latency. Isolating the GPU resources for each model, by truly virtualizing the GPU, can remove this interference. MPS allows spatial sharing within a single GPU. However, with MPS, the GPU% allocation is static for a process's lifetime. Changing it typically requires a long (2-5 seconds) downtime, with the GPU necessarily being idle, if it is done simplistically. Works such as GSLICE [5] demonstrate that controlled spatial sharing (CSS), where GPU resources are allocated and isolated while spatially sharing the GPU, improves overall utilization and throughput of the GPU subsystem. CSS uses MPS [13] with a fixed GPU%. To meet the challenge of dynamically changing the GPU%, our software primitive `Overlapped-Execution` helps to continue to serve inference requests, while the model with the new GPU% is prepared. Once the model with new GPU% is ready, we switch to inferring with the newer model. We see

that our primitives cut the GPU downtime to just a few hundred microseconds compared to seconds of downtime otherwise.

2) GPU Utilization of a DNN: Profiling DNN applications, we observed that DNN models often cannot utilize all of the GPU resources provided to them. Just increasing the GPU% for a model does not reduce the inference latency. Similar observations were made in [5], [12]. Therefore, providing just the right GPU% to a DNN model can result in as good inference latency and throughput as providing 100% GPU. We name this right GPU%, after which we see diminishing returns in the latency, as the "Knee." We can also increase the throughput of a DNN model significantly by running multiple instances of the same model, each with a share of GPU% just around its knee. But, running additional instances occupies more aggregate GPU memory. To balance this, `Multi-Model` supports running multiple instances providing the same service but at the same time meeting a tenant's performance (latency) requirements. We further demonstrate that the parameter sharing technique, `Param-Share`, shares the same set of DNN parameters across multiple instances of the same model, thus reducing the total GPU memory demand.

3) Multiplexing of the GPU: Many DNN applications infer data with more than one DNN model to get a final result. *e.g.,* a model such as Faster-RCNN [31] and Masked-RCNN [32]. These constituent models have varying GPU% and can leave the GPU under-utilized. Some other DNN models can be run to utilize the unused GPU%. For this, an appropriate scheduling mechanism that schedules based on time and space (GPU%) needs to be considered. Moreover, 'Batching' can also greatly help increase the throughput [9]. To meet deadlines, the scheduler needs to know the runtime of a batch of requests for each constituent DNN. This runtime can vary greatly with different GPU%. This motivate use to design the `Batch-Latency` primitive, that provides information on the runtime of a DNN based on batch size and GPU% to aid with scheduling. Together with `Sync-lite`, which allows the scheduler to know when a DNN has finished its tasks, `Batch-Latency` then helps to find which batch size of a DNN model can fit in the available GPU% increasing the overally system throughput and GPU utilization.

IV. Measurement-driven Understanding of DNNs

We now demonstrate utility of CSS in our testbed as well as discuss DNN's limitations in utilization of GPU's parallelism.

A. Testbed Environment

Our single-GPU experimental testbed uses a Dell Poweredge R740xd with Intel(R) Xeon(R) Gold 6148 CPU with 20 cores, 256 GB of system memory, and one NVIDIA V100 GPU, and an Intel X710 10GbE quadport NIC. The V100 has 80 streaming multiprocessors (SMs) and 16 GB of memory. Our multi-GPU testbed consists of 8 NVIDIA Tesla T4 GPU. We use 4 quadport Intel XL710 10 GbE NICs (40Gb aggregate capacity). Each Tesla T4 GPU has 16 GB of GPU memory and 40 SMs.

We use the 'GPU%' terminology used in CUDA MPS to indicate the amount of GPU resources is allocated. *e.g.,* a model getting 50% in Tesla T4 GPU means it can use at most 20 SMs (out of a total of 40). This GPU% can be set as an environmental variable CUDA_MPS_ACTIVE_THREAD_PERCENTAGE before the application performs the GPU initialization. It is possible to run multiple applications with their GPU%, adding up to more than 100%. We call this ***over-subscription*** of the GPU.

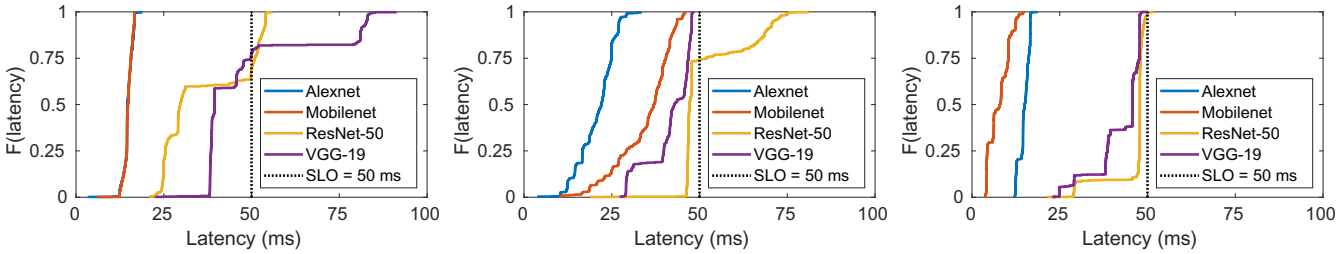
We use the PyTorch [33] and TensorRT [34] platforms for our evaluations. Our DNN workload consists of color images of resolution 224×224 for object recognition DNNs. This is the image size utilized widely by most object recognition models, and most of the pre-trained models can only infer that particular resolution; therefore, we use images of this resolution exclusively. To evaluate the overheads of data transfer to the GPU, we transmit each image through the network as 588 UDP packets, where each packet has a payload of 1 KB. We use Moongen [35] and TCPreplay as traffic generators for transmitting images. With a 10 GbE link between the traffic generator and the receiving server, we can transmit 1920 images per second. For all our experiments, we mainly report the execution time of inference and exclude the additional network-related latencies contributed by network protocols, including HTTP, TCP, and/or UDP. We use OpenNetVM [36], a network function virtualization platform, to create our inference framework where, we host machine learning models as VNFs and infer on streaming workload from traffic generators.

B. Controlled Spatial Sharing (CSS) of GPU

We conducted an experiment to demonstrate the necessity of controlled spatial sharing. We multiplexed 4 popular DNN models, Alexnet [37], Mobilenet [38], ResNet-50 [39] and VGG-19 [40] on a NVIDIA V100 GPU. We use temporal sharing, spatial sharing with default MPS, and controlled spatial sharing with resource isolation. Among these models, Alexnet and Mobilenet are latency optimized, *i.e.,* they have relatively low compute requirement and complete quicker compared to the accuracy optimized ResNet-50 and VGG-19 models. We enable adaptive batching which dynamically sets the batch sizes to meet the deadline [5], [9] and set the SLO as 50 milliseconds to enable a robust and interactive system. We present the cumulative distribution function (CDF) of latency of different models in Fig. 1a to demonstrate the variation in inference latency for each experiment. With temporal sharing of the GPU, the accuracy optimized models exceed their SLO frequently. Temporal sharing provides an equal timeshare of the GPU irrespective of the DNN workload, which results in penalizing models that require more computation time.

