GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform

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

Sameer G Kulkarni
IIT, Gandhinagar
sameergk@iitgn.ac.in

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

ABSTRACT

The increasing demand for cloud-based inference services requires the use of Graphics Processing Unit (GPU). It is highly desirable to utilize GPU efficiently by multiplexing different inference tasks on the GPU. Batched processing, CUDA streams and Multi-process-service (MPS) help. However, we find that these are not adequate for achieving scalability by efficiently utilizing GPUs, and do not guarantee predictable performance.

GSLICE addresses these challenges by incorporating a dynamic GPU resource allocation and management framework to maximize performance and resource utilization. We virtualize the GPU by apportioning the GPU resources across different Inference Functions (IFs), thus providing isolation and guaranteeing performance. We develop self-learning and adaptive GPU resource allocation and batching schemes that account for network traffic characteristics, while also keeping inference latencies below service level objectives. GSLICE adapts quickly to the streaming data's workload intensity and the variability of GPU processing costs. GSLICE provides scalability of the GPU for IF processing through efficient and controlled spatial multiplexing, coupled with a GPU resource re-allocation scheme with near-zero ($< 100\mu s$) downtime. Compared to default MPS and TensorRT, GSLICE improves GPU utilization efficiency by 60-800% and achieves 2-13× improvement in aggregate throughput.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Machine learning**.

ACM Reference Format:

Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421284

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10.
https://doi.org/10.1145/3419111.3421284

1 INTRODUCTION

Deep Neural Networks (DNNs) used for machine learning (ML) and inference have been continually improving in accuracy, while becoming more complex and memory and computation resource-hungry. Executing DNNs on Graphics Processing Units (GPUs) has become key to meet quality of experience and service level objectives (SLO) [12, 42].

DNN models used for inference services inherently have a rich amount of parallelism. However, even the most complex of the DNN models (VGG-19 [53] with 20 giga floating point operations (GFLOPS), GNMT [58] with 210 million parameters [14]) are unable to fully exploit the potential of massive compute and memory-bandwidth capabilities offered by the modern GPUs. Especially the discrete-GPUs (d-GPUs) *e.g.,* NVIDIA Tesla V100 offers ~125 Tera floating point operations per second (TFLOPS), and 900GBps memory bandwidth) [45]. As such, time-sharing the GPU to execute different DNN models often leads to wastage of GPU resources [13, 46, 62]. Hence, it is desirable to spatially multiplex the GPU across multiple DNNs and accordingly allocate just the right amount of GPU resources to boost overall throughput without sacrificing the individual model's inference performance. This requires us to understand the limits of parallelism that a particular DNN model can exploit.

The state-of-the-art CUDA MPS (multi-process service) helps improve GPU utilization by enabling spatial sharing of the GPU across multiple processes. But the default mode of MPS oversubscribes the GPU[1] resulting in uncontrolled spatial sharing. Thus, it cannot guarantee performance isolation and often results in unpredictable application throughput and latency [34]. Multiple streams and large batches further exacerbate this unpredictability with MPS [60]. As an alternative to its default operation, MPS also allows us to set a fixed limit for a process's GPU %, with the goal of isolating the performance of different processes. However, this static provisioning means the MPS provisioning would not be adaptive to workload variations. When the workload changes, that statically fixed GPU% can result in over-provisioning (thus wasting the GPU) or under-provisioning (hurting application

[1] All applications are allowed to get the full 100% share of the GPU.

performance). Our work, GSLICE, addresses these challenges by providing the appropriate CPU-GPU coordination and dynamic GPU resource-allocation mechanisms.

GSLICE advances the state-of-the-art spatial multiplexing (MPS) and supports many ML frameworks that can be run AS-IS. Additionally, GSLICE innovates by designing novel self-learning adaptive batching, parameter sharing in GPU for different inference instances and optimized zero-copy data transfer of streaming data to the GPU. We believe these are applicable even for non-CUDA or non-NVIDIA GPU environments such as OpenCL [54], ROCm [9], and METAL [6].

GSLICE is a Data Plane Development Kit (DPDK)[1] based **Generic/Integrated Inference Platform**. GSLICE can host a number of ML frameworks such as CNTK [51], Darknet[49], PyTorch [48], MXNet[15], TensorFlow[11], TensorRT[44], *etc.* with minimal changes[2], and support concurrent execution of different real-time inference and machine learning applications with streaming data (§3.1).

GSLICE builds on top of the CUDA MPS to provide performance isolation through **dynamic resource provisioning and spatial sharing** of GPU. DNN models have inherent limitations in exploiting parallelism. We extensively profiled different models (*e.g.,* Alexnet[38], GNMTv2[58], Jasper[39], Mobilenet[31], ResNet[29], VGG[53]), and observed that after a certain GPU% (we call it the **'kneepoint'**), performance improvement reaches a point of diminishing returns[3]. GSLICE overcomes this by allowing multiple inference applications (we call them inference functions (IFs)) to spatially share the GPU. We implement a low-overhead monitoring scheme to track an IFs' bottleneck and system-wide GPU resource usage. Then, GSLICE **dynamically readjusts the allocated GPU resources** to meet the desired SLO and demand (arrival rate) for each IF in a **self-tuning, adaptive manner**. In CUDA [22], adjusting GPU resources requires the IFs to be restarted[4]. However, restarting IFs incurs very high downtime (2-15s) due to the framework startup costs [48]. We innovate by introducing an **active-standby IF pair with overlapped execution**. We create a shadow IF and transparently re-provision the IF's GPU resources by providing new allocations to its shadow. We then prevent the GPU from being idle by a careful switchover to the shadow IF, thus reducing downtime to less than 100μs. Similarly, when an application requires a new instance with additional GPU resources, we use an efficient overlap technique to mask the startup latency of instantiating a new IF (§3.2.1, §3.2.2, §3.2.3).

[2]We support all python based ML frameworks using Py-c plugin.

[3]the decrease in the marginal (incremental) output of a production process as the amount of a single factor of production is incrementally increased, while the amounts of all other factors of production stay constant [56].

[4]GPU% needs to be set as an environment variable before CUDA initialization, and it cannot be changed till the end of the process [46]

GSLICE includes several optimizations to improve GPU utilization and loading time of IF models to the GPU (§3.2.4).

We recognize 'Batching' improves throughput and GPU utilization for inference applications, but impact latency. Dynamically adapting the batch size is key in achieving improved throughput while keeping latency below the SLO. We devise a **self-learning, adaptive batching scheme** that factors the network, CPU and GPU processing costs, and SLO constraints, to batch just the sufficient number of requests (*e.g.,* images) for inference. (§3.3.1)

GSLICE also supports **zero-copy data transfer** to the GPU. We share DPDK's page-locked memory with GPU and leverage the GPU's DMA to directly scatter-gather data from the network packets. (§3.3.2). Many popular DNN models have large memory requirements for storing the model weights and parameters. They account for ∼10-30% of total GPU memory footprint (Table 1). Memory intensive parameters (*i.e.,* Weights) of IF models do not change across inference executions. Thus they can be shared and reused across multiple instances. Hence, we implement a method for **parameter sharing** across multiple IFs once transferred to GPU memory. This allows us to drastically reduce the IF's memory footprint on the GPU, and allows for scaling and multiplexing a larger number of IFs on memory constrained GPU devices. (§3.3.3). Our work complements the optimizations proposed in works [16, 28, 32] to lower the GPU memory footprint for IF models.

