D-STACK: High Throughput DNN Inference by Effective Multiplexing and Spatio-Temporal Scheduling of GPUs

Aditya Dhakal, *Member, IEEE*, Sameer G. Kulkarni, and K. K. Ramakrishnan, *Life Fellow, IEEE*

***Abstract*—Hardware accelerators such as GPUs are required for real-time, low latency inference with Deep Neural Networks (DNN). Providing inference services in the cloud can be resource intensive, and effectively utilizing accelerators in the cloud is important. Spatial multiplexing of the GPU, while limiting the GPU resources (GPU%) to each DNN to the right amount, leads to higher GPU utilization and higher inference throughput. Right-sizing the GPU for each DNN the optimal batching of requests to balance throughput and service level objectives (SLOs), and maximizing throughput by appropriately scheduling DNNs are still significant challenges.This article introduces a dynamic and fair spatio-temporal scheduler (D-STACK) for multiple DNNs to run in the GPU concurrently. We develop and validate a model that estimates the parallelism each DNN can utilize and a lightweight optimization formulation to find an efficient batch size for each DNN. Our holistic inference framework provides high throughput while meeting application SLOs. We compare D-STACK with other GPU multiplexing and scheduling methods (e.g., NVIDIA Triton, Clipper, Nexus), using popular DNN models. Our controlled experiments with multiplexing several popular DNN models achieve up to 1.6× improvement in GPU utilization and up to 4× improvement in inference throughput.**

***Index Terms*—Datasets, neural networks, gaze detection, text tagging.**

I. Introduction

DEEP Neural Networks (DNNs) are widely used for many applications, including image recognition, natural language processing, *etc.* Accelerators have become indispensable for DNN learning and inference. Accelerators such as GPUs, TensorCores [1], and TPU [2] reduce the DNN inference times, often by 2-3 orders of magnitude compared to even using a high-end CPU cluster. These accelerators are widely used by cloud services as a part of their *inference-as-a-service* (IaaS) offerings, where trained DNN models are hosted in a Cloud or an Edge Cloud (especially for low-latency operation). User requests are inferred using the GPUs deployed in the cloud.

Most DNN models running in inference frameworks (PyTorch [3], TensorFlow Serving [4], NVIDIA's Triton [5] *etc.*) often execute far fewer floating-point operations per second (FLOPS) than the capacity of these high-end GPUs [6], [7], [8], TPUs [9] and other accelerators [10]. In our previous work [6], we observed that performing inference using DNN models, even using a single GPU, do not significantly reduce the DNN's processing latency when provided with additional GPU resources (i.e., number of Streaming Multiprocessors (SMs) - GPU compute units analogous to CPU cores) beyond a certain point. We call this point as a **"Knee"** for the DNN (expressed as a percentage of the total SMs available in the GPU, e.g., 50% of a V100 GPU (which has 80 SMs in total) is 40 SMs.). Running applications with resources matching the Knee is desirable for a cloud operator providing Inference as a Service, since multiplexing a GPU (or similar accelerator) across as many applications as possible keeps costs low. Operating at the Knee also keeps the latency low for the user. When more GPU resources are provided for a DNN (e.g., by giving the full GPU to an application, possibly using temporal sharing), it is wasteful as the GPU is not fully utilized.

There are three fundamental reasons for the under-utilization of multi-core accelerators, such as GPUs, by DNNs when given more than the Knee's resources: i) Amount of parallelism over the entirety of DNN's execution is not uniform, i.e., many DNN functions (e.g., convolution, ReLU *etc.*) are unable to fully utilize the parallelism offered by the accelerator. Furthermore, memory-bound kernels cannot utilize GPU compute resources fully due to limited memory bandwidth. ii) DNN operations also involve other overheads (e.g., kernel launches, memory read-write, *etc.*). While users and cloud providers can utilize larger batches of DNN operations to be executed concurrently and increase utilization, this comes at the price of increased latency. When the results are needed quickly, to meet a small latency target, such as during inference, increasing the batch size is not an ideal option, and batch sizes have to be limited.

Thus, this may result in insufficient utilization of a GPU's parallelism for many applications.We study the execution of a variety of DNN models to understand the root causes of under-utilization of such accelerators, particularly GPUs, and develop methods to improve the overall system utilization, thus improving throughput and reducing inference latency.

Multiplexing GPUs in the Edge Cloud:

DNN inference requests for applications such as autonomous driving, augmented reality, *etc.*, have stringent deadlines (e.g., $<$ 100 ms). A cloud providing IaaS also has to account for the

Received 30 December 2023; revised 12 September 2024; accepted 22 September 2024. Date of publication 7 October 2024; date of current version 6 December 2024. This work was supported in part by the U.S. NSF under Grant CRI-1823270. Recommended for acceptance by A. Jog. *(Corresponding author: K. K. Ramakrishnan.)*

Aditya Dhakal and K. K. Ramakrishnan are with the University of California, Riverside, Riverside, CA 92521 USA (e-mail: kk@cs.ucr.edu).

Sameer G. Kulkarni is with the IIT Gandhinagar, Gujarat 382355, India.

Digital Object Identifier 10.1109/TCC.2024.3476210

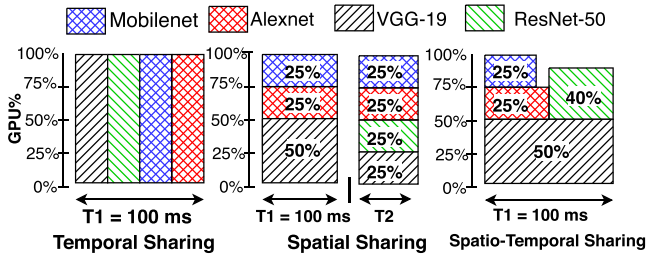


Fig. 1. GPU multiplexing scenarios.

network latency. Edge Clouds offer a sweet spot reducing both latency and offering the necessary processing resources, although more constrained than centralized cloud services. Multiplexing the expensive hardware accelerator is therefore very desirable. Current GPU virtualization and inference service frameworks such as Nexus [11], NVIDIA's Triton Inference Server (Triton) [5], gPipe [12], and PipeDream [13] either use a 'single GPU per DNN' model or time-share the GPU across multiple DNN models. These current state-of-the-art frameworks for DNNs allocate the full GPU (i.e., 100% of GPU) for the time quantum as shown in Fig. 1(left).

However, dedicating an entire GPU to run a single DNN model at a time can be wasteful. Furthermore, interleaving execution of tenant applications by temporally sharing increases inference latency for all of them, because of the significant cost of frequent switching between applications. Multiplexing several applications on the GPU to run concurrently, through spatial as well as temporal multiplexing, helps to better utilize the GPU and achieve much higher aggregate inference throughput.

Our approach utilizes the CUDA Multi-process Service (MPS) [14] to spatially share the GPU across several applications. We build on top of our earlier GSLICE [6] work. Existing approaches of spatial multiplexing with the GPU either only statically partition the GPU for each application or does not guarantee computing resource isolation while multiplexing. This has the potential to allocate fewer resources than necessary for an application. It also causes interference among the multiplexed applications when too many models share the GPU, thus, increasing the inference latency.

We illustrate with an example when four different models have to be run on a V100 GPU (three are already executing and a fourth is added). Temporal sharing allocates the GPU to each model for a time slice. Static spatial sharing with CUDA-MPS will allow all 4 models to run in an uncontrolled manner, causing interference as noted in [6]. GSLICE will initially spatially share the 3 models, and allocate GPU resources according to their Knee GPU% capacities. When the fourth model is added (in Fig. 1(middle)), the VGG-19 model's GPU% is reduced from 50% to 25%, causing increased inference latency for that more complex VGG-19 model, which also is undesirable.

On the other hand, our GPU virtualization framework, with our spatio-temporal scheduler, **Dynamic Spatio-Temporal pACK** (D-STACK), can run on multiple NVIDIA GPU-based systems (single GPU or GPU clusters). **D-STACK** schedules DNNs based on spatial resources (Knee GPU%, number of SMs), and the appropriate time slice. Combining spatial and temporal scheduling, D-STACK is designed to meet the inference deadline for each DNN model. D-STACK goes well beyond the basic idea of simple temporal or static spatial multiplexing of a GPU presented in earlier works [5], [6], [8]. The example of Spatio-Temporal scheduling in Fig. 1(right), has all 4 models getting their Knee GPU%. When a model completes its inference, another model utilizes the GPU resources, thus, sharing the GPU resources both temporally and spatially. D-STACK's scheduler further utilizes the idle processing resource of the GPU by dynamically running any 'ready' models, thus maximizing GPU utilization.

D-STACK's Innovations:

i.) Understanding a DNN's demand: For efficient utilization of the GPU, D-STACK requires information about the resource requirements of each DNN model. Providing the right resources for the DNN is not just a challenge for the GPU, but is fundamental for all such accelerators that utilize a multitude of compute engines for parallel processing. In this paper, along with our analytical models of DNN execution and scheduling, we estimate what would be theoretically possible for a DNN to exploit available parallelism by knowing exactly how much computational capacity is required, assuming that instantaneous switching between multiplexed tasks is possible. We then show how close we come to that theoretical optimal by implementing our GPU virtualization framework using our D-STACK scheduler on a GPU cluster.

ii.) Dynamic Resource Allocation in GPU: Currently, dynamic resource allocation of the GPU requires reloading of applications with their new desired GPU%. For typical DNN models, this reloading time can be 10s of seconds, during which the GPU is idle, lowering the overall system utilization and throughput. In D-STACK, we address the dynamic allocation of GPU resources by overlapping the loading of a DNN model with the new resource allocation, by continuing to execute the existing DNN model, thus effectively masking the loading latency. We thus reduce the time the GPU is idle to less than 100 micro-seconds with D-STACK.

iii.) Multi-GPU Cluster: Understanding the use of a single GPU and increasing its utilization translates to improving overall throughput of a GPU cluster. D-STACK's optimization can be easily extended to a multi-GPU cluster. In this paper we present the implementation of D-STACK's Spatio-temporal scheduler across multi-GPU cluster to increase the system throughput by 200%.