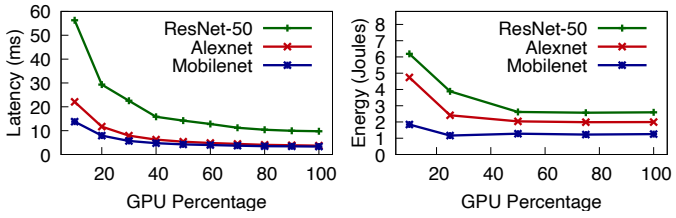
We repeated the experiment by multiplexing GPU using default MPS, and the CDF is shown in Fig. 1b. VGG-19 meets the SLO, but latency for all other models increases significantly. Furthermore, ResNet-50's inference misses the deadline many times. Default MPS spatially shares the GPU but does not isolate resources. VGG-19 requires more computation and thus interferes with the execution of other models. This increases the latency for all the models.

Finally, we present the CDF of latency of controlled spatial sharing (CSS) in Fig. 1c. We provided 15% GPU each to



(a) Temporal Sharing (b) Default MPS (c) Controlled Spatial Sharing

Fig. 1: CDF of latency of multiple DNNs with different methods of multiplexing GPU



(a) Latency vs. GPU% (b) Energy per batch

Fig. 2: (a) Latency vs. GPU% for inferring batch of 8. (b) Energy spent per batch. TensorRT with V100 GPU

Alexnet and Mobilenet and 30% and 40% GPU to ResNet-50 and VGG-19. We picked those GPU% values roughly based on their computational complexity (floating-point operations) of each model. With CSS, the latency of each model does not vary as much as temporal sharing or default MPS. This is due to the resource isolation provided by CSS. Moreover, all these models do not violate their SLO of 50 ms, thus, allowing them to run all of the models twice in the 100 ms time. Being able to execute the DNN models more often than temporal sharing or default MPS increases overall throughput.

C. GPU utilization during DNN inference

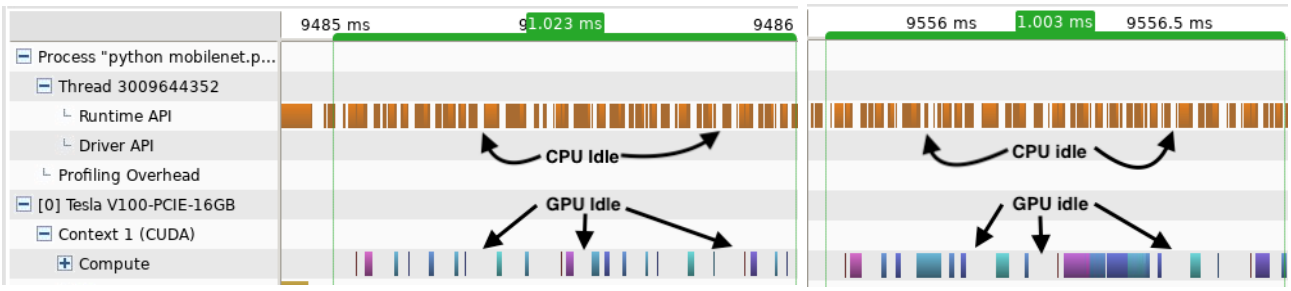
We profiled several DNN applications, giving them different GPU% with CSS, and observed that inference latency does not decrease much when provided more than a certain GPU%. We show the inference latency of different DNN models in the TensorRT DNN platform in Fig. 2a. We see that for all the models, latency is highest with a very low GPU% (10%) but decreases rapidly as the GPU% is increased. However, after a certain knee GPU%, further reduction in latency is slight or nonexistent.

We further investigated this by measuring the energy used by the GPU to infer a batch of requests at different GPU percentages. We present the measurement results in Fig. 2b. One noticeable feature is that the energy plot shows the same trend as the latency plot. It takes more energy to infer a batch for the more complex models (ResNet-50) than, the less computationally complex models. Furthermore, it takes more energy to infer the batch of requests when the inference task is provided a lower GPU%. This is because, at a lower GPU%, the DNN model takes longer to complete the inference task, drawing (albeit lower) power throughout. At a higher GPU%, the inference is faster and power is drawn for a shorter period. We can also see that beyond a certain GPU%, the GPU spends the same amount of energy to process a batch, meaning the DNN model is not able to utilize more GPU resources beyond certain GPU%.

We profiled the Mobilenet model running on the PyTorch platform using the NVIDIA NVPROF [41] profiler, shown in Fig. 3. The first profile (Fig. 3a) shows Mobilenet running with 100% GPU, while the second profile (Fig. 3b) shows Mobilenet inferring with only 15% GPU. We extracted one millisecond time slice out of the whole profile to depict the CPU and GPU behavior at a fine granularity. Having 100% of the GPU lets the individual kernels run quickly, but the next kernel is not yet ready to run and leads to gaps (idling the GPU). These idle times are caused by the inefficient launching of DNN kernels by the ML platform. The runtime API (3rd row in Fig.) also shows gaps while submitting the tasks to GPU. In the profile where Mobilenet runs with just 15% GPU, the runtime API's gaps remain, but we see that the GPU kernels run a little slower (thus taking longer). However, the added latency completely fits within the idle time between the kernels. Therefore, the overall inference latency is virtually identical (1.023 ms with 100% GPU and 1.003 ms with 15% GPU). Therefore, we can see that having a lower GPU% allows the model to use the gaps between kernel execution and still get the inference latency to be as good as with 100% GPU. Giving a lower GPU% (knee%) to a model that cannot use a higher GPU% then allows us to share the GPU across multiple other services (*e.g.,* in an edge cloud server.) We note that DNNs with compute-heavy kernels that run longer do not show much of a gap between consecutive GPU kernel executions. Therefore the knee for compute-heavy DNN models is a much higher GPU%.