2 BACKGROUND & MOTIVATION

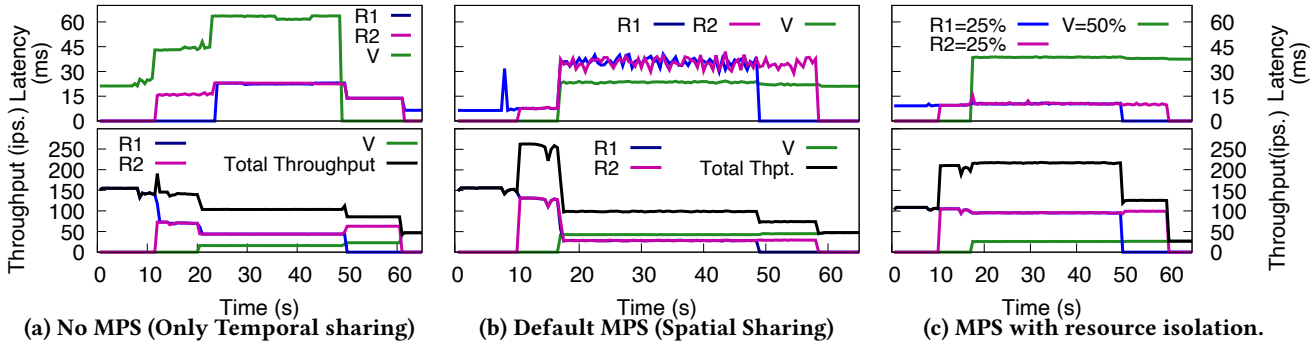
We study several widely used ML frameworks and models, and share experimental observations with spatial sharing and the limited use of GPU resources by different IF models.

2.1 Maximizing GPU Utilization

We briefly describe the key relevant approaches proposed to improve performance and GPU utilization.

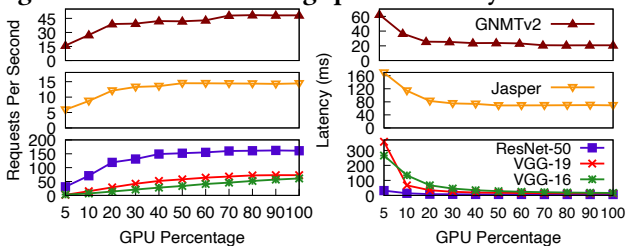
2.1.1 Batched execution. Batch processing improves GPU utilization because: i) the GPU can spawn additional compute threads to work on multiple requests in parallel; ii) helps amortize the CPU-GPU interaction and GPU transaction overheads [27]. However, it can result in high latency, which can be undesirable. We propose adaptive batching schemes, to reap the benefits of batch processing, while controlling the latency (see section §3.3.1).

2.1.2 Spatial GPU Multiplexing. NVIDIA GPUs can be spatially shared using CUDA streams and MPS to provide concurrent execution of GPU kernels within and across multiple processes respectively. Streams allow launching and execution of kernels concurrently within a single process [47]. **CUDA MPS** enables spatial sharing of the GPU and allows



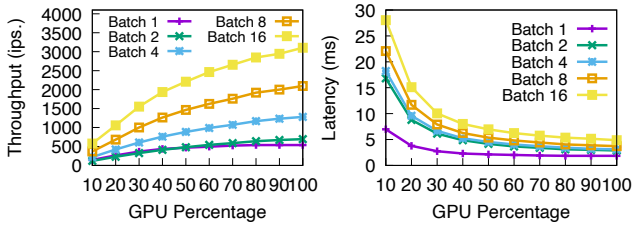
(a) No MPS (Only Temporal sharing) (b) Default MPS (Spatial Sharing) (c) MPS with resource isolation.

Figure 1: Inference Throughput & Latency - 2 ResNet-50 (R1& R2) & VGG-19 (V): TensorRT multiplexing modes.



(a) Throughput. (b) Latency.

Figure 2: Different DNN models at varying GPU %.



(a) Throughput. (b) Latency.

Figure 3: Alexnet with different batch size and GPU%.

concurrent execution of kernels from different processes. By default, an MPS client can use all the available threads (*i.e.,* 100% of GPU resources). However, this results in a large GPU memory footprint for the process, which is wasteful when the GPU is shared across multiple clients. Further, GPU resource contention among kernels from different processes can result in unpredictable latency for applications.

We experimented with three different IFs (IF-1 and IF-2 are ResNet-50 and IF-3 is VGG-19). We start with IF-1 and after 10 seconds(s) we launch IF-2 and bring up IF-3 after 20s. We repeat the same experiment for different scenarios a) without MPS. b) Default MPS. c) explicit allocation of GPU%. Fig. 1 shows the impact on throughput and latency for these scenarios. With the baseline, temporal sharing of GPU 'No MPS' case, both the latency and throughput degrade with every new addition of an IF. The latency of heavier model (VGG-19) rises from ~20 ms with 1 model to ~60 ms when all 3 models are running concurrently. Moreover, the first instance of ResNet-50's latency also rises from ~15 ms to higher than 20 ms. With temporal sharing, adding a model on the GPU causes the latency to rise for all concurrently running models. With 'Default MPS' mode, we can observe that uncontrolled spatial sharing results in the heavier VGG-19 getting the major share of GPU and getting better throughput than with 'No MPS'. But, the two lighter ResNet-50 instances see degraded performance (both in throughput and latency), with latency rising from ~10 ms to ~30 ms due to GPU resource contention. Throughput drops from ~100 images/sec to ~30 images/sec. Hence, in both cases ('No MPS' and 'Default MPS'), the interference due to addition/removal of IFs results in unpredictable throughput and latency.

MPS with resource provisioning: On the NVIDIA Volta architecture, CUDA MPS allows limiting the available GPU threads (in units of Streaming Multiprocessors (SMs)) for each process. This helps us improve the GPU utilization by apportioning just the right amount of GPU resources, while at the same time providing the benefits of performance isolation and reduced GPU memory footprint [46]. While this feature is useful, it still requires more careful consideration and resource management to properly reap the benefits, while fully utilizing the GPU. We repeat the previous experiments with fixed resource provisioning. We explicitly restrict each IFs to use only a fixed portion of available GPU resources. We limit the heavy VGG-19 IF to 50% GPU, and allocate the two ResNet-50 IFs with 25% GPU each. This results in almost interference-free execution of the IFs (when all 3 IFs are multiplexed). Further, it outperforms the other two modes, providing almost 50% higher aggregate throughput. Note that when only a single function (IF-1), or two functions (IF-1 and IF-2) are executing (between 0 and 15sec), the throughput is lower than the 'Default MPS' and 'no MPS' modes. This is primarily due to our limiting the GPU resources for these two instances. This demonstrates that, although fixed resource allocation and isolation is helpful, it has to be further improved to achieve good system performance. Thus, along with MPS, a judicious and *dynamic* resource allocation for the contending IFs is vital to maximize GPU utilization. We should note that using GSLICE's dynamic provisioning of

different GPU% does not affect results of DNN operation as the exact same DNN kernels are used to infer requests.

We also conducted experiments with different combinations of models. Due to lack of space, we omit these results. Interestingly, we observed that the 'No MPS' case always provided an equal share (round-robin time-sliced scheduling). However, the 'Default MPS' mode preferred the heavier models (due to uncontrolled spatial sharing), resulting in performance degradation of the light-weight models. We also ran experiments to see if running multiple models concurrently with resource isolation will cause an impact on throughput or latency due to contention for other GPU resources such as memory. We observed the same average inference latency for a model running at a fixed GPU%, whether a single instance of a model was running in the GPU, or the model was concurrently running with other models. We find this to be true, as long as the contention from other models doesn't bring the GPU% allocated to this model below its "knee".

2.1.3 Identifying the operating point for IF models. We profiled several different types of IF models, *viz.* Image (VGG-16, VGG-19, ResNet-50), Audio (Jasper) and text (GNMTv2) to understand their limits on exploiting parallelism and to derive the kneepoint for an IF in terms of the amount of GPU resources that achieve the best balance between achieved performance and the GPU resources expended. Figures 2a and 2b show the throughput and latency for processing a task (batch size 1) by a variety of IF models provided with different GPU percentages. The first insight we derive is that different models exhibit different degrees of parallelism. Individual DNN layers can have many parallel kernels that can concurrently execute in a GPU, exhibiting quite a bit of parallelism. However, most of the DNN layers are still unable to fully utilize the significant parallelism offered by high-powered GPUs. Note, both the throughput and latency reach a point of diminishing return for Jasper at 50% GPU, for GNMT, VGG-19 after 70%, for Resnet-50 at ~60%. We believe this limit arises due to inherent communication overheads and non-parallelizable work performed across different layers of the DNN models. Therefore, it is wasteful to provide full GPU to the models (as is done when the GPU is only temporally shared). It is preferable to spatially share the GPU to effectively utilize the GPU resources. Second, we can observe that provisioning the GPU% has significant impact on both throughput and latency; and the relationship is non-linear. It is crucial to find the right GPU% that would allows us to meet the demand (arrival rate of requests) while being within the budgeted latency (SLO) for each IF. At the same time, it is also necessary to maximize the aggregate throughput for all concurrently executing IFs and balance the overall GPU resource demand across the contending IFs. This requires a mechanism to apportion the GPU% to each of the concurrently executing inference applications, as a function of their request arrival rate, inference computation cost and SLO.