Comparing with State-of-the-art: We present a comparison of D-STACK with NVIDIA's Triton Inference Server. We evaluate the total time taken to infer with 4 different DNN models, Alexnet, Mobilenet, ResNet-50, and VGG-19 being multiplexed on one V100 GPU, each concurrently inferring 10000 images each. The results in Table I show that the Triton server takes about 58 seconds to finish inference. The D-STACK scheduler completes inference on all requests more than 37% faster (only **36** seconds). D-STACK's spatial multiplexing, providing just the right amount of GPU% and its dynamic spatio-temporal scheduling results in more effective use of the GPU and achieving higher DNN inference throughput than NVIDIA's Triton

TABLE I
TRITON AND D-STACK WITH 4 DNN MODELS

	Triton Server	D-STACK	Latency Reduction(%)
Task completion (sec.)	58.61	35.59	37%

server, while also lowering task completion time. Based on these experiments, we see that implementation of Spatio-temporal scheduling can further enhance throughput when inferring with multiple different models concurrently.

Contributions: D-STACK improves GPU utilization by 60% and increases in DNN inference throughput by 4× compared to a pure temporal scheduler, while still avoiding any deadline (SLO) violations. Our key contributions are:

- We investigate the extent to which a DNN can exploit parallelism (Section III), and devise an analytical model to demonstrate this limitation of typical DNNs when performing inference with GPUs (Section IV).
- We develop an optimization framework to determine the optimal DNN Batch size and GPU%. We evaluate the efficacy of GPU usage when choosing the optimal batch size and Knee GPU%. (Section V).
- We develop a Spatio-Temporal scheduler for DNNs, using the GPU% and batch size derived from our analytical models, to maximize inference throughput while allocating GPU resources fairly (Section VI).
- We compare D-STACK's approach with the Triton server and other state-of-the-art scheduling algorithms.
- We present results of D-STACK in multi-GPU cluster.(Section VII-A).

II. RELATED WORK

GPU Multiplexing: Multiplexing GPU to increase the GPU utilization and system throughput has been discussed in many studies. Proprietary products such as Nutanix [15], vGPU [16] utilize GPU virtualization to multiplex GPU across VMs. Many consider temporal multiplexing and seek increased GPU utilization through batching and better scheduling [11], [17], [18], [19], [20], [21], [22]. Gandiva [23] and Mystic [24] address multiplexing the GPU while observing but not solving the interference caused while multiplexing DNNs in the GPU. Unlike these, our workcan concurrently run multiple applications in GPU, improve GPU utilization *and* reduce or eliminate the interference through controlled spatial multiplexing.

Spatial Multiplexing of GPU: GSLICE [6] utilizes CUDA MPS to spatially share the GPU among multiple DNN applications. However, it partitions the GPU statically and does not schedule the execution of DNNs. With GSLICE, executing a large number of models potentially cause each model get a small GPU slice (less than the Knee), leading to higher inference latency and lower throughput. However, D-STACK uses a dynamic spatio-temporal scheduler compared to GSLICE's static spatial-sharing. GSLICE only looks at the initial resource requirements for each application while determining which applications should run together. Moreover, the lack of a scheduler means it is insufficient for deadline-driven inference scenarios. While, D-STACK can schedule work once the previous application ends and resources free up, thus, increasing the GPU hardware utilization. We compare D-STACK's performance with GSLICE in Section VII.

Laius [7], G-Net [25], Gost [26] and Baymax [27] spatially multiplex GPU kernels. Unlike these works, our platform focuses on the spatially multiplex entire DNNs consisting of multiple kernels. Moreover, we run DNN applications in their native DNN framework (e.g., PyTorch, TensorFlow) without any algorithmic modifications, unlike the whitebox approach of Laius and Baymax. S3DNN [28] (uses Streams) and Prophet [29] (uses MPS) and CuMAS [30] profile each kernel and use a shim to capture kernel launches and reorder kernel executions for proper spatial sharing. In contrast, our approach does not require a shim or reordering of kernels and works in a black box manner, without requiring an application's individual kernel profile (which may not be available).

SMGuard [31] calculates the number of GPU threads each kernel requires, captures the kernel when launched, and multiplexes the GPU by running kernels concurrently without exceeding the number of GPU threads each SM can run concurrently. Similarly, Qos Aware dynamic resource allocation [32] utilizes a kernel transformer that changes the GPU code to implement QoS policies. D-STACK allows applications to run as is without modification, while SMGuard and others need a kernel capture mechanism, which also brings additional privacy concerns. Zhao et al. [33] utilize a classification-driven technique (CD-Search) to classify applications as memory-intensive or compute-intensive and place compute-intensive and memory-intensive workloads together for higher overall performance/throughput. Applications are classified based on the use of SM's memory when running the applications. CD-Search enforces partitioning by occupying the SMs with dormant/sleep kernels and releasing them when required by an application. The spatial-sharing of a number of SMs for memory-intensive applications is determined by gradually stalling/decreasing SMs to find the right number to run for appropriate sharing of the GPU. We have the same goal in D-STACK to share the GPU through multiplexing. We profile applications to find the appropriate number of SMs needed by evaluating application performance for a range of GPU%. One main difference between the other approaches and D-STACK is that D-STACK uses CUDA MPS, which makes it much easier to implement spatial multiplexing as it only requires changing the environmental variables of the application.

DNN's limits on Utilizing GPUs: Several works [34], [35], [36] have discussed the under-utilization of GPU by DNNs, and have proposed algorithmic optimizations that make DNN kernel computation more efficient [37], [38], [39], [40]. These solutions require whitebox models that can be changed. There have been works analyzing how DNN's exploit parallelism. [41], [42] show that DNNs attain a much smaller number of FLOPS than what a GPU can provide. Poise [43] and [44] shows that the high data load latency from the GPU memory to the processing unit is also a reason for the limit in parallelism. [45] creates an analytical model to predict the inference latency and mainly utilize temporal queuing solution to meet deadlines. [45]'s model uses default MPS, and due to interference causing increased latency,

they limit the number of models spatially sharing the GPU at a time. On the other hand, D-STACK provides fine-grained spatial and temporal control of resources of the GPU and thus is able to run far more models with larger batch sizes without interference. With a spatio-temporal scheduler D-STACK utilizes resources both spatially and temporally to meet the inference deadline. [8] shows lack of resources in CPU and GPU spatial resources will greatly slowdown GPU execution. Our work complements [8] by demonstrating a method to find the Knee beyond which applications fail to utilize GPU efficiently. We utilize understanding from these related work to create an analytical DNN model that helps deriving the Knee% necessary for inference without slowdowns. Furthermore, we evaluate our methods in a real system.

Multi-Instance GPUs (MIGs) such as the NVIDIA A100 are hardware-based approaches for coarser-grained, spatial multiplexing. MIGs allow static partitioning of a GPU into multiple smaller GPU instances (up to 7 instances with the A100). However, MIGs require the GPU to be reset or VMs to be restarted to change the resource allocation. This causes significant downtimes as all the processing using the GPU has to be restarted. D-STACK's spatio-temporal scheduling avoids the GPU reset and quickly allocates the desired GPU resources. Moreover, note that A100 and H100 are also able to run MPS (similar to V100). Thus, they can benefit from D-STACK without any modification.

III. Understanding DNN Parallelism Through Measurement

Experimental Setup and Testbed:

We used a Dell Server with Intel(R) Xeon(R) Gold 6148 CPU with 20 cores, 256 GB of system memory, and one NVIDIA V100 GPU, and an Intel X 710 10 GbE NIC as our testbed. The V100 has 80 SMs and 16 GB of memory. Our workload for the vision based DNNs (Alexnet [46], Mobilenet [47], ResNets [48], VGG [49], Inception [50], ResNext [51]) consists of color images of resolution 224 × 224. This resolution choice is inspired by initial work [49], [52], [53]. For BERT [54], a natural language processing DNN, we utilize sentences of 10 words.

We use OpenNetVM [55] to host our framework that runs multiple DNN models for inference. We use Moongen [56] to transmit ˜1920 images/sec. on a 10 Gbps Ethernet link. Our platform can batch input data to the desired batch size. We primarily report the execution time for inference in the GPU for all our experiments and do not consider the additional latency contributed by network protocols. Therefore, our results are independent of the network transport protocol used. We utilize CUDA Multi-Process Service (MPS) to spatially multiplex the GPU. We use CUDA_MPS_ACTIVE_ THREAD_PERCENTAGE environmental variable to provide GPU%. Once set, the GPU% cannot be changed for a process.

A. Finding the Knee

We profile the models to find the knee. If there are no time constraints, then we usually collect latency for 10 different GPU configurations (in 10% increment), each with 3 batch sizes.

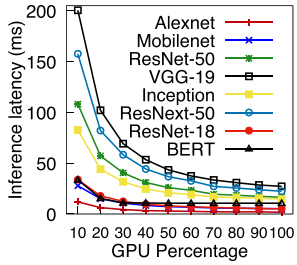


Fig. 2. V100 latency versus GPU% (Batch of 16 images/sentences).

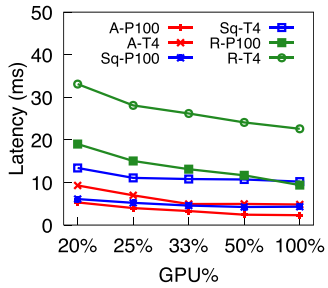


Fig. 3. P100 and T4 GPUs profile.

Thus, 30 different runs for each application to form a profile. For applications that we cannot afford to run many times, we cut the time to find the knee by looking at the latency of the application's execution when GPU resources are cut by half (50%, 25%, and 12.5%) in subsequent execution.

Furthermore, we profile the workload as a whole to find the knee. We chose to find one knee for a workload as the reconfigurations of MPS/MIGs are not fast enough to partition GPU for each kernel. Thus, providing knee value for each kernel would add a large amount of latency.

B. Measurement With ML Models

We now present measurements performed on our testbed with multiple DNNs, to demonstrate the limits in the parallelism of those DNN models. We measured the latency for inferring a batch of 16 images/sentences using different GPU% for several popular DNN models using PyTorch framework. We utilize models with different compute requirements.

From Fig. 2, we see that the inference latency remains unchanged above 30-50% of GPU for most models (Knee point). With a smaller batch size, the Knee% is lower (20%-35%). However, we also observe that using fewer than necessary SMs (low GPU%) leads to an exponential increase in model latency (also observed in [8]). We observed a similar knee with other GPUs as well. We evaluated computationally light models, Alexnet (A-P100 and A-T4) and Squeezenet (Sq-P100 and Sq-T4) on both the P100 and T4 GPUs. The T4 GPU supports CUDA MPS with a GPU%, but the P100 only supports the default MPS without being able to define a GPU%. We present their results in Fig. 3. Even with different GPUs, we see the knee behavior in Alexnet and Squeezenet. Only the computationally dense ResNet-50 (R-P100 and R-T4) does not show an obvious

knee. Both the P100 and T4 GPUs have lower computational capacity than the V100, therefore, ResNet-50 can fully utilize those GPUs. As the knee for these models exists in other GPUs as well, our platform can be used more generally in other GPUs as well.