V. Evaluation of Primitives in GPU Runtime Software for Enhanced Performance

We now describe each software primitive in detail. We implement these primitives in the DNN inference framework, as seen in Fig. 4. We utilize the OpenNetVM shared memory buffer to receive data from NIC. OpenNetVM can run multiple inference applications (IAs) as virtual network functions (VNFs). They load DNN models and a software primitives library that facilitates spatial sharing of the GPUs improved data transfer of the payload of network packets to the GPU, the notification from GPU, and the GPU resource allocation module, which facilitates dynamically changing the GPU resource allocation. Each IA maintains a buffer in the GPU for DNN inference data. IAs also load the DNN model to the GPU and start the inference when the application data is ready. In a multi-GPU cluster, each GPU is shared spatially across multiple IAs with each IA receiving a certain fraction (GPU%) of the total GPU resources, as seen in Fig. 4. GPU 1 is shared between model 1



(a) Mobilenet with 100% GPU (1 ms profile)

(b) PyTorch Mobilenet (15% GPU)

Fig. 3: Inference profiles (from NVPROF tool, 1 millisecond for both (a) and (b)), Mobilenet on PyTorch.

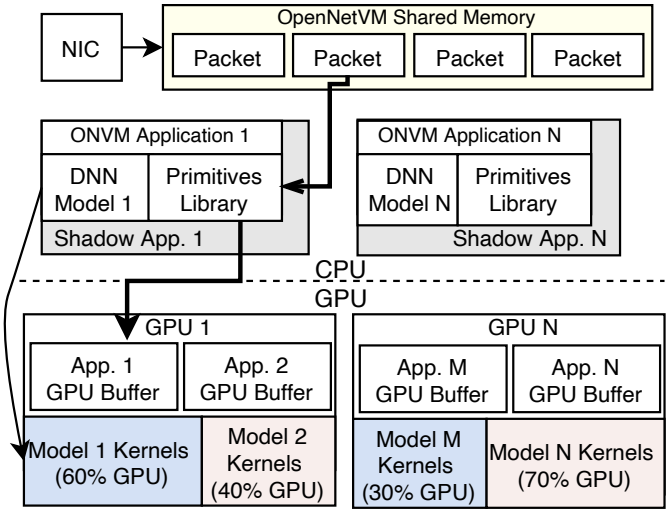


Fig. 4: DNN Inference Platform Architecture

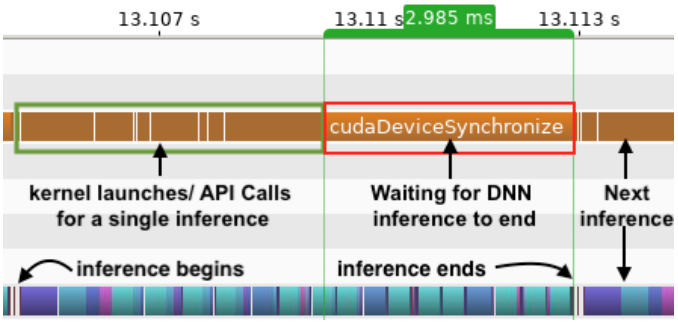


Fig. 5: ResNet-18: CPU wait time

(60%) and model 2 (40%). Similarly, GPU N may be shared between two models with a 30% and 70% split. For a multi-GPU cluster, our framework maintains a data structure keeping track of the application running in each GPU and uses packet metadata to transfer the application data to appropriate GPU.

A. *Task Completion Notification*

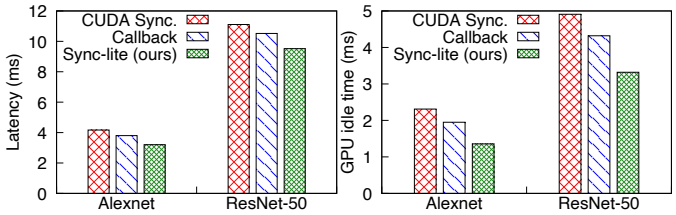
GPUs maintain a queue, where the GPU's runtime API calls and kernel launches are en-queued for subsequent GPU processing. A single CPU thread can en-queue an entire DNN model, composed of multiple kernels processing an input. Then the CPU can switch to other tasks. However, this asynchronous execution in the GPU implies that there can be difficulty in knowing when the submitted task has completed execution.

An alternative to notification of task completion is to use CUDA Graphs [42]. CUDA Graphs uses a predefined execution graph for GPU kernels with their dependencies and lets the GPU runtime handle launching of tasks in the right order. However, we do not use CUDA graphs as it does not support running DNN models in different processes, a requirement for resource isolation.

Another commonly used method for notification of task completion in a GPU is to put explicit synchronization barriers between the CPU and GPU and wait for the GPU to finish a given task. Fig. 5 shows the profile of inference with ResNet-18, which shows a large amount of CPU time being spent on running (actually being blocked on) an API function `cudaDeviceSynchronize()`. Explicit synchronization idles the CPU for prolonged intervals waiting for the GPU kernel to complete and return the result. Yet another method is to use a `callBack` API to notify the end of execution of a kernel by invoking a callback. However, the current implementation of callbacks poses challenges for GPU multiplexing. First, the callback routine blocks all the subsequent execution on the GPU until the callback is resolved, resulting in the idling of the GPU. Second, the callback context on the CPU is forbidden from invoking any of the CUDA APIs. Thus, the callback can only function as a signal that some task ended in GPU but cannot launch another pending GPU-related task. This limitation requires an additional CPU context and signaling scheme to perform any GPU-related operations. To overcome this, we devise a lightweight method for the CPU to obtain the GPU task completion status, `Sync-lite`.

```
void* Sync-lite(DNN-Model, Input-Batch,
    cudaEvent, Timer)
```

Our primitive takes as input, the DNN model, an input batch for inference, a cudaEvent object and a timer. The CUDA API function `cudaEventRecord()` allows us to put an event marker (cudaEvent) at the end of the DNN's execution. `Sync-lite` infers the Input-Batch with the DNN-model and subsequently *records* the completion of inference in cudaEvent object. The timer checks at an interval of 100 micro-seconds if the cudaEvent has been recorded by using another API function, `cudaEventQuery()`. Our event checking is lightweight, taking about 2 μseconds in our system. If the cudaEvent has been recorded, meaning the inference has been completed, our function returns the memory address of the inference result.



(a) Latency of inference (ms) (b) GPU idle time (ms)

Fig. 6: (a) inference latency and (b) GPU idle time during inference; 1 image w/ different synchronization primitives.