Batch Size & Resource Provisioning Dependency Further, to understand the implications of batched execution of tasks on CUDA MPS resource provisioning, we extensively evaluated different ML models (Alexnet, ResNet, VGG, *etc.*) on both CNTK and TensorRT frameworks for different fixed batch sizes. Figure 3 shows the impact of batch size when varying the GPU% for a particular Alexnet model. We observe that larger batch sizes increase both throughput and latency. Also, for a given batch size, increasing the GPU% increases the throughput (Fig. 3a) until it reaches a point of diminishing returns. *e.g.,* for a batch size of 1, we clearly observe that a GPU% beyond 30% ('kneepoint') results in only a marginal improvement, and the corresponding latency (Fig. 3b) reduction is also minimal. Moreover, we can observe that with larger batch sizes, the knee shifts towards larger GPU%. The results were similar for other models too. The key challenge here is to correctly identify and provision the GPU% for all the IFs.

2.1.4 GPU-based IF System Design Challenges. We summarize the major challenges in building a scalable spatially shared inference platform. First, determining the correct amount of GPU resources to provision different IF applications based on the desired SLO (latency) and throughput is key to effective multiplexing. However, provisioning the right amount of GPU resources is complex for the following reasons: i) IF models with different complexity demand different amounts of GPU resources. ii) Operating with different batch sizes and concurrent streams further change the GPU resource demand. iii) Underprovisioning the GPU resources for an IF can hinder meeting that application's SLO, while overprovisioning degrades GPU utilization. Finally, we observe that being IF application-aware in sharing parameters across instances and efficiently moving streaming data to the GPU subsystem is also important to maximize performance.

Quantifying GPU Utilization We noted that the 'GPU occupancy' provided by CUDA is misleading and shows 100% even when only one thread is active on the GPU; and as such there isn't a good indicator for measuring the effectiveness of GPU utilization. We introduce a metric, the GPU utilization efficacy (GUE (η)) for different modes of operation as: $\eta = \alpha * \frac{Throughput(IF)}{Latency(IF)}$, choosing $\alpha = 1$.

3 GSLICE: ARCHITECTURE & DESIGN

Fig. 4 shows the architecture of GSLICE. We describe the key components of GSLICE and their roles below.

3.1 Integrated Platform for IF

Today's general IF platforms need to support a number of heterogeneous ML frameworks (ML libraries) like CNTK [51],

Pytorch [19], TensorRT [44] *etc.* While they may run on the CPU alone, we focus on designing a platform that uses the GPU to provide low latency. GSLICE provides a 'C/C++' native platform to build and deploy IF functions (IFs). In order to support heterogeneous ML frameworks, we use a callback model to present a minimal set of generic interfaces that abstract out framework-specific functionality in a library (called *libml*), described below. This allows *libml* to provide basic IF services without being coupled with any particular ML framework.

The IF Manager component is the primary DPDK [1] process responsible for routing and delivering network packets to *libml*. While the IF Manager is agnostic of the ML frameworks running, it has an IF model loader component that holds the IF's profile *i.e.*, knee information for DNN model, CPU-GPU buffer requirement *etc.* .

libml implements a GPU resource allocation manager that determines and allocates the right amount of GPU resources to each IF. It also interacts with an Orchestrator service to restart IFs when the GPU (%) resources are readjusted.

The ***libml*** provides the following features to IFs, i) zero-copy network packet data transfer and aggregation (*e.g.,* of images), ii) the buffer pools (CPU and GPU memory) for batched execution. iii) GPU data transfer (DTTx in Fig. 4), iv) streaming engine to manage multiple streams (spatial GPU sharing) for performing inference, v) adaptive batching to intelligently perform batched executions within the latency limits specified by the SLO. Further, it provides the callback APIs to the IFs to setup framework specific functions. Once IFs setup the framework specific functions, *libml* can perform IF services transparently.We have kept *libml* lightweight with just 6 interfaces *(init, deinit, load_model, link_model, configure_batch, infer_batch)*. Fewer interfaces allows us to quickly adjust *libml* to work with newer version of ML frameworks, thus, keeping *libml* usable even with ML framework churn.

The Orchestrator is responsible for instantiating new IFs and decommissioning an existing IF. We leverage ZeroMQ [30] to build asynchronous message-based communication between the IF Manager and the Orchestrator.

3.2 Key Features and Design choices

3.2.1 Lightweight monitoring . We determine the bottleneck entity (CPU or GPU) in an IF operation based on timestamps tracked for task (*e.g.,* image) aggregation (first packet to last packet) and inference processing time on GPU and CPU respectively.[5] *libml* tracks the timestamps and notifies the IF Manager which takes timely actions to overcome the bottleneck. The CPU becomes the bottleneck when performing data copy operations (*e.g.,* preparing the aggregated image

[5]CPU time: T_{end} :(release all packets) - T_{start} :(when the first request arrives); GPU time: time to transfer & infer a batch of requests.

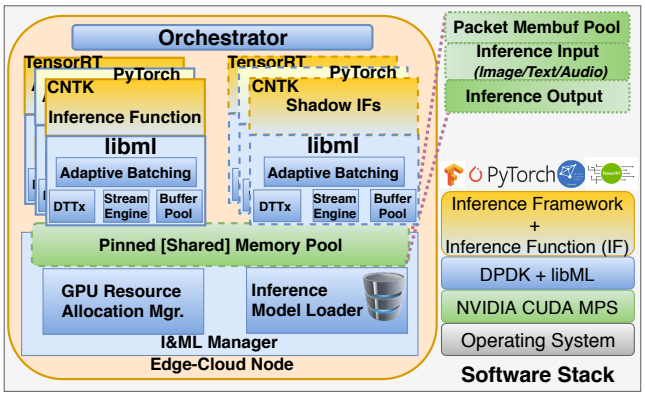


Figure 4: Architecture of GSLICE Inference platform.

by copying data from multiple packet payloads into a pinned contiguous buffer). We use a zero-copy GPU scatter-gather approach (*e.g.,* offloading image data transfer to GPU), to overcome the CPU bottleneck.

The IF Manager tracks notifications from *libml* and readjusts the GPU resources for bottlenecked IFs. However, such a readjustment is a very expensive operation, since we have to restart the IFs in order for the newly provisioned GPU resources to take effect. This time varies, but is generally of the order of 2-15s. Therefore, the IF Manager re-provisions the GPU% only when essential, readjusting over a coarse time-interval. We track the timestamp at each IF for the last readjustment and hold-down any further changes till a specified minimum time (*e.g.,* 10 sec.) has elapsed.

3.2.2 Self-tuning GPU Resources of an IF (GPU%) . In order to ensure timely inference operation and dynamic adaptation to demand variation (arrival rate), we develop a self-tuning mechanism to enable IFs to readjust their GPU resource share. We consider the following two aspects to distinguish the correct resource provisioning for an IF. i) 'Residual Latency capacity': the difference between SLO and observed latency for inference. ii) 'Residual Throughput capacity': the difference between the achieved GPU throughput and the actual demand (arrival rate). When either of these residual capacities is negative, we try to proportionally increase the GPU%, and when both the capacities are positive and larger than a specified threshold (greater than 5%, to accommodate variability in latency), we try to reduce the provisioned GPU% in proportion to the available capacity. The resource allocation manager handles re-provisioning of GPU resources for all the IFs.