C. Dynamic GPU Resource Reconfiguration

Due to the limitation of CUDA MPS [14], any GPU resource readjustment requires us to spin up a new CPU process with an updated GPU%. This results in several seconds of downtime (depending on the ML framework initialization). We utilize the overlapped execution approach of GSLICE [6], which maintain an *active-standby* pair of process, where an active process keeps processing incoming requests while a standby process loads the DNN model into the GPU with updated GPU%. The standby takes over inference when ready, thus, avoiding downtime.

While changing the GPU%, two instances of the same model, the original and the new model, occupy the GPU during the brief overlap time. This increases the GPU memory demand. We overcome this drawback through DNN parameter sharing utilized in GSLICE [6]. We use cudaIPC to share the weights and parameters loaded by the original model with the new loading model, thus removing the need to load the weights again. Parameter sharing reduces the memory required by the newly loaded DNN model by up to 40%.

D. Loading Models Without a Known Knee%

When a model that is not profiled and whose knee is not known is started, our platform initially provides it a nominal, 30%, GPU. The GPU% is then readjusted using Dynamic GPU resource reconfiguration to find the knee based on the inference latency using a simple binary search.

IV. Modeling DNN Parallelism

A. Compute Bound versus Memory Bound Workloads

The latency of accessing parameters and weights of the DNN layer from the GPU DRAM can be significant. Many studies [57] have suggested that memory-bound DNN kernels may have a small amount of compute and are likely to be limited by GPU memory bandwidth. NVIDIA has proposed an *arithmetic intensity* (A.int) metric [58] to estimate if a kernel is memory or compute bound. The *A. int* of a kernel is computed as a ratio of floating point operations to memory (bytes) it fetched. i.e., $A.int = \frac{\#operations}{\#bytes}$. NVIDIA reports the arithmetic index of V100 GPU (in our testbed) is 139.8 *FLOPS/Byte* [58]. Any kernel lower than the GPU's arithmetic index is memory-bound, while a kernel with higher index is compute-bound.

We analyzed the most frequently occurring kernels of CNNs Alexnet [52], ResNet-50 [48], VGG-19 [49], and an RNN, GNMT [59], to illustrate the behavior of compute and memory-bound DNNs. We present the results in Table. II. Most convolution layers exceed the GPU's A.int, thus, are compute-bound. These layers can reduce their runtimes if more compute is available. However, kernels like LSTM in GNMT, which operate with large input and output features (1,024 features in GNMT), require a lot of data but perform relatively fewer computations compared to convolution. Therefore, they score very low A.int. We should note that DNNs are not entirely constructed of convolution or LSTM layers. However, CNNs, in general, have more convolution kernels.

TABLE II
Compute & Memory Bound Kernels

Model	Layer	GFLOPs	Bytes (10^6)	Arit. Int.	Limit
Alexnet	Conv.2	0.30	0.22	182	Compute
ResNet50	Conv.2	0.103	0.121	393	Compute
VGG-19	Conv.11	3.7	9.44	391	Compute
GNMT	LSTM	0.016	8.38	2	Memory

TABLE III
Latency (ms) in Isolation and Multiplexed

Model	Knee%	Isolation	Multiplexed
Mobilenet	20%	9.8 (ms)	9.9
ResNet-18	30%	12.4	12.4
BERT	30%	9.3	9.3
ResNet-50	40%	28.9	28.5
VGG-19	50%	51.2	52.4

B. Understanding Memory Contention While Multiplexing

Studies [60], [61] of scientific computation workloads have shown that the GPU cache size and occupancy are important factors influencing the latency of kernel execution. We also examine the effect of cache contention while running multiple DNN models. However, we observe with DNNs, that the inference latency does not vary significantly *if* SM isolation is maintained. Since we indeed maintain SM isolation with spatial multiplexing using CUDA MPS, the impacts of contention in the GPU cache or other memory resources is minimal.

In Table III we have evaluated different DNN workloads in isolation as well as when they are multiplexed on the GPU. The intent of the experiment is to observe if there is any slowdown due to memory or other constraints, when multiplexing them on the GPU. As we see in Table III, multiplexing DNNs till we 'fill up' the GPU compute capability to 100% does not affect the final inference latency at all.

Our experiment compares the runtime with five different DNN applications running concurrently, with each application running with 20% GPU versus a single application running with 20% GPU, while the other 80% of the GPU is unused. We use the experiment to show that CUDA MPS, and D-STACK which is built on top of CUDA MPS, isolate the GPU resources appropriately. MPS enforces SM-level isolation so that SMs are not shared between applications, as long as all partitions add up to 100% or less. Thus, a task running in one partition does not affect other tasks in other partitions. Therefore, running 1 task alone with a 20% GPU partition and 5 tasks concurrently, each with 20% GPU partitions, have similar latency. We should note that Mobilenet, ResNet, and VGG have more compute-bound kernels whose performance can be easily isolated with CUDA MPS. Thus, their performance scales with multiple instances

TABLE IV
TABLE OF NOTATIONS FOR DNN MODEL

Variable	Description
b	Batch Size
p	1st kernel's number of concurrent ops. (tasks)
$Kmax$	Maximum number of kernels
K_i	i^{th} kernel
N_i	Number of parallelizable operations for K_i
R_i	Number of repetition of K_i in DNN
M	Memory Bandwidth per SM
d_i	Data for i^{th} kernel (parameters & input)
S	Number of allocated SMs

running together. BERT and applications such as large language models (LLMs) have several kernels that are memory bound as well as features that require compute-bound kernels. D-STACK helps multiplex these applications by isolating compute-bound kernels so that they do not affect each other, while still providing enough resources for memory-bound kernels. Thus, D-STACK is beneficial when using workloads with a mix of compute and memory bound kernels.

C. Modeling DNNs

We now model an analytical DNN model that exhibits the characteristics of most actual DNN models, in terms of the variation in the compute workload across their different kernels. We model the DNN composed of multiple sequential *kernels* executing in GPU (and other accelerators) instead of *layers* as often used in other ML studies. We have observed using NVPROF profiling that each layer (e.g., convolution layer) is often implemented as combination of multiple kernels in GPU, thus, we use kernel as basic component of DNN execution in this model. The model guides the determination of the best operating point (Knee) GPU% for a DNN. In our model, we breakdown the DNN workload into parallelizable operations (compute tasks), memory read/write as well as serialized (non-parallelizable) operations, and observe the effect of changing GPU resources. While our model is simple, it captures all the system level overheads that contributes to DNN latency, and provides us with good approximation of the Knee of each model. The simplicity of the model further aids in evaluating DNNs in different GPUs, with different numbers of SMs, as well as other accelerator hardware.

Selected notation used in the analysis is shown in Table IV. As in typical GPUs, each of the $\mathbf{S}$ SMs allocated to a DNN will process one parallel operation per $\mathbf{t_p}$ time. From a modeling perspective, we order the kernels by their amount of computation without losing generality. DNNs have an arbitrary order in kernel execution. However, the knee of the model is dependent on peak computation requirements of the kernels rather than the order of execution of each kernel.

We set the first kernel $\mathbf{K_1}$ as that with the greatest amount of parallelizable operations $\mathbf{N_1}$, which is selected as $N_1 = \mathbf{p}$ for modeling purposes. For subsequent kernels, the workload decreases by a fixed amount, so that $\mathbf{N_i} > \mathbf{N_{i+1}}$. Equation (1) specifies the amount of parallelizable operations for each kernel in the DNN. We decrease the amount of parallelizable tasks by a fixed amount, $\frac{p \times b}{Kmax}$,

$$N_i = \begin{cases} p \times b, & i = 1 \\ \lfloor N_{i-1} - \frac{p \times b}{Kmax} \rfloor, & i \geq 2 \end{cases}, \tag{1}$$

for each subsequent kernel. The number of concurrent operations decrease and reaches ~ 0 for the last ($K_{\max}$) kernel. Correspondingly, we define the total execution time for each kernel's parallelizable tasks as $\mathbf{W_i} = N_i \times t_p$.

Note: Ideally, W_i can potentially be completed in t_p units of time when we allocate greater than or equal to the N_i SMs to execute W_i. If we consider that the GPU hardware is able to provide S SMs to execute K_i, then, without loss of generality, we can show that the time taken to finish processing the kernel would depend on the minimum of the inherent parallelism, as defined by N_i, and the number of SMs allocated for executing the operation. Thus, the execution time for parallelizable operations at each kernel of the DNN can be computed using (2). Individual kernels

$$E_i = \frac{W_i}{\max(1, \min(S, N_i))}, \tag{2}$$

in the DNN often run repeatedly during a DNN inference. We define the number of repetitions of kernel K_i as $\mathbf{R_i}$. We then factor the time taken to run all the serialized operations, including for kernel starting and kernel waiting for data. The kernel starting time is considered a constant, $\mathbf{t_{np}}$, per layer. The kernel's time waiting for data, however, depends on the kernel's input and parameters. Each kernel of a DNN has a certain amount of data (model parameters, input data) that has to be fetched from GPU DRAM (main/global memory of GPU) to the CUDA cores in the SMs. We have observed that the total global memory read/write bandwidth increases with the proportion to the number of SMs allocated. Other studies [62], [63] also point to a proportional increase.We define the latency per kernel, caused by kernel waiting for parameters, input, and other data to be loaded, as (3). Thus, we can define the total time of non-parallelizable (sequential) operations $\mathbf{W_{se}}$ as (4). We use (2) and (4) to compute DNN execution time, $\mathbf{E_t}$ as in (5).

$$E_m = \frac{d_i \times S}{M} \tag{3}$$

$$W_{se} = b \times \sum_{i=1}^{K_{\max}} R_i \times (t_{np} + E_m) \tag{4}$$

$$E_t = W_{se} + \sum_{i=1}^{K_{\max}} R_i E_i. \tag{5}$$

We now simulate the total time to execute a DNN under varying conditions i.e., by varying the amount of parallelizable and non-parallelizable operations at each kernel and the number of SMs in the GPU. As in typical GPUs, we assume the number of SMs allocated for an DNN remains static. Fig. 4(a) shows the impact on the DNN execution time when assigning different numbers of SMs. First, we created a DNN with 50 kernels i.e., $K_{\max} = 50$. We set the time taken for the parallel operation t_p