To evaluate the improvement, we inferred (Alexnet and ResNet-50 on TensorRT) with synchronization, callback, and our event-based `Sync-lite` notification approaches. We placed 1 instance of a model (*e.g.,* Alexnet) in each of 4 different GPUs (1 model per GPU), and each model inferred images with a batch size of 1. Our intent on running 4 models in 4 GPU is to simulate realistic conditions when multiple models are running and finishing their job in different GPUs. We computed the 99^{th} percentile tail latency for inference with each method and present the results in Fig. 6a. For the Alexnet model, `Sync-lite` is the fastest, with one inference taking 3.2 ms. The Callback approach added $\sim$600μs idle time on GPU, resulting in a 3.8 ms average inference latency. The synchronization approach is the slowest, taking 4.17 ms. Our event-based approach `Sync-lite` is **(18% and 30% faster)** than the callback and synchronization approach, respectively. Similarly, for ResNet-50, `Sync-lite` was **(10.5% and 16.7% faster)** respectively. For compute-heavy models such as VGG-19, the notification approach makes a relatively small difference as the model inference time is large. We also computed the time GPU is idle and present in Fig. 6b. The time spent by GPU to infer an image is the same for all three notification methods; therefore, looking at the GPU idle time shows how much CPU time each method uses. We observe that `Sync-lite` reduces idle time of GPU by more than 40% and 30% for Alexnet and ResNet-50, respectively. Therefore, effective task completion notification reduces inference latency and increases GPU utilization by reducing GPU idle time, especially with lighter models.

B. GPU-DMA for Network Data

Transferring streaming data arriving from the network to the GPU is expensive in terms of CPU cycles. The usual technique is to have the CPU copy the data from the packet payload and then again transfer the data using GPU runtime APIs such as *cudaMemcpy*. Hereafter called *CUDA-Copy*. We utilize the GPU-resident DMA engine rather than CUDA API functions running in the CPU to transfer data. Our GPU-DMA primitive is:

```
int GPU-DMA(network-pkts-addr, numPkts,
    gpu-buffer, gpu-id)
```

`GPU-DMA` takes the address of all the received packets, the number of packets, GPU buffer, and GPU id of the GPU where the data should be stored as arguments. `GPU-DMA` launches GPU kernels with the threads that use NVIDIA's unified virtual address (UVA) [43] mechanism to trigger the GPU DMA to scatter-gather the data from system (CPU) memory. As the DMA has the ability to scatter-gather, removing the need to accumulate packet payload into a contiguous buffer. Once the data is fetched, GPU threads read the metadata in the packet payload and place the data in the right place in the GPU buffer. After the data is transferred, the primitive returns the number of requests ready to be inferred in the GPU and launches the DNN kernels to infer the data in the GPU buffer. To accommodate batching, we only initiate the DMA transfer to GPU once we are sure enough data for a batch has already arrived through the network. This is to ensure that batch in GPU buffer is in contiguous memory as required by DNN kernels. To support DNNs running on multiple GPUs, the primitive checks where the data is destined to, by checking the packet five tuple, then activates the GPU-DMA in the target GPU. Our `GPU-DMA` implements a similar design as in [44]. However, we are different in that we support batching and DMA transfers across multiple GPUs in a cluster.

We experimented by comparing the `GPU-DMA` with the alternative CUDA-copy We noted the time taken to infer a batch of images by Alexnet, ResNet50, and VGG-19. We see from Table I, that `GPU-DMA` cuts overall inference time. Moreover, in our system, the CPU cycles spent to transfer 8 images to GPU is on average 2506216 for CUDA-Copy while 86263 for `GPU-DMA`, two orders of magnitude lower. We evaluate `GPU-DMA` further in an 8 GPU cluster in § VI-A.

TABLE I: Inference latency with batch of 8 in 1 V100 GPU

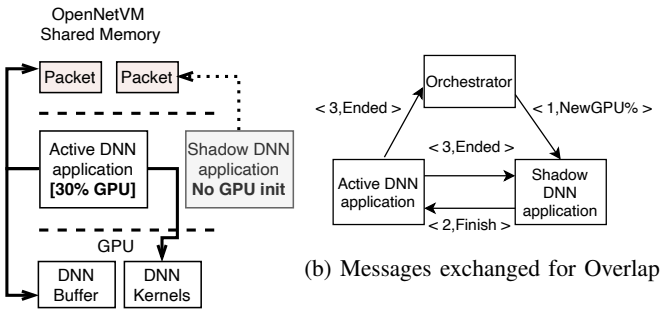
DNN Model	CUDA-Copy Inference Latency (ms)	`GPU-DMA` Inference Latency (ms)
Alexnet	4.01	3.67
ResNet-50	10.53	9.76
VGG-19	30.95	30.49

C. Dynamic adaptation of GPU resources

The GPU resources provided to an application might require reconfiguration due to the variations in the workload, *i.e.,* the change in the arrival rate of the tasks (especially with streaming data) or the variations in the number of concurrently executing applications. We need a system that can dynamically adjust the GPU resource partition of all the active applications. A typical technique would be to start a new process with a new required GPU%. However, doing so has a drawback; it takes a few seconds to get a DNN ready to infer as it takes time to load the model onto the GPU. We present an `Overlapped-Execution` primitive to overcome these drawbacks. We explain the two main functions of the `Overlapped-Execution` primitive next.

Monitoring and Detection: Our primitive facilitates tracking the arrival rate and service rate of inference tasks and the achieved SLO for each application using simple counters and a lightweight *DPDK [45] timer* interface and determine if the application is overloaded or under-utilizing GPU resources, and correspondingly triggers reconfiguration of the GPU%.

Resource readjustment: We create application replicas (multiple processes on the CPU in active(1):shadow(n) mode) and employ a deferred GPU initialization for the replica (shadow/standby). Also, we incorporate a switchover scheme such that the active application processing the inference continues to make progress until the new shadow instance,



(a) Active and Shadow's access

(b) Messages exchanged for Overlap

Fig. 7: Overlap Execution Process

with its updated resource allocation, is ready to take over. As seen in Fig. 7a, the active DNN application has access to the OpenNetVM shared memory for accessing packets and GPU memory and kernels for DNN inference processing. Meanwhile, the shadow DNN application only accesses the CPU side entities such as shared memory and avoids interaction with GPU, so that it waits to initialize the GPU.

This orchestration framework uses the event-action control loop and asynchronous message notification to the active and shadow processes, as seen in Fig. 7b. We use the following software primitive in the orchestrator to trigger the GPU% change.

```
int ChangeGPU%(NewGPU%, shadowDNN)
```

The primitive sends the notification to the shadow DNN to start loading a DNN model with the desired *NewGPU%* (1 to 100). The function returns a positive integer if the message was successful. The shadow DNN can now start initializing the GPU and configure the GPU resources with the input *NewGPU%*. Once the shadow DNN application completes the initialization phase, the shadow application next notifies the Active DNN (original process) to complete its current set of inference tasks and stop processing any further inference operations with message *<2, Finish>*. After the active instance completes processing all the remaining tasks, it sends a message to the shadow DNN application and the orchestrator *<3, Ended>* indicating it has completed all its tasks and indicates that the shadow DNN instance can now start performing the inference operations. The shadow is transitioned to be the new active GPU task, while the earlier active process is terminated. A new shadow is created subsequently. This mode switching ensures that active DNN is running while the shadow is readied; thus, it provides a loss-free and interruption-free inference processing for the DNN model.