The pseudocode for the self-tuning GPU resource allocation is described in Algorithm 1. We determine the GPU% based on current inference latency and throughput to best match the SLO for latency and improve throughput the most (line 12/13). The output feeds into a MAX-MIN fair GPU resource allocation algorithm which computes the GPU% to IFs based on their demand, with the demand of IF's requiring

Algorithm 1: GPU Resource Tuning Algorithm

Input: *currentGPU%*, *SLO*, *arrivalRate*
Function `ComputeNewGPU%()`:
  avgL←LatencyHistogram.EWMA()
  avgT←ThroughputHistogram.EWMA()
  residualL←SLO − avgL
  residualT←avgT − arrivalRate
  diffL%← $\frac{residualL \times 100}{SLO}$
  diffT%← $\frac{residualT \times 100}{arrivalRate}$
  if *(diffL% < threshold) && (diffT% < threshold)* **then**
    return *currentGPU%*
  end
  changeFactor ←MAX$\left(\frac{abs(residualL)}{avgL}, \frac{abs(residualT)}{avgT}\right)$
  changeGPU% ← currentGPU% × changeFactor
  if *(residualL < 0)* || *(residualT<0)* **then**
    newGPU% ← currentGPU% + changeGPU%
  end
  if *(residualL > 0) && (residualT>0)* **then**
    newGPU% ← currentGPU% − changeGPU%
  end
return *newGPU%*

lowest GPU% being fulfilled first. To avoid oscillations, an IF's GPU% is reallocated only when the change is above a threshold (we set it to 5% as default).

3.2.3 Resource Allocation Manager. During initialization, each IF indicates the desired model to the IF Manager and gets the portion of the GPU it can use. This communication and setting of the GPU% is completely abstracted within *libml* and performed transparently without any explicit involvement of the IF. The IF Manager looks up the model repository and identifies the optimal GPU% ('kneepoint') for the requested model - this serves as the 'elastic demand' for the model.[6] Further, we choose a conservative value of 30% (a compile time configurable parameter) for IF models that do not have an apriori profiled GPU%, and our self-tuning resource allocation allows the system to dynamically determine the appropriate GPU% based on the observed latency, throughput and arrival rate. Our resource allocation scheme provides a weighted (the weight being the newGPU% required by an IF in Algo. 1) allocation as follows:

- Resources (GPU%) are allocated to IFs based on demand, maximizing the minimum resource allocation for any IFs with unsatisfied demands. The unsatisfied IFs get resource shares in proportion to their normalized (sum of newGPU%) weights across all currently running IF models.
- No IF obtains a resource share larger than its demand, unless the GPU is underutilized.

Note: Any new IF instance addition or removal triggers a resource reallocation and the resources for other active IFs are readjusted (as we show, with very little downtime).

3.2.4 IF resource adjustment with zero downtime. To meet the SLO and varying traffic demands, it is necessary to re-provision the GPU resources so that we can maximize GPU utilization and IF throughput. This is non-trivial because after computing the correct GPU%, we have to restart the IF process and reapply the GPU resources for the IF[7].

In order to amortize the long startup cost, we developed a low-cost 'shadow IF' and 'overlapped execution' mechanisms. They combine to transparently re-provision the GPU resources of an IF with a quick switchover of processing to the 'shadow IF'. Our evaluations show the mean idle time for this IF switchover is less than 100μ seconds.

Shadow IF: For every IF (primary/active), we also instantiate a shadow IF (hot-standby). Shadow IFs share the same resources (buffer pool, packet ring buffers, ML framework, CPU core) as the primary, but the key difference is that they do not access the GPU and packet ring buffers. Note: After initialization, the shadow IFs do not consume any CPU until they are explicitly transitioned to active state by the IF Manager. Thus, they have no adverse impact on the performance of the primary IFs. The shadow IF masks the instance initialization (DPDK and ML framework) time and brings down the GPU load time of IF model. However, this solves only half the problem, as the GPU and ML platform (*e.g.,* PyTorch) initialization, still incur delays of the order of a few seconds. Therefore, we let the 'Shadow IF' perform GPU initialization and overlap its execution with active IF.

Intermediate load: Depending on the type of ML framework, the shadow IFs load the IF model to CPU and immediately yield[8]. This intermediate loading of the IF model on the CPU helps to significantly reduce the load time to GPU by exploiting the CPU cache memory and avoiding disk reads and re-serialization of the model data. Only after the IF Manager provisions the GPU resources and wakes up the shadow IF, does it continue loading the model to GPU.

Overlapped Execution: Upon a GPU% readjustment, the IF Manager configures the GPU%, wakes up the shadow IF and switches its role to being the primary/active IF. This requires strict co-ordination with the previous primary/active IF to ensure correctness and avoid data corruption (network packets, which reside in common buffer pool). While the shadow IF model is loaded into the GPU to become ready for execution,

[6] The demand is considered elastic because the IFs can get more or less than their demand's GPU%.

[7] Currently, this stems from the limitation in NVIDIA, CUDA MPS, and is likely to remain for Volta and the upcoming Turing GPUs [8].

[8] The only exception being the TensorRT framework; TensorRT optimized models lack the support for loading on CPU.

the previous primary/active model continues to execute. The IF Manager waits till the previous primary IF completes the current inference completely. It then handles the transitioning and switch over of roles for the primary and shadow IFs in co-ordination with *libml*. It switches the shadow IF to 'active' and eventually terminates the previously active IF. This final coordination by manager incurs around 50-60μ seconds (idle time). Overall, this mechanism masks the IF's startup time (order of 2-15s) and improves overall system performance.

3.3 How many CUDA Streams?

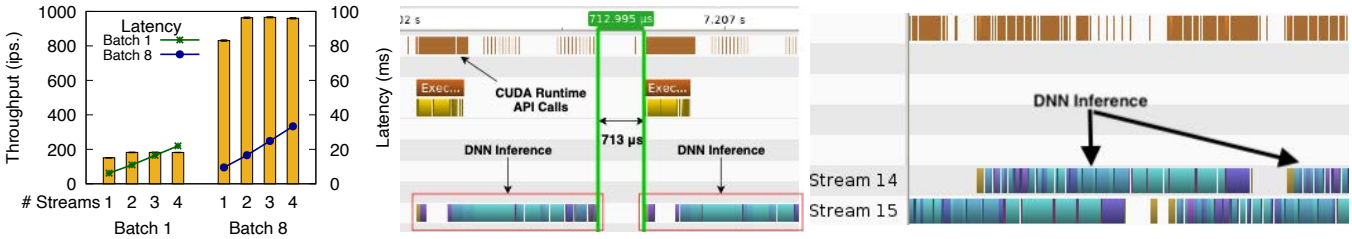
Although, CUDA Streams improve GPU utilization and overall throughput, they have an adverse impact on latency. We show through experiments with different batch sizes, the impact of using multiple streams. In particular, we see the results for ResNet-50 model with TensortRT in Fig. 5a. Beyond 2 streams (for batch sizes of 1,8), there is barely any improvement in throughput. But, the latency keeps increasing. We observed the same with VGG-19. We saw that Alexnet was an exception that showed improvement with more streams, but only for the small batch size (1). Our analysis is that streams are helpful only when there are sufficient resources to take advantage of multiplexing. Otherwise, the tasks contend for the limited resources resulting in high execution overhead, reflected by the latency increase. Hence, streams are beneficial only for light weight models and lower batch sizes. However, we do see some drawbacks of using a single stream for processing. We profiled for the single stream case for Alexnet, as shown in Fig. 5b. The GPU remains idle for a fairly large amount of time, ~700 μs, between every execution. This corresponds to the time taken to notify the CPU of inference completion and the processing on the CPU side to perform cleanup and return the callback. Note that, until the callback is completed, the CUDA driver does not launch the next inference execution, even if the tasks were queued before. Fig. 5c shows the profiler results with two streams for Alexnet. We still observe similar gaps in each of the streams, but since the two streams have overlapped execution, the GPU utilization is higher in this case.