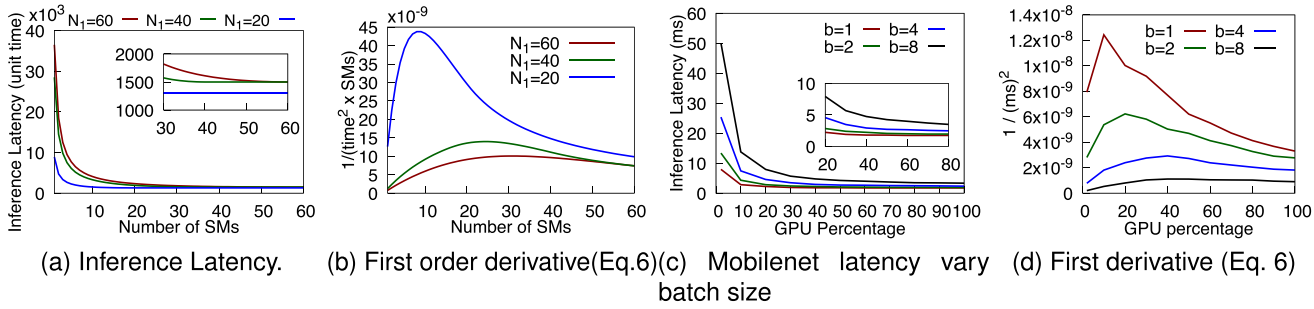


Fig. 4. (a), (b) Inference characteristics of analytical DNN models with varying amounts of parallelism and hardware resources.(c), (d) Demonstration of analytical model's understanding on real DNN Mobilenet.

to be 40 units and for serialized operations t_{np} to be 10 units. We repeat the simulation for 3 cases, varying the maximum amount of parallelization (concurrent operations at the first kernel) N_1 as 60, 40, and 20.

For all three cases, the execution time is very high when the number of SMs is small (1 to 5 SMs), reflecting the penalty of insufficient resources for the inherent degree of parallelism while executing the DNN kernel. However, as the number of SMs increases, the execution latency decreases. Interestingly (see zoomed part of Fig. 4(a)), there occurs a point when giving more SMs beyond a point does not improve latency further, in each of the scenarios. When the number of SMs provisioned exceeds the amount of parallelism inherent in the DNN kernel, there is no further reduction in the latency. Even before reaching this point, the latency improvements from having an increased number of SMs reaches a point of diminishing returns.[1] We seek to find the most efficient number of SMs (S) needed for executing a given DNN, so that the utilization of the allocated SMs is maximized. To compute this, we have to find the maximum of $\frac{1}{E_t * S}$, which represents the DNN work processed per unit time per SM. For this, we differentiate $\frac{1}{E_t * S}$ with respect to the time taken to execute the DNN.

$$\frac{d}{dE_t}\left(\frac{1}{E_t * S}\right) = -\frac{1}{(E_t)^2 * S}. \tag{6}$$

Fig. 4(b) shows this first order derivative of the inverse of latency (6), showing that SMs for $N_1 = 20, 40$ and 60 reaches a maximum at 9, 24 and 31 SMs respectively. Hence, operating at this derived 'maximum' point for a DNN guarantees that there are sufficient number of SMs to provide low latency while achieving the most efficient use of the SMs. Moreover, we can see from this that the 'maximum' peaks at a much lower SMs than the corresponding value of N_1. This is due to the impact of performing serialized tasks adjacent to the parallelizable tasks. This results in lower (or no) utilization of many of the allocated SMs for the serialized tasks. Thus, further reduction in latency by increasing SMs is minimal.

[1] i.e., showing marginal improvements. The DNN execution latency is impacted by both the number of parallelizable and non-parallelizable operations and it varies inversely with the number of allocated SMs, by Amdhal's law [64]. Batching increases parallelizable work [65].

D. Analyzing Execution of Typical DNNs

We profiled and analyzed Mobilenet, ResNet and GNMT DNNs using the NVPROF profiler [66] to capture the GPU resource usage and the execution time of the DNN kernels.

1) CNN Model: Mobilenet: We profiled the inference of Mobilenet using 100% of a V100 GPU. For each kernel, we show the GPU thread count on the y-axis (in log scale) and the corresponding runtime as the area of the bubble in Fig. 5. The approximate GPU% required for all the threads to run concurrently is on Y2-axis (log scale, on the right). We approximate this GPU% by considering that only 2048 threads can run in an SM concurrently, due to limits on the number of concurrent blocks and warps [67]. The kernel's design and thread distribution across different threadblocks can lead to a higher SM demand than absolutely required.

We plot 11 distinct kernels of a Mobilenet model (each identified by a different color in Fig. 5). These kernels are executed a total of 156 times per inference. We observe that few of the kernels (kernel 3, 4 and 6, in particular) require more than 100% of the GPU to run. These kernels demand more threads than a GPU can run concurrently. However, these kernels run for a very short time and do not contribute significantly to the total inference latency. The kernels that contribute more to the total latency, such as kernels 10 and 7 utilize less than 10% of the GPU. This is due to the fact that the DNN's inference feature matrix gets smaller, thus, resulting in limiting the inherent parallelism. Thus, these kernels use fewer parallel GPU threads and run for long time with low GPU% demand. They contribute to lowering the Knee GPU% of the entire DNN model. From this understanding, when the amount of parallelism of a kernel is low, increasing the number of GPU SMs will not reduce the execution time of the kernel, since the additional SMs will not be utilized.

We also analyzed the inference time with different batch sizes of Mobilenet (Fig. 4(c)). In all the cases, for a given batch size, the latency reduces with an increase in GPU%. But, across all evaluated GPU percentages, the latency *increases* with increasing batch sizes. Fig. 4(d) shows the first derivative of the inverse of Mobilenet's latency obtained using (6). The maximum of the derivative, i.e., the most efficient point for DNN operation, for batch sizes of 1, 2, 4 and 8 occurs at GPU% of $\sim$ 10, 20, 40, and 50 respectively. This shows that with

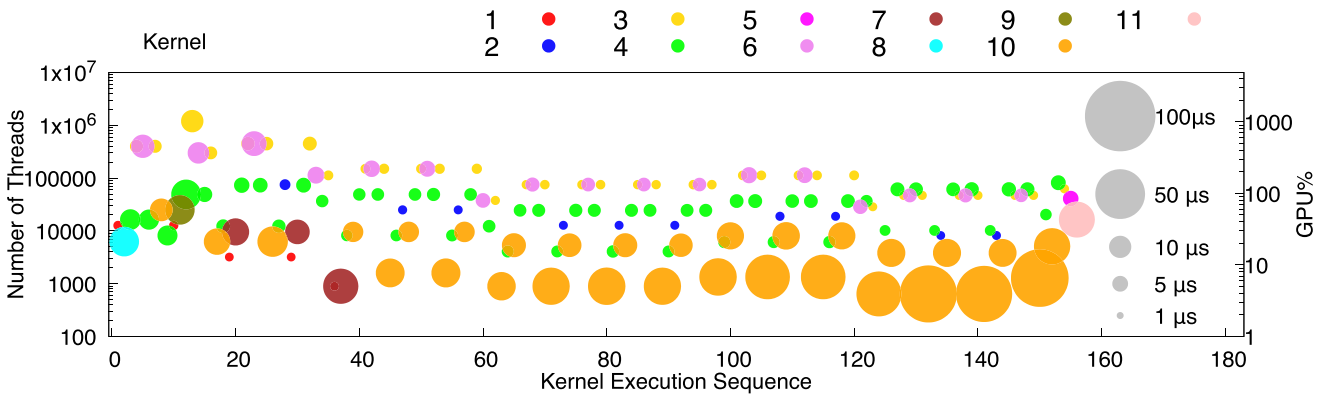


Fig. 5. Thread count & runtime (shown as area of circle) of 156 kernel of Mobilenet. Each colored circle labeled 1-11 represents a kernel (e.g., convolution kernel, ReLU, fully connected). The area of the circle represents the time it takes to run in GPU and left Y-axis represents the number of GPU threads each kernel uses. Right Y-axis represents how much GPU% a kernel will utilize with all its threads. The kernels in the left run earlier than kernels in right.

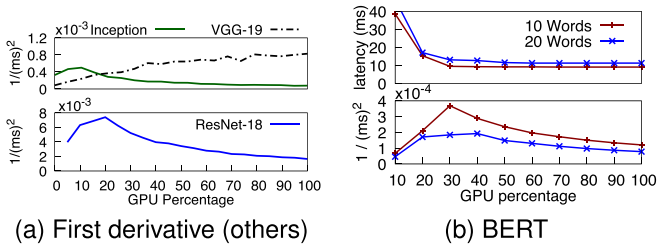


Fig. 6. DNN Latency, first derivative as in (6).

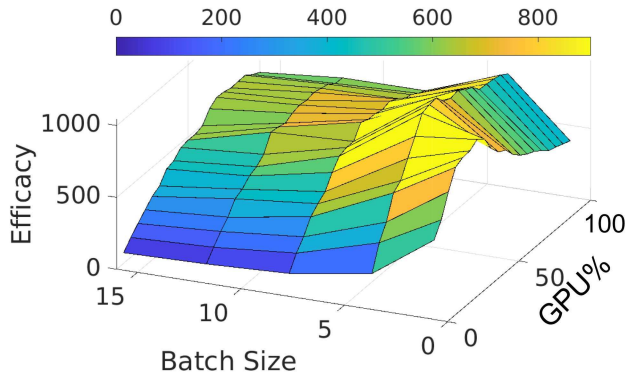


Fig. 7. *Efficacy* of ResNet-50.

increasing batch size, i.e., increased parallelism, the GPU% at which the maximum utilization point occurs, based on (6), also increases. Fig. 6(a) shows the different maximum utilization points for the different models. Lightweight models such as Inception and ResNet-18 have a maximum at a lower GPU%, while compute-heavy VGG-19 does not see an inflection point up to 100% GPU. These characteristics of the individual DNN's execution strongly correlate and match with the theoretical DNN model we presented.

2) Transformer Model BERT: We also present the evaluation of the inference latency for the transformer-based natural language processing DNN, BERT, as well as the first order derivative, per GPU% in Fig. 6(b). We evaluated sentences with 10 and 20 words. We can observe that longer sentences results in higher inference latency. But again, we see that the inference latency does not improve after a point. The first order derivative of the latency for 10 and 20 word sentences shows a peak at around 30% and 40% GPU respectively. Thus, both our model prediction and our evaluation of representative compute-heavy CNN and memory-bound Transformer models show that there is indeed a limit to parallelism utilized by DNNs. This motivates our approach to further examine improving GPU utilization with spatio-temporal scheduling.