We demonstrate the overlapped execution while changing GPU% in Fig. 8. We use a ResNet-50 model in our V100 GPU testbed. In this experiment, we show the baseline (Top plot) where the GPU% does not change. We show the GPU% change without overlap in the middle plot and GPU% adjustment with overlap at the bottom plot. In the timeline, we change the request rate from 520 to 680 at the 3-second mark. As the request rate becomes higher, our system determines the GPU% should be changed from 40% to 50% to meet the requirement as well as meet the SLO for higher rate of request . The baseline plot does not change and continues at the lower throughput. In the

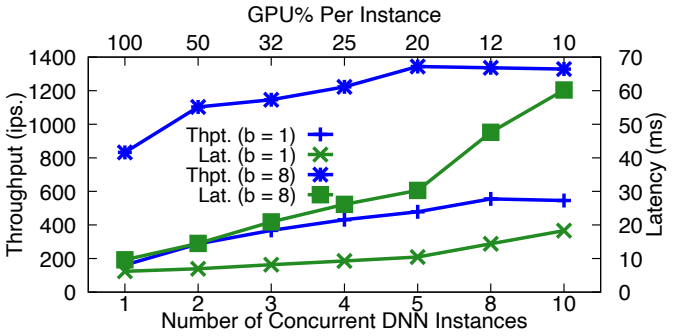


Fig. 9: Total throughput and 99 percentile latency of multiple concurrent instances of ResNet-50 (batch size 1 and 8) in a GPU

middle plot without overlap, the DNN process has to be killed, and a new one started. In this case, ResNet-50 takes 3 seconds to load and get ready with a higher GPU%. During those 3 seconds, none of the requests are processed as the service is not active. With overlapped execution (bottom plot), the ResNet-50 active process with 40% continues processing requests until the shadow ResNet-50 application with the 50% GPU allocation is ready. Then it nearly seamlessly switches the inference to the new application and gets higher throughput to meet the incoming rate. Our actual downtime is only about 200μs, continuing to provide service almost throughout the switchover.

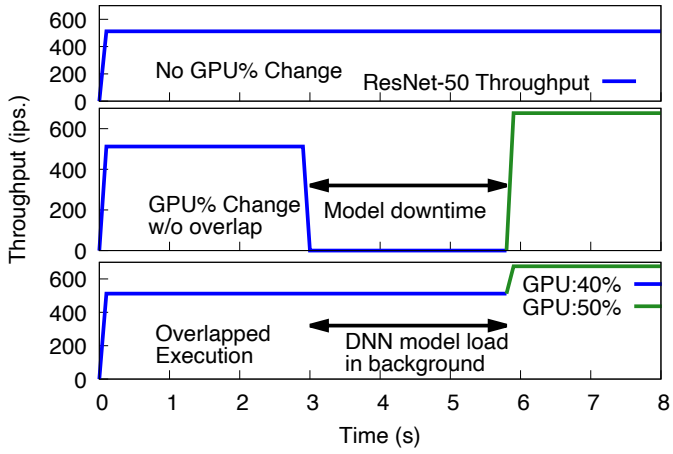


Fig. 8: ResNet-50 Overlapped execution while adjusting GPU

D. Creating Multiple DNN Instances for Higher Throughput

Fig. 2a shows that for a batch of 8, a DNN model with lower GPU% can still infer DNN requests with latency not too large compared to inferring requests with 100% GPU. Therefore, running four instances of the DNN model, say Alexnet at 25% GPU each, will provide 4× the throughput compared to running 1 Alexnet model with 100% GPU, without sacrificing the latency for inference. Similarly, running two instances of ResNet-50 model at 50% GPU will provide higher throughput than running one at 100%.

We conducted an experiment with ResNet-50 model inferring batch sizes of 1 and 8 to evaluate how running multiple instances will affect the throughput and latency of a model. We increased the number of instances of the model while proportionally reducing the GPU% of each request. *e.g.,* when

two instances are running, each will get 50% GPU, with 4 instances, each will get 25% each, and so on. We present throughput and latency of such setup in Fig. 9. With a batch size of 1, the throughput more than doubles going from 1 instance with 100% to 4 instances with 25% each, while latency only increases by about 25%. A similar trend is seen with a batch size of 8. The throughput increases by 1.5× going from 1 instance with 100% GPU to 5 instance of 20% GPU each. We however, see the latency increase is steeper when inferring batch of 8. Inference with a larger batch size is able to utilize more GPU resources. But, with this lower GPU% there is an increase in latency. Nonetheless, the latency remains well below 50 ms and continues to be useful for an interactive system. We should note after a certain number of instances; there is no further increase in throughput but a rapid increase in latency. The resulting low GPU% is insufficient for this model and results in a drastic increase in latency.

We use the following primitive that takes the DNN-model profiling data, such as Fig. 2a and Fig. 9, to determine the appropriate number of instances of the same model to be used.

```
int[] Multi-Model(DNN-model, available-
    GPU%, deadline)
```

This software primitive takes DNN-model profiling data, such as Fig. 2a, the available unused GPU%, and the inference deadline the DNN model has to meet. With this information, the primitive returns two integers, specifying the number of instances of the DNN-model to be run and at what GPU%.

Placing multiple instances of a DNN model in a GPU has two advantages. First, more requests for the model can be directed to the same GPU buffer, simplifying the data transfer to GPU. Second, we can utilize parameter sharing, *i.e.,* the new instance of the model can use the same DNN weights and parameters already existing in the GPU buffer of the existing model.

E. Parameter sharing of DNN models

DNN models have learned weights and parameters necessary for inference. Often these weights occupy a significant amount of GPU memory when the model is loaded to GPU. However, these weights and parameters are invariant as the inference process does not change these weights. Therefore, many instances of the same model can share the same sets of weights *i.e.,* they can share the parameters without needing to upload their own weights to the GPU. We utilize this parameter sharing in two scenarios. First, when adjusting the GPU resources, the replica (shadow) instance has to load its model into GPU with an updated GPU% specified, while the active process infers for incoming requests. We have observed that the loading of another model does indeed consume additional memory temporarily, but parameter sharing substantially mitigates the increase in the total memory footprint. Second, when we are starting another instance of the same model to increase inference throughput, as described above, the new model can just link to the weights of an already running model. This reduces the burden on GPU memory. We use the following primitive to share the weights among the models.