However, the streams are problematic while multiplexing models with unequal compute requirement e.g., ResNext-50 running concurrently with Alexnet. In such cases, streams do not provide any meaningful overlap in the execution of the kernels of the distinct streams, as seen in Fig. 6. Here, ResNext-50's kernels (stream 14) do not overlap with the execution of Alexnet's kernels (stream 16). Thus, the streams are restricted to time-share share the GPU, resulting in increased inference task completion time. This causes lower utilization of GPU as well, since only one application is running at a time in the GPU. We can extract higher GPU utilization by rather running each DNN as an individual application and spatially sharing the GPU using MPS with resource usage limits. Fig. 7 shows that ResNext-50 and Alexnet can concurrently run in the GPU if we run them with MPS with a resource usage limit. This allows for much better utilization of GPU as well as lower latency, as one DNN's inference does not have to wait for other's to end. Therefore, in GSLICE we prefer using MPS with resource provisioning instead of CUDA streams, whenever possible. We limit to 2 streams in the cases where streams can help boost the throughput.

3.3.1 Self-learning Adaptive Batching. A larger batch size improves throughput and GPU utilization (amortizing the interactions between CPU and GPU). A naive approach would be to batch as many requests as possible till the previous inference operation completes. This approach yields high throughput, but can result in high latency. Alternatively, we can learn and adapt (limit) the batch size to be just sufficient to meet the SLO. This adaptive batching, used in Clipper and Nexus [21, 52] works well when the GPU is temporally shared. However, when the GPU is spatially shared, several additional factors also contribute to latency, namely i) variation in provisioned GPU resources; ii) multiple streams that contend to spatially share GPU; iii) interference due to concurrently executing GPU tasks, with MPS. Most of these factors are dynamic in nature and can happen outside the scope of an individual IF. Therefore, we develop a 'self-learning' approach to dynamically adapt the maximum operational batch-size.

Self-learning Adaptive Batching (SLAB): On the host CPU, *libml* tracks the inference completion time of each batch and heuristically determines the batch size that allows us to operate within the SLO. SLAB works by increasing or decreasing the batch size in proportion to the current batch size, observed latency and deviation from the specified SLO, by checking the headroom (SLO - avg. latency for the previous batch). Thus, it quickly readjusts the batch size. An IF is considered to be underprovisioned when it misses its SLO even with a batch size of 1. Instead, if the IF operates within the SLO, then the IF may be overprovisioned. *libml* tracks (5 successive) such occurrences and notifies the IF Manager to readjust (increase/decrease) its GPU resources.

3.3.2 Data Aggregation and Transfer to GPU. DNN applications such as Imagenet models require the entire image's data (and all of the batch, if batch size is > 1) to be available in a contiguous GPU memory buffer before an inference is started. To detect requests that are ready, GSLICE implements a CPU side zero-copy aggregation list of network packets (*i.e.,* store only the packet pointers in an indexed array) and tracks aggregation status for up to 64 distinct requests (as bit fields). This allows quick detection of all ready requests (images, text, etc.) to be transferred to the GPU. We buffer packets until the IF operation completes. Alternatively, we could release the packets as soon as the data is copied to



(a) Throughput & latency (b) Inference with 1 CUDA stream (c) Inference with two CUDA streams

Figure 5: (a)ResNet50 throughput & latency w/multiple streams (b),(c) Profile of Alexnet (TensorRT) using streams

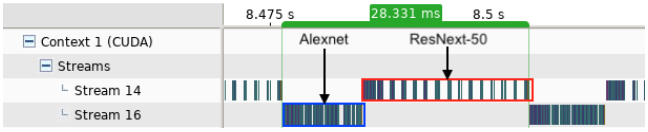


Figure 6: Sequential kernel execution with streams

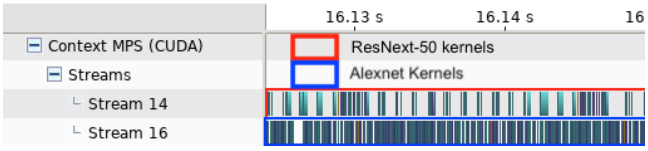


Figure 7: Overlapping inference: MPS+resource limits

GPU, requiring a callback from GPU to notify copy completion. This is expensive (40-100μs). We found it has negligible benefit, and sometimes detrimental because of host-side context switching to process CUDA driver callbacks.

Data transfer to GPU: To efficiently transfer data without taxing the host CPU, we pin the DPDK hugepages (used for network packets) with CUDA. Then, using NetML's approach [23] the GPU's DMA performs zero-copy scatter-gather of the packet data using a GPU kernel. An alternative is a NIC to GPU transfer using GPUDirect. However, GPUDirect places all ingress packets into GPU memory and the GPU kernels have to perform packet processing. This has many limitations, such as inefficient packet processing in GPU [55]. Using NetML's approach [23], we only transfer data for ML processing to the GPU, leaving packet processing to the CPU. This avoids the CPU copy overhead.

3.3.3 Shared Inference Parameters. The IF Manager includes a model loader component that loads the inference model related parameters only once to the GPU and allows reuse of those parameters across different instances of the same IF. This has two benefits: i) increased multiplexing due to reduced GPU memory footprint per IF; ii) faster module loading due to reuse of already mapped GPU parameters.

ML frameworks (*e.g.*, Pytorch, CNTK), export APIs to retrieve the parameters (weights and biases) of the DNN models and their GPU addresses. The IF Manager takes advantage of these platform specific API's to build the GPU address mapping for the parameters of different models. It then exports these GPU addresses to different IF instances through the CUDA API `cudaIpcGetMemHandle()`. We extended the CNTK and Pytorch libraries to make them attach parameters to the specified GPU address. These changes are minimal (less than 30 lines of code).

Table 1: Benefits of Parameter Sharing (PS) (CNTK).

	Alexnet	Resnet-50	Resnet-152	VGG-19
Model Parameters size (MB)	240	98	232	549
SA Load time (ms)	781	430	747	1561
PS Load time (ms)	97	89	103	107
SA mode GPU Memory (MB)	1037	917	1101	1389
PS mode GPU Memory (MB)	795	805	805	825
SA num. of instances	15	17	14	11
PS num. of instances	18	18	18	17

Table 1 shows the benefits of parameter sharing (PS) in comparison to standalone (SA) mode of operation. Parameter sharing is both time and space efficient, enabling us to swiftly load the models to GPU with 8-10× reduction in load time, and fit 5%-54% more IF instances.

3.4 Implementation Details

We implemented GSLICE as a DPDK [1] based platform. Overall, implementation of GSLICE is ~2.5k lines of 'C/C++' code. We minimally enhanced the CNTK and PyTorch libraries to support parameter sharing and improve inference performance. Key extensions to CNTK and PyTorch include:

- **Parameter Sharing:** We extend ML frameworks to attach ML parameters from a specified GPU address - helps minimize startup time and memory footprint.
- **Reuse of GPU I/O buffers:** We allocate the input and output buffers only once (typically done for every inference operation), and reuse the existing GPU buffers for subsequent inference operations - helps avoid expensive GPU memory alloc/dealloc for each inference operation.
- **Share GPU I/O buffers across multiple processes:** We enable sharing of GPU I/O buffers directly across processes (Primary and Shadow IF) - helps overlap execution and eliminates multiple data copy overheads.

4 EVALUATION

Our experimental testbed uses Dell PowerEdge R740xd servers with Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz dual-socket

CPUs with 20 cores each, 252GB RAM and includes one NVIDIA Tesla V100 GPU and quadport Intel XL710 10 GbE NIC. The Tesla V100 GPU has 16 GB of memory, 80 Streaming Multiprocessor and 640 Tensor Cores. The server node runs Ubuntu SMP Linux kernel 4.4.0-150-generic with DPDK v19.02. We used Moongen [24] and tcpreplay [10] as workload generators to transmit data at up to 30 Gbps. We used 3 ML frameworks, CNTK, PyTorch and TensorRT. We ran imagenet models in CNTK (Table 1) and TensorRT (§ 4.1). GNMT and Jasper in Pytorch. We chose Alexnet (1 GFLOPs), ResNet-50 (4 GFLOPS), VGG-19 (20 GFLOPs) [14] and to represent diverse GPU computational loads. Our IF workload consists of color images based on Imagenet dataset [50], with resolution of 224×224, and size of 588 KB transmitted as 588 UDP packets (1 KB each); and for GNMTv2, 5-word sentences as packet payloads.