V. Optimal Batching for DNNs

Batching is a trade-off between improving throughput at the cost of higher latency. Inferring a batch of requests requires more computation, thus increasing inference time. We consider the batch size as a function of network bandwidth. Therefore, preparing a bigger batch, i.e., receiving and transferring data from the network to GPU also contributes additional latency. Providing a higher GPU% for a bigger batch can mitigate the inference latency increase. However, giving more than a certain GPU% may be wasteful. We use the metric

$$Efficacy(\eta) = \frac{Throughput}{Latency \times GPU\%}, \tag{7}$$

of *Efficacy* (η) of using GPU resources as the basis to find a good operating point with respect to batch size and GPU%. We define η of a DNN at a certain batch size and GPU% as (7). Efficacy, η, lets us know how much throughput the GPU produces per unit time, per unit of GPU resource (GPU%).

A. Optimum Batch Size for Inference

We profiled the ResNet-50 model for inference at different batch sizes & GPU% configuration. Fig. 7 shows that both very high and very low batch size leads to low Efficacy due to high latency and reduced throughput respectively, thus, an optimal batch size is desired. We now develop an optimization formulation that can provide us with the right batch size and

TABLE V
NOTATION FOR OPTIMIZATION FORMULATION

Notation	Description
p_i	GPU% for Session i
b_i	batch size for Session i
$f_L(p_i, b_i)$	inference latency of batch b_i for model M_i at GPU% p_i
C_i	Request assembly time for Session i

GPU% for a model, given a deadline. First, we present the key notations used for the optimization in Table V.

The batch size is a product of the average incoming request rate and request assembly time. Thus, $b_i = \texttt{Request-Rate} \times C_i$. Throughput T_i is number of images inferred per unit time (8). Knowing throughput (8) we can write η (7), as (9). Equation (9) is of the same form as the first derivative of inverse of latency, (6), Section IV-B.

$$T_i = \frac{b_i}{f_L(p_i, b_i)} \tag{8}$$

$$\eta = \frac{b_i}{(f_L(p_i, b_i))^2 \times GPU\%}. \tag{9}$$

We seek to maximize Efficacy (η) to get the best balance in parameters based on the constraints (10), (11), and (12). The constraints express following requirements: (10): Batch size must be less than or equal to maximum batch size a

$$1 \leq b_i \leq Max\ Batch\ Size \tag{10}$$

$$f_L(p_i, b_i) + C_i \leq SLO_i \tag{11}$$

$$f_L(p_i, b_i) \leq \frac{SLO_i}{2}, \tag{12}$$

model can accept. Equation (11): The sum of times taken for aggregation of batch via network (C_i), and its inference execution, which has to satisfy the SLO. Equation (12): When working with a high request rate, we can regularly gather large batch sizes for inference. However, a request that cannot be accommodated into the current batch due to constraint (11), has to be inferred in the next batch. Then the deadline for next batch is the deadline of the oldest pending request. Therefore, we make sure that SLO is twice the time required to run a batch.

We computed the latency function $f_L(p_i, b_i)$, by fitting the latency observed while inferring DNN models with a batch size of 1,2,4,8,10,12,16 and GPU% from 10-100 at 10% intervals on our testbed. The optimization is solved using the non-linear programming solver 'fmincon' in MATLAB. Requests (images of resolution 224×224) arrive over a 10 Gbps link. 1 image is assembled every $\sim 481\ \mu s$. We use an SLO of 50 ms, allowing for an interactive system that can be used in safety critical environments such as autonomous driving [68].We present the feasibility region (where the SLO constraints are fulfilled) and optimal point provided by the optimization formulation in Fig. 8. The infeasible area is in a lighter shade. It is particularly revealing that Mobilenet has an optimal point close to 30%.

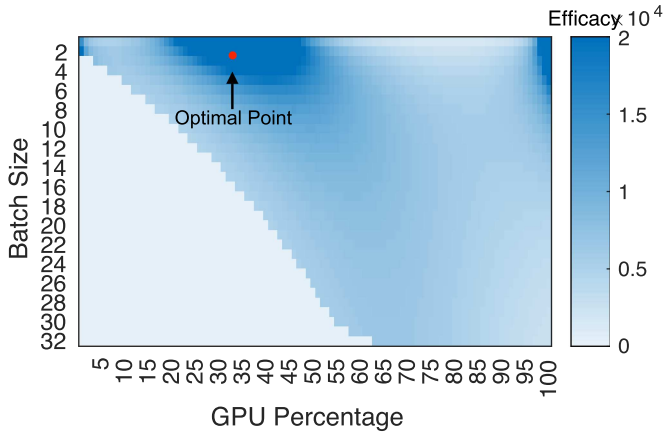


Fig. 8. Mobilenet feasibility region (darker shade).

B. Estimation of the Knee for Real Systems

We view these optimal values in relative terms, representative of the limit to parallelism that the model exhibits, because the optimization does not necessarily factor all the aspects that influence the execution of the model in the real system. We, however, pick a batch size and GPU% values from the high efficacy region in the optimization output in Fig. 8 and over-provision the GPU% by 5-10% while deploying the model in a real system.

VI. GPU SCHEDULING OF DNN MODELS

We now discuss the Spatio-temporal scheduling with D-STACK. We run the DNN models concurrently and meet their SLO while keeping the GPU from over-subscription. Over-subscription occurs when the aggregate GPU% of concurrent models exceed 100%.

A. Scheduling With Varying SLO

We schedule multiple models with different SLOs (deadlines), optimal batch sizes, and GPU% with D-STACK. Our scheduler considers two primary constraints. First, the DNN model must be scheduled at least once before an interval equal to its SLO, using an optimal batch size as predicted by the model in Section V. Second, the aggregate GPU demand at any point in the schedule should not exceed 100%. We choose a time period defined by the largest SLO to be a *Session*. Models with an SLO smaller than a session will run multiple times in a session. e.g., for a 100 ms session, a model with 25 ms SLO will run at least 4 times. Our spatio-temporal scheduling also accommodates dynamic arrivals of requests by utilizing a Fair, Opportunistic and Dynamic scheduling module which dynamically recomputes the schedule, thus increasing the effective utilization of the GPU.

We use 8 different DNN models and present their optimal batch size, GPU% and the latency of inference at that batch-size/GPU% in Table VI. We obtain the knee GPU% and Batch Size from the model in Section V. We chose our SLO based on safety-critical work such as autonomous driving [68], where it is determined that less than 130 ms processing is required to

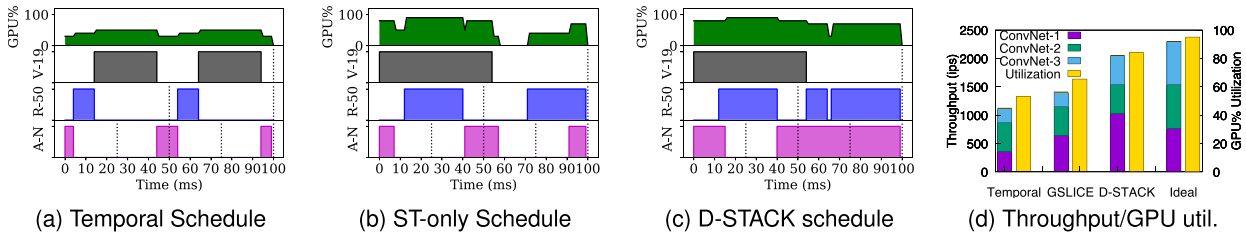


Fig. 9. (a, b, c) Scheduling Algorithms; (A-N=Alexnet, R-50=ResNet-50, V-19=VGG-19) (d) Comparison with ideal scheduler.

TABLE VI
CHARACTERISTICS OF DIFFERENT DNN MODELS

Model	Knee%	SLO (ms)	Batch (B_i) Sentence len.	Runtime (L_i) (ms)
Mobilenet	20	25	16	10
Alexnet	30	25	16	8
BERT	30	25	16 (10-words)	9
ResNet-50	40	50	16	28
VGG-19	50	100	16	55
ResNet-18	30	25	16	12
Inception	40	50	16	25
ResNeXt-50	50	100	16	40

safely stop a car running at 80 miles/hr (∼130 kmph). We choose a much more conservative 100 ms (effectively about 50 ms as rest is spent for preparing batch) for higher accuracy (VGG-19 and ResNext-50) and smaller SLOs (50 ms and 25 ms) for latency-optimized models (ResNet-50, Inception, Mobilenet, Alexnet and ResNet-18) aimed for application such as 30fps video stream. Unlike [7], we realistically consider that a model's execution cannot be preempted from GPU.

We first examine a temporal schedule with Alexnet, ResNet-50, and VGG-19. We provide time slices proportional to the model's SLOs. We utilize an adaptive batching algorithm mentioned in clipper [17] and Nexus [11] to obtain the batch size for each model's time slice. Fig. 9(a) is the visualization of such a schedule. The SLOs are visualized as the vertical dotted lines. We compute GPU utilization by using Knee% for each model as shown in Table VI. With temporal sharing, we achieve mean GPU utilization of 44%.

1) D-STACK: Spatio-Temporal Scheduling: Our D-STACK's scheduler aims to fit as many models as possible (potentially being different from each other) and run them concurrently in the GPU. We seek to be able to meet each model's (potentially different) SLO. We employ a simple version of the Earliest Deadline First Scheduling (EDF) algorithm to schedule all the models. EDF schedules the model with the tightest deadline to run first. However, we should note that as a model's inference is not preempted, this simple schedule cannot guarantee that the GPU will not be oversubscribed at any moment in the schedule. To aid in fitting in as many models as possible, we schedule consecutive executions of any model with the shortest SLOs to be as far apart as possible. This allows us to fit longer running models in the GPU in the interim without oversubscribing it. We demonstrate a schedule generated by spatio-temporal only algorithm in Fig. 9(b). We observe that the model with the smallest SLO, Alexnet (bottom), is scheduled to meet its SLO, but the time between the execution of the first instance and the second can be large because its execution time is short. This allows us to run ResNet-50 (second from the bottom) and VGG-19 (third) in between consecutive executions of Alexnet. Note that D-STACK's scheduler can also schedule a model with GPU% lower than its Knee, albeit with high inference latency when necessary. D-STACK also considers the additional latency of launching a new DNN model at lower GPU% into the schedule. This latency-GPU% trade-off has to be considered carefully before starting inference. Once a DNN process starts with its allocated GPU%, it cannot be changed for that instance's execution lifetime.