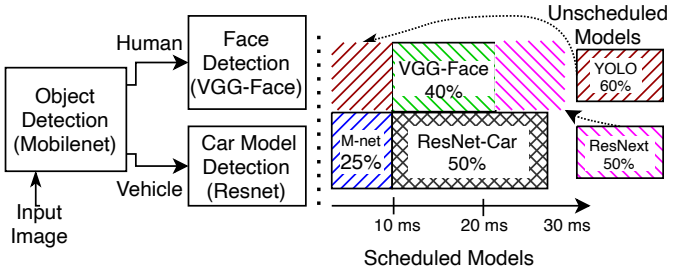


Fig. 10: Scheduling Multiple DNN Applications

```
int share-parameter(cudaIPC GPUParamAddr,
    newDNNModel)
```

We create CUDA Inter-Process Communication (IPC) pointers for all of the parameters of a DNN model loaded in the GPU. We then share these IPC addresses to every new instance of the same model. These models can link to the existing GPU buffer rather than loading their own weights. We now show how parameters sharing can fit more models in the GPU. We show 3 different models in Table II. Each of the model's parameter takes some space in GPU, while the model also occupies other private GPU buffer space necessary for its inference. With parameter sharing, each new model sharing the parameter will occupy less space in the GPU, allowing us to add more instances of the model. As we see from Table II, parameter sharing allows 1 more ResNet-50 instance and 2 more VGG-19 and ResNext-50 instances in *one* NVIDIA T4 GPU (16 GB memory). In a multi-GPU cluster, these memory savings can allow even more models to fit and run concurrently.

TABLE II: No. of models hosted with parameter sharing (P.S.)

Model	Weights (MB)	Other GPU buffer (MB)	No. of models Without P.S.	With P.S.
ResNet-50	100	1400	10	11
VGG-19	549	1475	8	10
ResNext-50	248	1252	10	12

F. Primitive to support scheduling of DNN models

We now present a scenario where spatial and temporal scheduling of DNN applications can improve utilization of the GPU and get higher overall system throughput. Fig. 10 (left) shows an example DNN service (hereafter called **DNN-Recog**), which first infers input images with Mobilenet to find if it includes humans or vehicles. The application processes human images with a face detection DNN (VGG-Face [46]) and vehicle images with a car model detection DNN (ResNet-Car [47]). We present a schedule of these DNNs in Fig. 10 (right). We show the schedule of the 3 models (Mobilenet, VGG-Face, and ResNet-Car) with solid boxes in Fig. 10. As VGG-Face utilizes 40% and ResNet-Car uses 50% GPU, they can run simultaneously. However, two other models, ResNext [48] (50% GPU) and Yolo [49] (60% GPU), cannot be fit into the schedule. At the beginning of the schedule, Mobilenet infers images using 25% GPU. Here, the scheduler has an opportunity to run Yolo (60% GPU demand) with the remaining GPU. But, to avoid interfering with ResNet-Car, the scheduler needs to estimate how long Mobilenet will run. Using that, our primitives help find the right batch size for Yolo, so it completes

TABLE III: DNN applications latency with different scheduling

App.	Latency (ms)		Batch-Size		Throughput (rps)	
	Temporal (Triton [6])	Batch-Lat.	Temp.	Batch-Lat.	Temp.	Batch-Lat.
DNN-Recog	24.8	28.7	2	8	67	266
Yolo	5.1	6.1	0.5	2	15	60
ResNext	4.9	6.3	0.5	1	15	30

inference before (brown shaded region) VGG-Face and ResNet-Car begin inference. Since a model's runtime may vary (*e.g.,* VGG-Face ends execution earlier than ResNet-Car). Then, our scheduler accommodates the ResNext model with a batch size provided by our primitive. We present our primitive next.

```
int Batch-Latency(DNN-Models[], GPU%,
    interval)
```

`Batch-Latency` takes as input the profiles of the different DNN models currently loaded to GPU, the currently available GPU%, and the time interval until an apriori scheduled model will run. We compare the scheduling utilized by our primitive to pure temporal scheduling, where models get exclusive GPU access on a round-robin basis. We fix the deadline of 30 ms to simulate object detection at 30 frames/sec. rate. We set the scheduling round time to be 30ms. We observe the average throughput and latency obtained by the models and present it in Table III for temporal (using NVIDIA Triton) and spatio-temporal scheduling. Also shown is the batch size provided by `Batch-Latency`. DNN-Recog gets 4× the throughput with spatial-sharing because two of its compute-heavy components, VGG-Face and ResNet-Car, can run concurrently. This is not possible with just temporal sharing of the GPU. Moreover, the other two applications, Yolo and ResNext, only get scheduled in alternate round-robin rounds with temporal sharing due to their high latency (with a batch size of 1 in alternate scheduling rounds). However, with the case of spatio-temporal sharing, `Batch-Latency` helps schedule them concurrently, as seen in Fig. 10. We achieve higher throughput without violating the deadline of 30 ms. Overall, spatial-temporal scheduling with `Batch-Latency` increases the throughput of the system by 3×.

VI. Evaluation of Overall Inference Framework

A. *GPU-DMA in Multi-GPU cluster:*

First, we evaluated the benefit of `GPU-DMA` in a multi-GPU high bandwidth scenario. We utilized our 8 GPU cluster to host different models and evaluated the throughput obtained while inferring with a maximum batch size of 32. We compare `GPU-DMA` with *CUDA-Copy* method. We also present the baseline throughput, which we name *Default MPS*, where we infer without batching (*i.e.,* batch size of 1). *Default MPS* is identical to CUDA-Copy method, except it only infers 1 image at a time. We present the results in Fig. 11. With the relatively lower complexity models (Alexnet and Mobilenet), `GPU-DMA` increases the throughput by 2× compared to the *CUDA-Copy* method and 3× compared to the *Default MPS* method. The CUDA-copy method uses more CPU cycles to gather the payload data from packets into a contiguous buffer. The CPU becomes the bottleneck and therefore achieves lower throughput. Throughput

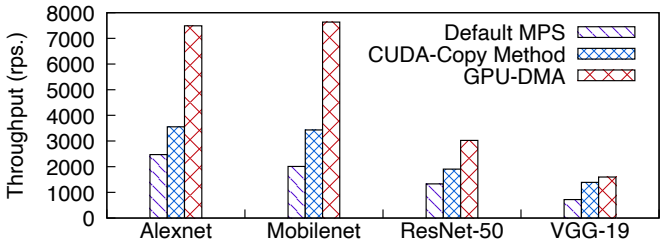


Fig. 11: Use of `GPU-DMA` in 8 NVIDIA T4 GPU cluster.

obtained with Default MPS also suffers because of both a small batch size as well as the overhead of copying the data to GPU Even in a compute-heavy model, `GPU-DMA` produces 1.5 × throughput in ResNet-50 and 1.2× the throughput of VGG-19 compared to the CUDA-copy method. Since these compute-heavy models spend more time inferring each batch, the relative impact of `GPU-DMA` is lower than with models such as Alexnet that run for a shorter time. Nonetheless, `GPU-DMA` shows significant improvement in throughput across all the models.