4.1 GSLICE: System Performance

First, we demonstrate the effectiveness of GSLICE in improving overall inference performance and GPU utilization, and compare it with default MPS (as baseline) and batched mode of MPS 'MPS+SLAB' (M+S). In this experiment, we run six distinct IFs (Alexnet, Mobilenet, ResNet-50, VGG-16,VGG-19 & GNMT) in isolation. We choose an SLO of 25ms to illustrate how GSLICE can meet a timeliness requirement, *e.g.*, for a real-time system processing videos at 30 frames/sec.

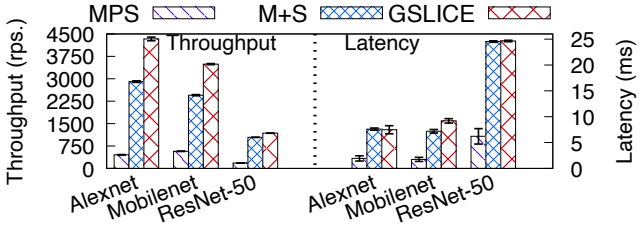


Figure 8: Performance of GSLICE and Default MPS (1)

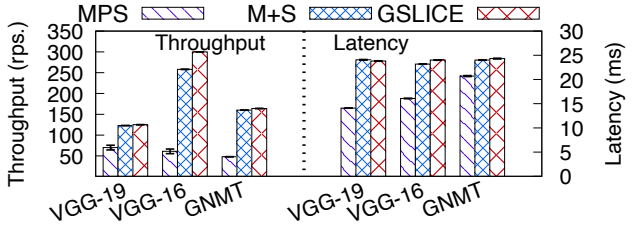


Figure 9: Performance of GSLICE and Default MPS (2)

Table 2: Measure of GPU Utilization Efficacy (GUE (η))

IF model	Alexnet	Mobilenet	ResNet-50	VGG-19	VGG-16	GNMT
Default MPS (η)	235.75	330.0	29.08	4.96	3.79	1.97
MPS + SLAB (M+S) (η)	382.4	341.2	42.4	5.11	11.13	6.66
GSLICE (η)	579.41	379.55	47.93	5.25	12.49	6.74
Improvement M+S (%)	62.54	3.36	46.11	2.92	193.32	187.23
Improvement GSLICE (%)	146.28	14.99	65.18	5.71	229.25	190.83

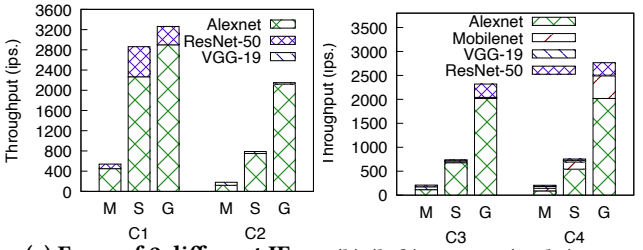
Figure 8 and 9 show the results for GSLICE (throughput (left) and latency (right)). For clarity, we split IFs with different complexities into two groups. The error bars indicate 95% confidence interval. GSLICE achieves significant improvement over 'default MPS' in throughput for Alexnet (~10×), and Mobilenet (~9×). VGG-19 (a computationally heavy model) shows ~2× improvement. Improvement comes from more effective GPU utilization with GSLICE. While we observe increased latency with GSLICE and 'M+S', judicious batching limits latency to be within the specified SLO. Since only one model executes at a time here, throughput is primarily improved by batching.The incremental benefit of GSLICE, beyond just our adaptive batching applied to MPS (M+S), is higher for lighter models, Alexnet, Mobilenet and VGG-16. However, GSLICE significantly improves throughput and latency, beyond the benefits from adaptive batching, with multiple heterogeneous models running concurrently (see below). GSLICE improves GPU utilization efficacy (GUE) across all IFs, compared to 'default MPS' by 5-229% (Table 2).

4.2 Concurrent execution of multiple IFs

Table 3: Measure of GPU Utilization Efficacy (GUE (η))

Concurrent IFs	Alexnet & ResNet-50	Alexnet & VGG-19	3 IFs	4 IFs
Default MPS (η)	38.82	6.39	3.38	2.01
MPS + SLAB (M+S) (η)	72.69	15.76	7.83	7.29
GSLICE (η)	94.04	44.79	24.59	19.98
SLAB Improvement (%)	108.7	146.65	131.74	261.41
GSLICE Improvement (%)	170.02	600.79	627.67	890.10

We demonstrate the benefit of GSLICE in multiplexing different IFs in the GPU. We use combinations of 2, 3 and 4 IFs, which run concurrently in the GPU, while achieving an SLO of 25ms with varying GPU demand/load.



(a) Exec. of 2 different IFs **(b) (left) 3 IFs & (right) 4 IFs**

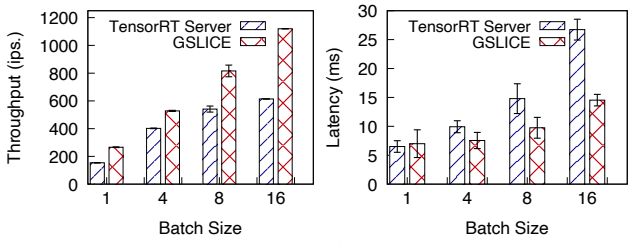
Figure 10: Concurrent execution of multiple IF combinations: C1 (Alexnet & ResNet-50), C2 (Alexnet & VGG-19), C3 (C1 & VGG-19), C4 (C3 & Mobilenet) with Default MPS (M), SLAB (S) and GSLICE (G)

Fig. 10a shows the combined throughput (with breakdown by IF) of two IFs, Alexnet and ResNet-50 (Fig. 10a(left)) and Alexnet and VGG-19 (Fig. 10a(right)). A similar experiment with 3 different concurrent IFs (Alexnet, ResNet-50 & VGG-19) is shown in Fig. 10b (left) and 4 different IFs (Alexnet, Mobilenet, ResNet-50 & VGG-19) in Fig. 10b (right). GSLICE outperforms both 'default MPS' and 'M+S' to provide ~6-13× and 1.2-4× improvement in throughput respectively, across all the combinations. We note in Fig. 10b that the throughput of VGG-19 in GSLICE (22 images/sec) is lower than in SLAB (32 images/sec). In GSLICE, an IF's share of

the GPU resource is determined using a Max-min fair share algorithm to maximize overall system performance without penalizing any IF more. There is some penalty for compute heavy DNN models, compared to letting them grab all the GPU resources at the expense of other models, as in the SLAB experiment. Similarly, Table 3 shows that GSLICE provides ~1.7 to 9× improvement in GPU Utilization efficacy. Thus, GSLICE provides improved, consistent throughput and performance isolation for each IF by a judicious allocation of a fixed GPU% (refer §3.2.3).

4.3 Comparison with TensorRT Server

We focus on single GPU to present subsequent benefits and use case demonstrations for simplicity. We compare GSLICE with state-of-the-art NVIDIA TensorRT (Triton) Server ver.19.02 [44]. For this experiment, we used the performance evaluation tool published by NVIDIA [3]. For a fair comparison, only for this experiment, we restrict GSLICE's adaptive batching to not exceed a configured maximum batch size. Note: the throughput and latency shown are purely for the execution time of a ready batch on the GPU. Thus, we eliminate any differences as a result of the overheads due to HTTP/TCP in TensorRT and UDP processing in GSLICE.



(a) Throughput (ips). **(b) Latency (milliseconds).**

Figure 11: Comparison with Triton (ResNet-50 model)

Table 4: GSLICE's Improvement vs. TensorRT server

Batch size	1	4	8	16
Throughput Improvement (%)	73.85	31.43	50.83	92.34
Latency Improvement (%)	-7.52	23.96	34.12	45.64

From Fig. 11a, we observe that GSLICE achieves higher throughput across all batch sizes. Especially for large batches (4..16), GSLICE is able to achieve 31-92% higher throughput than NVIDIA TensorRT server and 23-45% lower latency for larger batches. Also, in Fig. 11b, GSLICE outperforms TensorRT server in providing low-latency service for most batch sizes. Table 4 summarizes the performance improvements of GSLICE. Especially, with the largest batch size (16), GSLICE provides almost 2x improvement in throughput with about half the latency of TensorRT server. TensorRT server provides slightly lower latency for batch size of 1, but has much lower throughput than GSLICE. Note that although we restrict batch size in both cases, GSLICE incorporates adaptive-batching and two streams that seek to maximize GPU occupancy and minimize idling of GPU (*i.e.,* avoids wait for a batch to complete), which is evident from the lower latency with GSLICE. In addition, Zero-Copy GPU scatter-gather offloads the CPU to further reduce the batching overhead, resulting in overall better throughput.