2) Fair, Opportunistic, Dynamic Scheduling: To efficiently utilize the GPU resource while ensuring that the system meets SLO guarantees, we further propose an opportunistic dynamic scheduling enhancement. The dynamic scheduling is triggered when a new request dynamically arrives for a model and when a model ends inference. The dynamic scheduler picks a model that is not active. This opportunistic addition is allowed as long as the GPU is not oversubscribed (so as to not interfere with the already scheduled models). To ensure fairness among available models, we use a scoreboard that tracks how many times each model has run in the last few (e.g., ten) sessions and prioritizes the models that have run the fewest. The algorithm then finds a time slice for the model to finish inferring and also determines a batch size that can complete within the time slice. If the highest priority model cannot be run, the algorithm picks the model with the next higher priority. We show the output of the D-STACK scheduling in Fig. 9(c). With this dynamic scheduling packing more models to be scheduled opportunistically, the average GPU utilization increases from 60% in the plain spatio-temporal schedule (Fig. 9(b)) to 74% with the D-STACK schedule (Fig. 9(c)).

Aggregate throughput is the addition of throughput of all the models. DSTACK does not prioritize any particular-sized neural network. As described in Section V(A, 2), we track the model execution in a scoreboard. So, any smaller model does not get the GPU all the time, but rather the available GPU. The GPU execution in time and space is fairly divided across all the neural networks running in the GPU. The smaller models can get a large throughput boost even with a small amount of time that the GPU SMs are allocated for them. Thus, the aggregate throughput is influenced considerably by smaller models. This is true for both DSTACK and for Timesharing (as in Triton).

B. An Ideal Spatio-Temporal Schedule versus D-STACK

We compare D-STACK against an ideal scheduler, which is a theoretical spatial and temporal schedule at the granularity of individual DNN kernels. For the ideal case, we assume GPU kernel preemption is allowed, a DNN's instantaneous GPU demand is known and the GPU's allocated resources are adjusted instantaneously. Any realistic system that does not preempt a currently running DNN model until its inference is completed, together with scheduling overheads to switch from one model to another inevitably under-utilizes the GPU. Thus, the ideal scheduler provides a theoretical 'optimal' performance achievable by D-STACK or other schedulers.

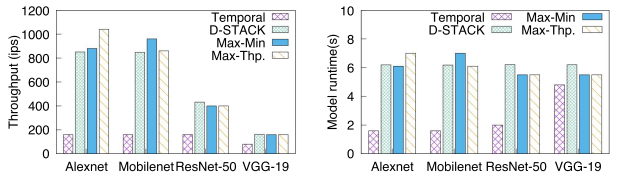
We consider a time-slotted system (e.g., 100 μs for experiments with a small scale DNN), where S_i represents i^{th} time slot in the schedule. We schedule the kernel k_m from DNN model m. We include as many model's kernels as will fit in the GPU at their Knee%, ordered by their earliest deadline. We compute the aggregate GPU% as $G_{ui} = \sum_{k \in S_i} GPU\%_k$ for each time slot S_i. We use an exhaustive search-based schedule to maximize the GPU utilization for every time slot (13). The overall GPU utilization G_u is maximized as

$$\max G_u, \quad \texttt{where} \;\; G_u = \sum_i G_{ui} = \sum_i \sum_{k \in S_i} GPU\%_k \tag{13}$$

$$\texttt{such that} \;\; G_{ui} \leq 100\% \;\; \texttt{and} \;\; k_i \in E \Rightarrow k_{i-1} \in E. \tag{14}$$

The first constraint for scheduling kernels of different models (14) is that the sum of the GPU% of all concurrent kernels in a time slot should not exceed 100%. Second, only eligible kernels (set (E)) can run concurrently in the time slot S_i being scheduled. DNN kernels are executed sequentially.

We experimented by scheduling 3 convolution neural networks (ConvNet) based on LeNet [69]. Each ConvNet has 3 convolution, 2 average-pool and 2 linear kernels. The dimensions of filters of the convolution layers are varied, varying the compute requirement for each ConvNet model. The inference image has a resolution of 224 $\times$ 224. The knee-runtime combination for ConvNet-1, ConvNet-2 and ConvNet-3 are 30%-10.3 ms; 40%-14.6 ms, and 60%-15.4 ms, respectively. We computed the knee of each kernel of each model, for use by the ideal scheduling during inference. We present the GPU utilization and throughput in Fig. 9(d). Temporal scheduling has a much lower GPU utilization, as it runs a single kernel on the GPU at a time. GSLICE improves the GPU utilization, but its static schedule leads to lower utilization when not enough models are running on the GPU. Ideal scheduling attains almost 95% GPU utilization, because it schedules kernels leveraging preemption. D-STACK schedules without preemption of a kernel, runs a DNN kernel to completion even if a kernel that could utilize the GPU better is waiting. Nonetheless, D-STACK still achieves ~86% GPU utilization. The throughput attained by the three CNN models follows the same trend. D-STACK's overall throughput is slightly higher than 90% of the throughput of ideal scheduling - a measure of how close it comes to the ideal scheduler.



(a) Throughput (b) Runtime for each model (sec)

Fig. 10. (a) Throughput of models running with different scheduling algo. and (b) Total runtime (s) per model.

C. Evaluation of D-STACK Scheduler

We evaluate D-STACK using four popular DNN models (Alexnet, Mobilenet, ResNet-50, and VGG-19) that are run with fixed SLOs, GPU%, and runtime as presented in Table VI. We ran the models concurrently for 10 seconds. We took the workload mix from the Imagenet [70] (vision DNNs), and IMDB dataset [71] (sentence classification with BERT). We introduce a random, uniformly distributed inter-arrival delay between requests destined for the same DNN model.

We compare the throughput, and GPU runtime of D-STACK with the baseline temporal sharing, and a schedule that maximizes the sum of the throughput across all the models (*max-throughput*). We also evaluate the fairness of the schedulers, measured by the GPU runtime each model gets. For this, we compare D-STACK against a Max-Min fair scheduler [72], which maximizes the placement of the minimum (smallest) demand (GPU%). The throughput result is shown in Fig. 10(a), and the GPU runtime each model gets is in Fig. 10(b).

D-STACK gets 2$\times$ the throughput of temporal sharing for the two compute-heavy models, ResNet-50 and VGG-19 (Fig. 10(a)). At the same time, the lighter-weight Alexnet and Mobilenet get 4$\times$ higher throughput. In temporal scheduling, running compute-heavy

DNNs with longer runtimes results in fewer opportunities for the other models, as there is no spatial sharing. Temporal scheduling runs models for only 1.6sec. out of 10 secs. time, negatively impacting their throughput. Fig. 10(b) shows that the D-STACK runs all the models longer than temporal sharing. This is because D-STACK can run multiple DNNs concurrently, providing higher throughput compared to temporal sharing (Fig. 10(a)). We compare D-STACK's throughput with the 'max-throughput' schedule. D-STACK gets more than 80% throughput of the max-throughput for the model with the lowest runtime (Alexnet) while providing better fairness as we see next.

The Max-Min fair schedule provides higher runtime for Mobilenet (Fig. 10(b)) than D-STACK since Mobilenet has the minimum demand (25% knee%). However, D-STACK achieves higher throughput than Max-Min for the medium runtime ResNet-50 (Fig. 10(a)). D-STACK's fairness measure picks the model that has run for the least time in the GPU over past sessions to schedule. Thus, D-STACK seeks to act like a proportional fair scheduler, as with the Completely Fair Scheduler (CFS) in Linux [73]. The fairness of D-STACK is shown in Fig. 10(b). Max-Min gives more time to a low-demand model like Mobilenet. With D-STACK, all the models get similar

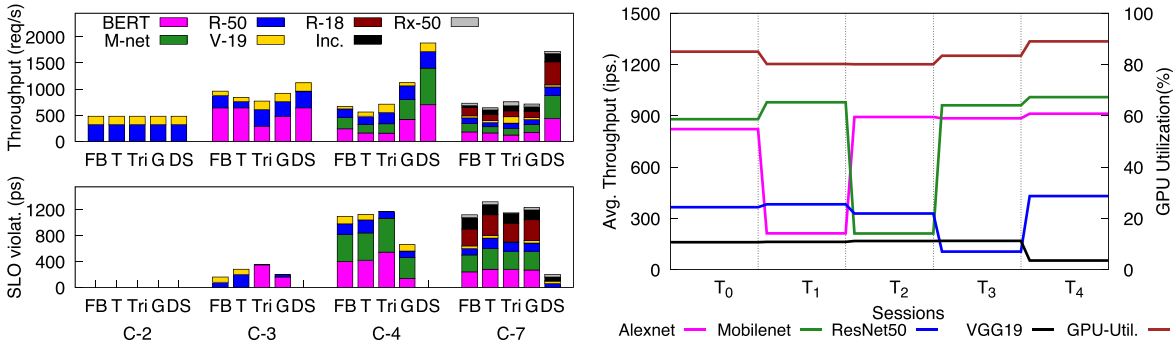


Fig. 11. (a) C-2 = ResNet-50 + VGG-19, C-3 = C-2 + BERT, C-4 = C-3 + Mobilenet, C-7 = C-4 + ResNet-18 + Inception + ResNeXt-50. (b) Throughput adjustment in D-STACK with varying request rate.

GPU time, thus boosting the total throughput of higher demand models like ResNet-50. Overall, the D-STACK scheduling beats temporal sharing's throughput by 4×, gets more than 80% of the max-throughput scheduler and fairly shares GPU execution time while meeting SLOs.

VII. Validating Our Overall Approach

We compare D-STACK with other multiplexing methods.

Multiplexing DNN models on the GPU: We evaluate three different cases of multiplexing by running 2, 3, 4 and 7 DNNs, respectively. By multiplexing 7 different DNNs, we demonstrate how D-STACK is still successful in scheduling a number of models with tight latency constraints, even if the sum-total of their demand (i.e., knee-capacity) is substantially higher than 100% GPU. We show D-STACK can improve throughput and utilize the GPU better while reducing the SLO violations compared to the other approaches, with all, including D-STACK having to compromise by missing the deadline on some inference requests. We compare our approach, including D-STACK, with four other methods of GPU multiplexing, namely, Fixed batching with Default CUDA MPS (FB), and temporal sharing (T), Triton Inference Server (Tri) (Also temporal sharing) and GSLICE (G). In Fixed batching with CUDA MPS (FB), the largest batch size of 16 is picked for inference every time and the multiplexing models share the GPU with MPS without an explicit GPU%. In temporal sharing (T), time slices are set in the proportion of the models' SLO length. With Triton server (Tri), we request the inference with multiple clients concurrently, allowing Triton server to dynamically batch and infer our requests. With GSLICE (G), we use all GSLICE's features, including adaptive batching and spatial sharing of the GPU at each DNN's knee. Finally, in D-STACK, we use the batch size and GPU% from our optimization formulation and utilize D-STACK scheduling to schedule the models.