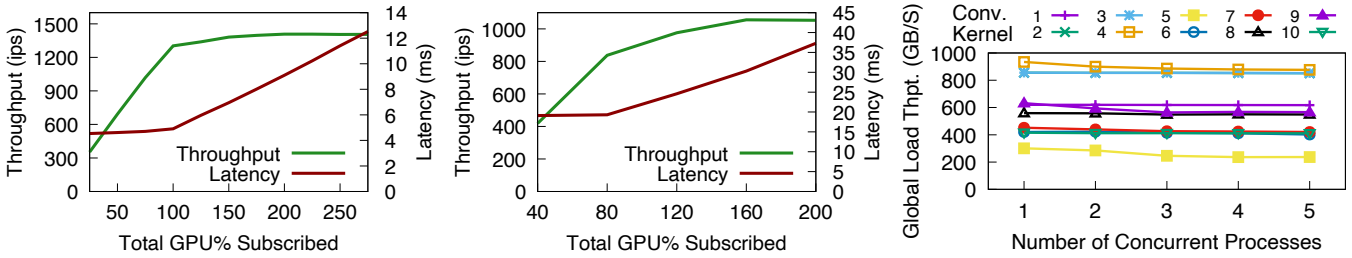
B. *Effectiveness of Creating Multiple DNN Instances*

We evaluated a scenario where we see the throughput and latency when multiplexing several instances of DNN models on the GPU compared to running only one DNN model per GPU. We ran four different models, Alexnet, Mobilenet, ResNet-50, and VGG-19, in our multi-GPU (NVIDIA T4 GPU) cluster. For the baseline, we only ran one model per GPU and sent a request rate of 3800 images per second (Max. possible with 20 Gbps in our system). We set a tight deadline (SLO) of 20 milliseconds for Mobilenet, which is optimized for low latency inference. We set a deadline of 30 milliseconds for ResNet-50, which is optimized for higher accuracy than Mobilenet but also incurs higher latency. We use an SLO of 50 milliseconds for VGG-19, which is optimized for very high accuracy without regard for latency. We utilize our *`Multi-Model`* primitive to determine the number of instances of each DNN that can run in a GPU. This enables getting higher throughput while still meeting the deadline. *`Multi-Model`* suggested running 4 instances of Mobilenet and Alexnet with 25% GPU, 2 instances of ResNet-50 with 50% GPU each and 2 instances of VGG-19 with 50% GPU each. We compared the latency and throughput with an option of running a single instance with 100% of GPU. We utilize `GPU-DMA` to transfer images for both cases. We present the results in Table. IV.

We observe that running multiple DNN instances increases the throughput of all models by nearly 2×. Moreover, using the GPU% suggested by our primitive allows us to fulfill the SLOs set for each model. Giving the entire GPU to a single model may allow for bigger batch sizes. But, the DNN model is also limited by how much GPU it can use. Having multiple models with a smaller portion of the GPU for each allows the GPU to be utilized more efficiently, thereby achieving higher throughput.

C. *Effect of Oversubscribing the GPU*

We conduct an experiment to determine the effect of oversubscribing the GPU, *i.e.,* running multiple models concurrently



(a) Alexnet (b) ResNet-50 (c) VGG-19 Kernels' memory thoroughput

Fig. 12: GPU over-subscription (a) Alexnet (batch = 8) and (b) ResNet-50 (batch = 8) (c) memory throughput

TABLE IV: Throughput & Latency with NVIDIA T4 GPUs

Model	One Model Per GPU		Multiple DNN Instances in one GPU		
	Latency (ms)	Thrpt. (Ips)	# inst.	Latency (ms)	Thrpt.
Mobilenet	18.8	1218	4	19.77	2310
Alexnet	16.9	1656	4	18.9	2539
ResNet-50	25.7	155	2	29.1	274
VGG-19	45.1	88	2	49.1	162

with the aggregate GPU% exceeding 100%, on the inference throughput and latency. These are shown for multiple instances of Alexnet (each with 25% GPU) and ResNet-50 (each with 40% GPU), with the total exceeding 100% GPU, in Fig. 12a and Fig 12b, respectively. Both show that the throughput does not increase when the GPU allocation goes beyond 100%, but the latency climbs rapidly. We see similar trends with other models, *i.e.,* providing beyond 100% of GPU does not improve throughput but only increases latency. Therefore, we do not oversubscribe the GPU while multiplexing with CSS.

D. Memory Impact of GPU Multiplexing with CSS

We profiled several DNNs when they are multiplexed in the GPU using CSS to check if the DNN models were memory-bound, *i.e.,* if concurrently running multiple DNN models has a secondary effect (*e.g.,* memory contention) that can impact model inference latency. In this experiment, we increase the number of concurrently running instances of the VGG-19 model from 1 to 5, while simultaneously reducing the GPU% proportionally, *i.e.,* from 100% to 20% for each instance. We measured the memory throughput attained by 10 different convolution kernels of each instance of VGG-19. We present the average memory throughput in Fig. 12c. The global memory throughput is for reading from the GPU main memory (GPU RAM).

As the number of concurrently multiplexed DNN kernels increases, the reduction in global (GPU main memory/GPU RAM) memory loading throughput is marginal for most kernels (less than 5%). The exception is kernel 5, which sees a more significant reduction, a little more than 20%, when the number of DNNs increases from 1 to 5. While we do observe this brief memory contention, most DNN models we have studied do not appear to have a significant reduction in inference throughput. Overall, the benefit from CSS for DNNs outweighs any concerns regarding memory bandwidth contention within the GPU.