4.4 Benefit from each GSLICE component

To illustrate the effectiveness of GSLICE's proposed techniques, we run the same experiment as in §4.1 and enable each component of GSLICE individually, viz., multi-streams, Zero-Copy GPU scatter-gather and adaptive-batching. We set the SLO to a nominal value of 25 ms. Figs. 12a and 12b show the throughput and median latency respectively.

Table 5: Throughput & Latency, different batch sizes

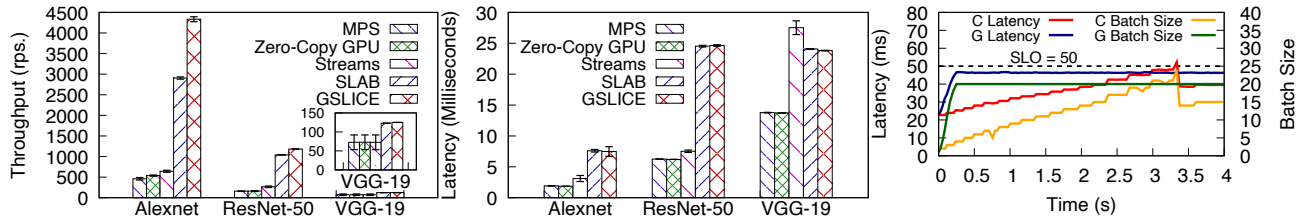
Model	Batch = 8		Batch = 16		Batch = 32	
	Thpt.(ips)	Lat. (ms)	Thpt.	Lat.	Thpt.	Lat.
ResNet-50	816	9.75	1088	14.5	1152	25.9
VGG-19	264	30.54	400	40.95	512	63.12

With multi-streams (we limit to using only 2 CUDA streams) alone, the lighter models Alexnet (39%) and Resnet-50 (66%) show throughput improvement. With the heavier model VGG-19, as the computation load dominates, the throughput contribution from streams support is limited. But, multi-streams does increase latency. Nonetheless, unlike single stream, multiple streams avoid idling the GPU during the execution of the GPU callback.

The zero-copy approach helps improve both throughput and latency, although it depends on the complexity of the IF model. For Alexnet, the data transfer takes about 100 μs, which is a significant percentage of the inference time of ~1 millisecond. We see about a 17% increase in Alexnet's throughput (Default MPS: 460 images/sec, Zero-Copy GPU: 540 images/sec) in Fig. 12a, and the latency of inference also decreases slightly. However, the improvements are marginal for more compute intensive models like ResNet-50 and VGG-19, which spend more time (~10-15ms) for DNN computation, dwarfing the data movement time. Nonetheless, the main benefit of Zero-Copy GPU scatter-gather is offloading the CPU, avoiding it from becoming the bottleneck.

We note that batching a number of inference tasks improves throughput. But, this is only up-to a point where all SMs of a GPU are used. Forming and inferring larger batch sizes hurts inference latency, which can be a concern for real-time operations. Table 5 shows the impact of batching on throughput and latency for ResNet-50 and VGG-19 models on our testbed. We can see the throughput improvement is much higher from Batch 8 to 16. However, from 16 to 32 the throughput improvement is smaller, but the latency increases quickly for both models. Thus, to balance between throughput and latency, we utilize adaptive batching in GSLICE.

Adaptive batching helps improve throughput for all models. In fact, the heavier the model (e.g., VGG-19) the larger the improvement. With lighter models (e.g., Alexnet, ResNet-50),



(a) Throughput (ips). (b) Latency (milliseconds). (c) Batch size tuning at startup

Figure 12: (a), (b) Benefits of individual GSLICE components. (c) startup comparison: GSLICE (G) vs. Clipper (C)

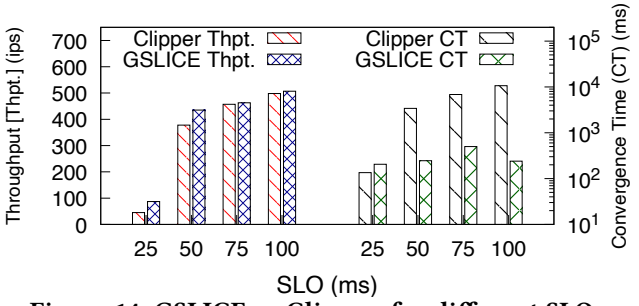


Figure 14: GSLICE vs Clipper: for different SLOs.

the improvement is still good, as shown in Fig. 12a. However, we do observe an increase in latency (2-3x) for all the models compared to the baseline MPS with each of our improvements, especially with VGG-19 and adaptive batching. We note that although the latency is higher than default MPS, we can still maintain an SLO of 25ms. GSLICE is able to use all the available time-budget to right-size the batch, maximizing throughput while limiting the latency increase.

4.5 Self-learning Adaptive Batching

We compare GSLICE's adaptive batching with another inference system, Clipper [21]. We integrated the adaptive batching code from Clipper's Github repository [4]. We use a VGG-19 on TensorRT IF (with latency ~14ms for a batch size of 1) and vary the SLO across different runs.

Fig. 12c shows the convergence time of the two adaptive batching schemes for SLO of 50ms. We chose a higher SLO of 50 ms with IF running VGG-19 to give it more time to form a bigger batch. VGG-19 being a compute heavy model would only form very small batch with SLO of 25ms. GSLICE uses the available latency headroom to rapidly increase the batch size (in less than 300ms) to the maximum that would not violate the SLO. On the other hand, Clipper's batch size increases gradually and takes ~3.5s to reach the right value. With Clipper, the convergence time keeps rising with higher SLOs, and exceeds 10s for SLO of 100ms, while GSLICE quickly converges within 500ms in all the cases as shown in Fig. 14 (right).

We also show the impact on throughput for different values of the specified SLO as shown in Fig. 14 (left). GSLICE shows ~2 to 50% throughput improvement across the measured cases. At lower SLOs (25-50ms), we observed Clipper to be sensitive to SLO violations (due to multiplicative decrease) and converge to a relatively smaller batch size, hurting throughput. However, GSLICE finds the right batch size to operate within the SLO and provide higher throughput.

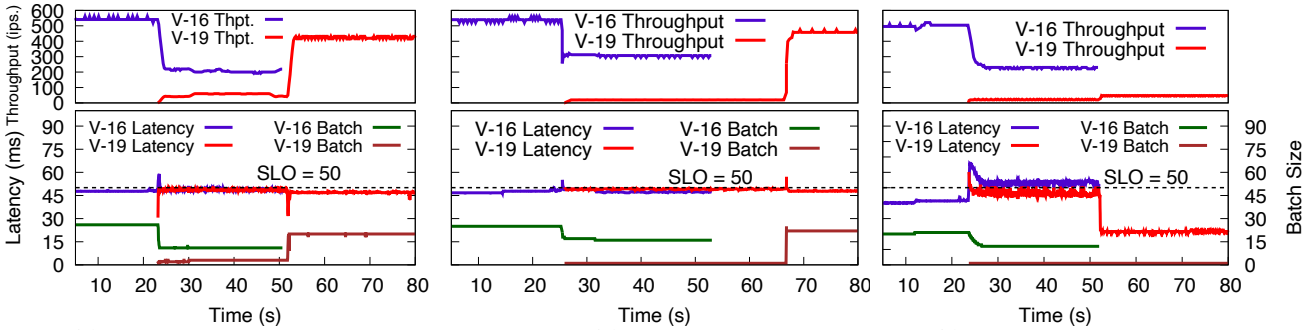
4.6 GPU resource re-provisioning

4.6.1 Adaptation to workload variation. To show GSLICE's ability to adapt to changing workload, we dynamically vary the workload presented to the GPU. We start with a VGG-16 IF active with an SLO of 50 ms. After 20s, we launch another VGG-19 IF. We run both (VGG-16 & VGG-19) IFs together and terminate the first IF (VGG-16) at 50s. Figure 15, shows the timeline of events, the observed batch size, throughput, and latency for each execution round of the two IFs.