We evaluate the throughput and the SLO violations per second for each model in Fig. 11(a). We measure SLO violations per second as the sum of all the inference requests that violate the SLO and all the unserved requests. Inference requests are generated at the rate of ∼ 1920 images/sec (max. request rate limited by the 10 Gbps link in testbed). Requests are divided into the multiplexed models in proportion to their SLOs. Thus, for the experiments C-2, C-3 and C4, Alexnet and Mobilenet get 700 inference requests/sec, ResNet-50 gets 320 requests/sec and VGG-19 gets 160 requests/sec.For the experiment with 7 DNN models running concurrently (i.e., C-7), Alexnet, Mobilenet and ResNet-18 receive 440 inference requests/sec, ResNet-50 and Inception receive 220 requests/sec while ResNeXt-50 and VGG-19 get 80 requests/sec.

We present *aggregate throughput* measure in Fig. 11(a). Aggregate throughput provides the sum of the throughputs achieved by all the models. D-STACK provides more than a 3× increase in aggregate throughput when multiplexing 7 different models. D-STACK achieves the highest throughput even when fewer models run concurrently.

We note that the smaller models can get a large throughput even with a small time durations they are scheduled on GPU. Thus, the aggregate throughput is influenced considerably by smaller models. This is true for both D-STACK and for other time and spatial sharing methods as well (as in Triton). With D-STACK's fairness mechanism (Section VI-A-2), we intentionally try to de-prioritize running small neural network just for increasing throughput. Other GPU sharing mechanisms have no such constraints. Therefore, we think aggregate throughput is a fair metric to measure the performance of GPU sharing across various schedulers.

For MPS, the lack of batching causes it to miss most of the SLOs for requests. Fixed batch, temporal sharing, GSLICE and Triton server provide good throughput while running just 2 models. However, as the number of models multiplexed increases, each new added model contends for GPU resources in Fixed Batch, decreasing the throughput. Meanwhile, in temporal sharing, each model gets less and less GPU time, impacting throughput.

Models hosted in Triton server too have to multiplex GPU temporally, thus, get lower throughput when more models are added. With GSLICE, multiplexing more models means some models get resources lower than knee GPU%, exponentially increasing the inference latency. D-STACK provides both the right amount of GPU resources and the appropriate batch size. Furthermore, there are no SLO violations in D-STACK when multiplexing 2-4 models. However, when overloading the GPU

by multiplexing 7 DNNs, we see a few SLO violations for the models with longer runtime (Inception, Resnet-50, ResNeXt-50 and VGG-19). D-STACK misses SLOs for 10% of all requests, compared to more than 68% for the alternatives. SLO misses for D-STACK are from the smaller fraction of requests sent to compute heavy models such as ResNet-50, ResNext-50 and VGG-19. Even with some of the medium-to-large sized models with longer runtimes, such as ResNet-50 and Inception, only 13% of requests see a SLO violation. This is due to the fact that running 7 models concurrently exceeds the capacity of GPU even with D-STACK. With D-STACK the average GPU utilization is 92% while multiplexing with 7 models. With all the models having a knee greater than 10%, this is close to fully utilizing the GPU.

Benefit of D-STACK Scheduler: Wherever possible, D-STACK tries to opportunistically schedule additional model instances during the session, possibly with a smaller batch size to utilize the available GPU. To show the effectiveness of the D-STACK, we present a scenario where the request rate of the multiplexed DNN models varies dynamically. To start with, in session T_0, we have 4 models, Alexnet, Mobilenet, ResNet-50 and, VGG-19, same as in 'C-4' in Fig. 11(a) running with their request rates high enough to support the optimal batch size, as determined in Table VI. The GPU utilization we achieve is $\sim 85\%$. We then change the request rate of one model (Alexnet in session T_1) by a random amount. We still allow for the optimal batch to form for each model. The throughput of the models dynamically adjust with the throughput of other models increasing due to use of the un-utilized resources left by Alexnet (see T_1). Since these three models have a high GPU% requirement, there is not enough GPU to accommodate an instance of another model. Thus, the GPU utilization drops very slightly. At T_2, Alexnet's request rate goes back up, while Mobilenet request rate lowers, once again by a random amount. Alexnet opportunistically uses the GPU to achieve a throughput higher than what it achieved in the baseline session T_0. Similarly, when ResNet-50 and, VGG-19's arrival rates drop at T_3 and T_4 , respectively, the other models increase their throughput. We also see that across these sessions, the GPU utilization is nearly unchanged, remaining high, indicating that the D-STACK effectively uses the GPU.

A. D-STACK in Multi-GPU Clusters

D-STACK can utilize multiple GPUs in the cluster. When the request rate of a model exceeds the throughput it is getting, D-STACK starts another instance of the model in another GPU in the cluster. These new GPUs will also be spatially shared with any new DNN models that are introduced.

We evaluated D-STACK in a multiple GPU cluster of 4 NVIDIA T4 GPUs, each having 40 SMs (fewer than a V100) and 16 GB of memory. We utilized 4 different vision models, Mobilenet, Alexnet, ResNet-50 and VGG-19 (knee GPU% is different for T4 GPU versus V100). We compare throughput of 3 different multiplexing and scheduling scenarios. First, we provide one T4 GPU for each DNN model exclusively. In the second scenario, we place all 4 models in each GPU, temporally sharing the GPU. Finally, we evaluate D-STACK with the 4 DNN models.

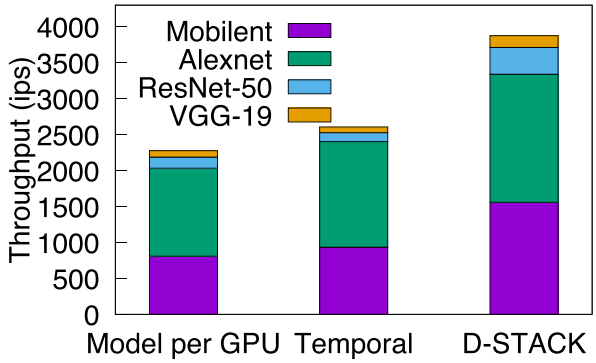


Fig. 12. GPU cluster throughput.

Fig. 12 shows temporal scheduling has almost the same throughput as each model having an exclusive GPU.

This is because of the under-utilization of the GPU by the DNN models. D-STACK has much higher throughput for every model, with 160% overall higher throughput than temporal sharing. The overall inference throughput increases substantially as the multi-GPU cluster is better utilized by D-STACK.

B. D-STACK 's Applicability With Different Devices

An application's performance on a V100 can be used to estimate how it would perform on another GPU in the same family. We do recognize that estimating the performance when we go across GPU generations may be a challenge. Streaming multiprocessors in a V100, A100, and H100 are drastically different with different cache sizes, and different hardware capabilities; thus, an application requiring 50% of a V100 GPU might require much less of a A100 GPU. This will be true for all applications whether they are using D-STACK or not. However, we can still take the initial guidance from the offline analysis done with another (say less powerful) GPU and divide the current more powerful GPU into several, when multiplexing different applications on the current more powerful GPU. The GPU resources for each application can be eventually adjusted, as new metrics as well as updated values of all metrics are collected when running the applications on the new more powerful GPU itself.

VIII. Conclusion

DNNs critically depend on GPUs and other accelerators, but often under-utilize the parallel computing capability of current high-performance accelerators. Due to uneven workloads of different DNN kernels, a DNN as a whole is unable to fully utilize all the parallelism of the GPU (i.e., all SMs). Furthermore, there are non-parallelizable tasks while executing a DNN on a GPU-based system limiting the effective use of a GPU's parallelism. We validated these conclusions from our model of a DNN through measurements of different types of DNNs (CNNs, and Transformers) on an V100 GPU. Since batching DNN requests improves inference throughput and GPU utilization, we develop an optimization framework to establish an optimal operating point (GPU%, Batch Size) for a DNN utilizing the GPU at the highest efficacy. We bring the optimal

batch size and GPU% together in D-STACK to develop a spatio-temporal, fair, opportunistic, and dynamic scheduler to create an inference framework that effectively virtualizes the GPU. D-STACK accounts for a DNN model's SLO, GPU resource allocation, and batch size, to provide a schedule that maximizes meeting SLOs, across multiple DNN models while seeking to utilize the GPU fully. D-STACK benefits both single GPUs and multi-GPU clusters. Our enhancements in D-STACK do not require modifications to the GPU architecture, the runtime, or the DNN models themselves. D-STACK's features can easily help improve existing DNN inference platforms (e.g., Triton server) as well. We show that D-STACK can attain higher than 90% throughput of an ideal scheduler, which we speculate can switch tasks instantaneously at a very fine time granularity, ignoring practical limitations.

Our controlled testbed experiments with 4 T4 GPU clusters show the throughput improvement of 160%-180% with D-STACK compared to providing an entire GPU to each individual DNN model. With an NVIDIA V100 GPU, D-STACK shows benefit in the range of ~1.6× improvement in GPU utilization and 3× to 4× increase in throughput with no impact in latency compared to the baseline temporal sharing.

REFERENCES

[1] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," in *Proc. 2018 IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2018, pp. 522–531.

[2] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 1–12.

[3] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-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

[4] Tensorflow serving, 2020. [Online]. Available: https://www.tensorflow.org/tfx/guide/serving

[5] NVIDIA triton inference server, 2021. [Online]. Available: https://docs.nvidia.com/deeplearning/triton-inference-server/master-user-guide/docs/

[6] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan, "GSLICE: Controlled spatial sharing of GPUs for a scalable inference platform," in *Proc. 11th ACM Symp. Cloud Comput.*, New York, NY, USA, 2020, pp. 492–506.

[7] W. Zhang et al., "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters," in *Proc. ACM Int. Conf. Supercomputing*, 2019, pp. 58–68.