VII. Conclusion

With the growing use of compute-heavy DNN applications for real-time inference, offloading their execution to edge-cloud platforms can achieve lower latency. Hardware accelerators in the edge cloud, such as GPUs, are helpful but are expensive and power-hungry. Thus, they may be available only in limited numbers, and it is desirable to multiplex several concurrent DNN applications to effectively utilize the limited GPU resources at the edge cloud. The large amount of CPU-GPU interactions for transferring and inferring requests received over the network introduces considerable inefficiency. We propose and evaluate several software primitives to achieve low latency and high throughput for inference. Our `GPU-DMA` primitive significantly reduces the CPU load and increases the inference throughput up to 2× in an 8 GPU cluster. Our `Sync-lite` primitive provides lightweight and low latency synchronization. `Sync-lite` reduces the inference latency by up to 30% and reduces GPU idle time by up to 40% during inference compared to other notification approaches. Our other software primitive, `Overlapped-Execution`, facilitates spatial sharing of the GPU by enabling the GPU allocation (GPU%) to be changed with minimal (100μs) downtime compared to 2-5 seconds without our primitive.

DNNs often do not utilize entire GPU compute resources. Our `Multi-Model` primitive better utilizes a GPU by giving just enough GPU% to run a DNN model, so to meet the inference SLO. With `Multi-Model`, we see the DNN model's throughput doubles compared to exclusively providing the GPU for a DNN model. To support running multiple models in GPU, our `Param-Share` primitive shares model parameters across model instances, thus fitting 20-30% more models in GPU memory. Finally, our primitive `Batch-Latency` aids spatial and temporal scheduling by providing the right batch size to execute when the GPU is available within a schedule. This primitive aids in increasing the number of models that can run in GPU concurrently, thus increasing the overall system throughput 300% while scheduling 5 DNNs compared to temporal sharing of GPU. Overall, our primitives enable support of more services in an edge cloud and increase inference throughput while also reducing inference latency.

VIII. Acknowledgement

We thank all the anonymous reviewers for their valuable feedback and the US NSF for their generous support of this work through grant CNS-1763929.

REFERENCES

[1] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 401–411.

[2] NVIDIA, "Dgx pod," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/resources/dell/h18597-dell-emc-powerscale-and-nvidia-dgx-a100-systems-for-deep-learning.pdf, online,accessed 15 Auguster 2020.

[3] "Amazon machine learning developer guide," 2021, [ONLINE].

[4] "Automl vision edge," 2021, [ONLINE].

[5] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan, "Gslice: Controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 492–506.

[6] NVIDIA, "Nvidia triton inference server," https://developer.nvidia.com/nvidia-triton-inference-server, 2021.

[7] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 595–610.

[8] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: a gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 322–337.

[9] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 613–627.

[10] T.-A. Yeh, H.-H. Chen, and J. Chou, "Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 173–184.

[11] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving dnns like clockwork: Performance predictability from the bottom up," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 443–462.

[12] W. Zhang, W. Cui, K. Fu, Q. Chen, D. E. Mawhirter, B. Wu, C. Li, and M. Guo, "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 58–68.

[13] NVIDIA, Tesla, "Multi-process service," *NVIDIA. May*, p. 108, 2019.

[14] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, "Machine learning at the edge: Efficient utilization of limited cpu/gpu resources by multiplexing," in *2020 IEEE 28th International Conference on Network Protocols (ICNP)*. IEEE, 2020, pp. 1–6.

[15] ——, "Ecml: Improving efficiency of machine learning in edge clouds," 2020.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[17] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for gpu computing," in *2013 International Conference on Parallel and Distributed Systems*, 2013, pp. 275–282.

[18] T. C. Carroll and P. W. Wong, "An improved abstract gpu model with data transfer," in *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, 2017, pp. 113–120.

[19] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-net: Effective {GPU} sharing in {NFV} systems," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 187–200.

[20] A. Nguyen, Y. Fujii, Y. Iida, T. Azumi, N. Nishio, and S. Kato, "Reducing data copies between gpus and nics," in *2014 IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*. IEEE, 2014, pp. 37–42.

[21] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar, "A server-based approach for predictable gpu access control," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017, pp. 1–10.

[22] J. Choi, D. F. Richards, and L. V. Kale, "Achieving computation-communication overlap with overdecomposition on gpu systems," in *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2020, pp. 1–10.

[23] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*, 2015, pp. 223–234.

[24] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.

[25] P. Jain, X. Mo, A. Jain, H. Subbaraj, R. S. Durrani, A. Tumanov, J. Gonzalez, and I. Stoica, "Dynamic space-time scheduling for gpu inference," *arXiv preprint arXiv:1901.00041*, 2018.

[26] P. Jain, X. Mo, A. Jain, A. Tumanov, J. E. Gonzalez, and I. Stoica, "The ooo vliw jit compiler for gpu inference," *arXiv preprint arXiv:1901.10008*, 2019.

[27] H. Shen, L. Chen *et al.*, "Nexus: a gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 322–337.

[28] T. T. Yeh, M. D. Sinclair, B. M. Beckmann, and T. G. Rogers, "Deadline-aware offloading for high-throughput accelerators," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 479–492.

[29] W. Zhang, Q. Chen, K. Fu, N. Zheng, Z. Huang, J. Leng, C. Li, W. Zheng, and M. Guo, "Towards qos-aware and resource-efficient gpu microservices based on spatial multitasking gpus in datacenters," 2020.

[30] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 681–696, 2016.

[31] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[32] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick, "Mask R-CNN," *CoRR*, vol. abs/1703.06870, 2017. [Online]. Available: http://arxiv.org/abs/1703.06870

[33] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[34] NVIDIA, "Tensorrt developer guide," https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html, 2019, [ONLINE].

[35] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.

[36] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. K. Ramakrishnan, and T. Wood, "OpenNetVM: A Platform for High Performance Network Service Chains," in *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, Aug. 2016.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[38] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[39] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

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

[41] "Nvidia visual profiler user guide," https://docs.nvidia.com/pdf/CUDA_Profiler_Users_Guide.pdf, accessed:2018-12-01.

[42] NVIDIA, "Getting started with cuda graphs," https://developer.nvidia.com/blog/cuda-graphs, 2019.

[43] T. C. Schroeder, "Peer-to-peer and unified virtual addressing," in *GPU Technology Conference, NVIDIA*, 2011.

[44] A. Dhakal and K. K. Ramakrishnan, "Netml: An nfv platform with efficient support for machine learning applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 396–404.
[45] D. Intel, "Data plane development kit," https://dpdk.org/, 2014, accessed: 2020-06-12.
[46] O. M. Parkhi, A. Vedaldi, and A. Zisserman, "Deep face recognition," 2015.
[47] H. Jung, M.-K. Choi, J. Jung, J.-H. Lee, S. Kwon, and W. Y. Jung, "Resnet-based vehicle classification and localization in traffic surveillance systems," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017, pp. 934–940.
[48] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.
[49] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.