[8] A. F. Inci et al., "The architectural implications of distributed reinforcement learning on CPU-GPU systems," 2020, *arXiv: 2012.04210*. [Online]. Available: https://arxiv.org/abs/2012.04210

[9] Y. Wang, G.-Y. Wei, and D. Brooks, "A systematic methodology for analysis of deep learning hardware and software platforms," in *Proc. Mach. Learn. Syst.*, 2020, pp. 30–43.

[10] H. Kong et al., "EDLAB: A benchmark for edge deep learning accelerators," *IEEE Des. Test*, vol. 39, no. 3, pp. 8–17, Jun. 2022.

[11] H. Shen et al., "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 322–337.

[12] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 103–112.

[13] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.

[14] NVIDIA Multi-Process Service, 2024. Accessed: Aug. 11, 2022. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[15] NVIDIA, "Driving digital transformation with GPU virtualization and enterprise cloud," 2017. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nutanix/pdf/nutanix-solution-overview.pdf

[16] NVIDIA, "Unlock next level performance with virtual GPUs," 2021. [Online]. Available: https://www.nvidia.com/en-us/data-center/virtual-solutions/

[17] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 613–627.

[18] J. Gu et al., "Tiresias: A GPU cluster manager for distributed deep learning," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, Boston, MA: USENIX Association, 2019, pp. 485–500. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/gu

[19] A. Gujarati et al., "Serving DNNs like clockwork: Performance predictability from the bottom up," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 443–462.

[20] P. Gao, L. Yu, Y. Wu, and J. Li, "Low latency RNN inference with cellular batching," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–15.

[21] AWS, "Host multiple models with multi-model endpoints," 2021. [Online]. Available: https://docs.aws.amazon.com/sagemaker/latest/dg/multi-model-endpoints.html

[22] 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 *Proc. 29th Int. Symp. High- Perform. Parallel Distrib. Comput.*, New York, NY, USA, 2020, pp. 173–184.

[23] W. Xiao et al., "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 595–610.

[24] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for GPU based cloud servers using machine learning," in *Proc. 2016 IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 353–362.

[25] K. Zhang et al., "G-net: Effective GPU sharing in NFV systems," in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 187–200.

[26] A. Zhu, D. Zeng, L. Gu, P. Li, and Q. Chen, "Gost: Enabling efficient spatio-temporal GPU sharing for network function virtualization," in *Proc. IEEE/ACM 29th Int. Symp. Qual. Service*, 2021, pp. 1–10.

[27] 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.

[28] H. Zhou, S. Bateni, and C. Liu, "S3DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads," in *Proc. 2018 IEEE Real-Time Embedded Technol. Appl. Symp.*, 2018, pp. 190–201.

[29] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 17–32.

[30] M. E. Belviranli, F. Khorasani, L. N. Bhuyan, and R. Gupta, "CuMAS: Data transfer aware multi-application scheduling for shared GPUs," in *Proc. 2016 Int. Conf. Supercomputing*, New York, NY, USA, 2016, Art. no. 31, doi: 10.1145/2925426.2926271.

[31] C. Yu et al., "SMGuard: A flexible and fine-grained resource management framework for GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 12, pp. 2849–2862, Dec. 2018.

[32] Q. Sun, L. Yi, H. Yang, M. Li, Z. Luan, and D. Qian, "QoS-aware dynamic resource allocation with improved utilization and energy efficiency on GPU," *Parallel Comput.*, vol. 113, 2022, Art. no. 102958.

[33] X. Zhao, Z. Wang, and L. Eeckhout, "Classification-driven search for effective SM partitioning in multitasking GPUs," in *Proc. 2018 Int. Conf. Supercomputing*, 2018, pp. 65–75.

[34] M. Jeon et al., "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *Proc. 2019 USENIX Annu. Tech. Conf.*, 2019, pp. 947–960.

[35] G.-F. Yeung, D. Borowiec, A. Friday, R. Harper, and P. Garraghan, "Towards GPU utilization prediction for cloud deep learning," in *Proc. 12th USENIX Workshop Hot Topics Cloud Comput.*, 2020, Art. no. 6.

[36] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, and P. Garraghan, "Horus: An interference-aware resource manager for deep learning systems," in *Proc. 20th Int. Conf. Algorithms Architectures Parallel Process.*, New York City, NY, USA, Springer, 2020, pp. 492–508.

[37] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimizing DNN computation with relaxed graph substitutions," in *Proc. Mach. Learn. Syst.*, 2019, pp. 27–39.

[38] X. Du, M. El-Khamy, J. Lee, and L. Davis, "Fused DNN: A deep neural network fusion approach to fast and robust pedestrian detection," in *Proc. 2017 IEEE Winter Conf. Appl. Comput. Vis.*, 2017, pp. 953–961.

[39] M. Song, Y. Hu, H. Chen, and T. Li, "Towards pervasive and user satisfactory CNN across GPU microarchitectures," in *Proc. 2017 IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 1–12.
[40] T. Chen et al., "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 578–594.
[41] P. Jain, X. Mo, A. Jain, A. Tumanov, J. E. Gonzalez, and I. Stoica, "The OOO VLIW JIT compiler for GPU inference," 2019, *arXiv: 1901.10008*. [Online]. Available: http://arxiv.org/abs/1901.10008
[42] P. Jain et al., "Dynamic space-time scheduling for gpu inference," 2018, *arXiv: 1901.00041*.
[43] S. Dublish, V. Nagarajan, and N. Topham, "Poise: Balancing thread-level parallelism and memory system performance in GPUs using machine learning," in *Proc. 2019 IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 492–505.
[44] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proc. 22nd Int. Conf. Parallel Architectures Compilation Techn.*, 2013, pp. 157–166.
[45] Q. Liang, W. A. Hanafy, A. Ali-Eldin, and P. Shenoy, "Model-driven cluster resource management for ai workloads in edge clouds," 2022, *arXiv:2201.07312*.
[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, Red Hook, NY, USA: Curran Associates Inc., 2012, pp. 1097–1105.
[47] A. G. Howard et al., "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv: 1704.04861*.
[48] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[49] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[50] C. Szegedy et al., "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.
[51] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 1492–1500.
[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[53] Torchvision model zoo, 2021, Accessed: Jun. 13, 2021. [Online]. Available: https://pytorch.org/docs/master/torchvision/models.html
[54] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Annu. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019, pp. 4171–4186.
[55] W. Zhang et al., "OpenNetVM: A platform for high performance network service chains," in *Proc. 2016 ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtualization*, 2016, pp. 26–31.
[56] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A scriptable high-speed packet generator," in *Proc. Internet Meas. Conf.*, Tokyo, Japan, 2015, pp. 275–287.
[57] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "DeepCPU: Serving RNN-based deep learning models 10x faster," in *Proc. 2018 USENIX Annu. Tech. Conf.*, Boston, MA: USENIX Association, 2018, pp. 951–965. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/zhang-minjia
[58] NVIDIA, "Deep learning performance documentation," 2021, Accessed: Apr. 07, 2021. [Online]. Available: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html
[59] Y. Wu et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.
[60] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 72–86, Jan. 2017.
[61] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta GPU architecture via microbenchmarking," 2018, *arXiv: 1804.06826*.
[62] W. Zhang et al., "Towards QoS-aware and resource-efficient GPU microservices based on spatial multitasking GPUs in datacenters," 2020, *arXiv: 2005.02088*.
[63] P. Micikevicius, "GPU performance analysis and optimization," in *Proc. GPU Technol. Conf.*, 2012, pp. 71–75.
[64] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. Spring Joint Comput. Conf.*, 1967, pp. 483–485.
[65] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988, doi: 10.1145/42411.42415.
[66] NVIDIA visual profiler user guide, 2021, Accessed: Jan. 12, 2021. [Online]. Available: https://docs.nvidia.com/pdf/CUDA_Profiler_Users_Guide.pdf
[67] NVIDIA tesla v100 GPU architecture, 2018, Accessed: Jan. 12, 2018. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[68] H. Qiu, F. Ahmad, F. Bai, M. Gruteser, and R. Govindan, "AVR: Augmented vehicular reality," in *Proc. 16th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2018, pp. 81–95.
[69] Y. LeCun et al., "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, 1989.
[70] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[71] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics: Hum. Lang. Technol.*, Portland, Oregon, USA: Association for Computational Linguistics, 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11--1015
[72] D. P. Bertsekas, R. G. Gallager, and P. Humblet, *Data Networks*, vol. 2. Hoboken, NJ, USA: Prentice-Hall International, 1992.
[73] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 2009, no. 184, pp. 184–187, Aug. 2009.

Aditya Dhakal (Member, IEEE) received the BE degree from the Kyushu Institute of Technology, Japan with the Japanese Ministry of Education scholarship, the MS degree from the University of Connecticut, and the PhD degree from the University of California, Riverside. He is a research scientist with Hewlett Packard Labs, Milpitas, California. His area of research included hardware multiplexing and neural network inference. His current research interests include GPUs, FPGAs, SmartNICs, communication fabrics and scalability in high-performance computing and Machine Learning.

Sameer G. Kulkarni received the PhD degree from the University of Göttingen, Germany. He worked as a postdoctoral researcher with the University of California at Riverside, Riverside. He is currently an assistant professor with the Department of Computer Science and Engineering, and Electrical Engineering, Indian Institute of Technology Gandhinagar. His current research interests include parallel and distributed computing, software defined networks, network function virtualization, network security and cloud computing. His PhD thesis received the IEEE Technical Committee on Scalable Computing Outstanding Dissertation Award, in 2019.

K. K. Ramakrishnan (Life Fellow, IEEE) received the MTech degree from the Indian Institute of Science, in 1978, and the MS and PhD degree in computer science from the University of Maryland, College Park, in 1981 and 1983, respectively. He is a distinguished professor of computer science and engineering with the University of California, Riverside. He joined AT&T Bell Labs, in 1994 and was with AT&T Labs-Research from its inception, in 1996, until 2013, as a distinguished member of Technical Staff. Before 1994, he was a technical director and consulting engineer in networking with Digital Equipment Corporation. Between 2000 and 2002, he was with TeraOptic Networks, Inc., as founder and vice president. He is an ACM fellow, and an AT&T fellow, recognized for his fundamental contributions to communication networks, including his work on congestion control, traffic management, and VPN services. His work on the "DECbit" congestion avoidance protocol received the ACM Sigcomm Test of Time Paper Award, in 2006, and he received the AT&T Technology Medal, in 2012 for his work on Mobile Video Delivery. He received the 2024 ACM SIGCOMM Award recognizing his lifetime contribution to the field of communication networks. He has published more than 300 papers and has 186 patents issued in his name.