With SLAB using default MPS (both IFs are provided with 100% GPU), shown in Fig. 15a, VGG-19 IF gets a throughput of ~550 at the start (and remains below the SLO of 50 ms). The IF's throughput drops to ~200 when VGG-19 IF starts at 23-sec mark. When the VGG-16 IF stops at 53 sec, the batch size and throughput of VGG-19 IF increases quickly.[9] Thus, adaptive batching in SLAB adjusts the batch size to utilize freed up GPU resources.

In Fig. 15b we further demonstrate GSLICE's capability to provide proper GPU isolation. We run the same experiment with GSLICE, which provides 100% GPU to first IF (VGG-16). We maintain the SLO of 50 ms and high throughput. When the VGG-19 IF starts, it restricts the GPU resources for VGG-19, to 30%, and provides the remaining 70% GPU allocation to VGG-16. We can see from Fig. 15b, that throughput of VGG-16 still remains quite high (~300) compared to the default MPS case even when the VGG-19 IF comes up. Once, the VGG-16 IF stops at 53 sec mark, GSLICE starts the process to increase the GPU% for VGG-19 IF by getting the Shadow VGG-19 IF ready with 100% GPU. The process of switching to a different GPU% is not instantaneous, as the shadow IF takes some time (~ 12 sec) to load the ML model into the GPU. However, the active 'primary' VGG-19 IF continues to infer new requests, with minimal downtime. When the shadow IF is ready to execute, at 66 sec., GSLICE almost

[9] To begin with VGG-16 has 100% GPU share, and once VGG-16 terminates, VGG-19 IF operates with 100% GPU share.



(a) SLAB with Default MPS (b) GSLICE. (c) Clipper with Default MPS

Figure 15: Dynamic adaptation of batch size; [VGG-16 (V-16) & VGG-19 (V-19)] w/Default MPS, GSLICE & Clipper

seamlessly switches (in about 100 μsec.) to the shadow VGG-19 IF, utilizing 100% GPU and quickly increases batch size and throughput. Thus, GSLICE achieves resource isolation and avoids the interference between IFs and improves throughput without impacting the SLO.

Finally, we compare with Clipper in Fig. 15c, which gets an initial throughput of ~500 with VGG-16 at a latency below the SLO of 50ms while running alone in the GPU. When the VGG-19 IF starts at ~23 secs., the VGG-16 model suffers from interference and struggles to maintain the SLO, and throughput drops to ~200 images per sec. When, VGG-16 stops at 53 sec., VGG-19's throughput increases slightly as more GPU resources are freed up for it. However, Clipper's batch size remains the same and the throughput remains at almost the same low value. This is due to the fact that Clipper's adaptive batching system is only reactive to violating the SLO, but once stabilized at a value, does not readjust to capitalize on now free resources, unlike GSLICE's continual adaptation.

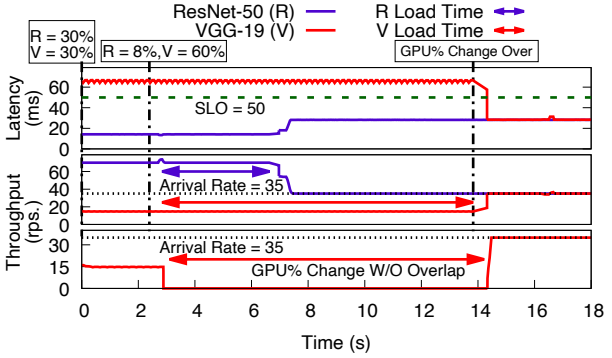


Figure 16: Self-tuning of GPU Percentage for two IFs.

4.6.2 Self-Tuning of GPU proportion Allocation. We extend the previous experiment (§4.6.1) to demonstrate GSLICE's ability to re-allocate GPU resources for the active IFs in a self-tuning manner, adapting to the demand without impacting their performance. We assume the 'knee' GPU percentages of the IF models are unknown, with no apriori profiling. We start with two models, VGG-19 and ResNet-50, allocating each IF with (somewhat arbitrarily chosen) 30% GPU, each receiving requests of 35 images/sec. as shown in Fig. 16. As the IFs process the inference requests, it is evident that VGG-19 IF is underprovisioned, as the throughput (15 ips, Fig. 16 middle plot) is below the request arrival rate of 35 ips. The latency (65ms, Fig. 16 top plot) is above the 50 ms SLO. On the other hand, ResNet-50 IF's throughput easily exceeds the arrival rate and GPU resources are over-provisioned. At the 2 sec. mark, GSLICE's manager which periodically employs a heuristic to compute GPU% that can best serve the request rate and the SLO for both of the IFs, begins adjusting the GPU percentage of ResNet-50 to 8% and VGG-19 to 60%. While the shadow IFs are being setup with their new GPU%, both active IFs continue processing incoming requests. The ResNet-50 shadow IF is loaded into the GPU with the new GPU% earlier (at 7 sec). GSLICE's manager switches the ResNet-50 active and shadow IFs. Eventually, the more complex VGG-19 IF completes loading the ML model to the GPU (14 sec). With GSLICE's self-tuning having increased GPU% for VGG-19, it now processes all incoming requests while maintaining the SLO. This self-tuning capability of GSLICE ensures that VGG-19 gets the required GPU share while not over-provisioning the ResNet-50 model either.

We compare our result with an implementation where a new IF instance is started from scratch to effect change in GPU% (Fig: 16 (bottom plot)). If the IF percentage is changed without utilizing our innovative overlap technique, no requests are served for a long interval - until the new IF is loaded and ready. This dramatically hurts servicing inference requests and degrades overall throughput.

5 RELATED WORK

GPU based Inference Platforms: Clipper [21] is a low-latency inference system providing a common model abstraction API, and predictive IF model & ML framework selection service to better suit user requests. It uses an AIMD-based adaptive batching scheme and delayed batching modes to optimize the batch size to meet a target SLO. GSLICE goes beyond Clipper to support heterogeneous ML frameworks and models to run concurrently by spatially sharing the GPU. GSLICE also provides a variant of adaptive batching that can swiftly adapt to GPU resource variations.

Nexus [52] and Themis [41] create an inference service with a cluster-wide GPU resource management framework. Nexus proposes batching-aware scheduling using prefix adaptive batching, and early drop mechanisms to improve overall performance and utilization for a GPU cluster. Nexus scheduler relies on temporal-only sharing of GPUs, with the adaptive batching operating on a fixed batch size for each epoch (30s). Themis proposes a two-level GPU scheduling approach with fixed batch size and a novel fairness measure called 'finish-time fair allocation' for the GPU cluster. Both Nexus and Themis operate at a higher management plane (cluster level management), while GSLICE spatially shares a GPU to maximize GPU utilization and improve IF application performance. Gandiva[59] is a DNN learning and inference architecture that increases throughput by 'packing' DNN applications in the GPU. Packing in Gandiva corresponds to temporally multiplexing applications. Temporal sharing increases utilization but not to the extent of spatial sharing. GSLICE uses controlled-spatial-multiplexing to share the GPU with applications and increase GPU utilization.

TensorRT [57] by NVIDIA Inc. is a high-performance deep learning inference library for production environments [26]. NVIDIA also released the container runtime for Docker with TensorRT support [2]. However, the current release does not yet fully support MPS. We used the publicly available TensorRT libraries and built the TensorRT IF on our platform with MPS support. PowerAI Vision Inference Server [33] by IBM Inc. is a new generation of video/image analysis platforms that can deploy both the image classification models and object detection models. EdgeEye [40] presents an edge-computing framework for real-time intelligent video analytics applications. It uses NVIDIA TensortRT inference engines to improve inference performance, and provides a high-level, task-specific API for developers. Unlike GSLICE, their focus is to provide a framework for real-time video analytics.

Works on CUDA MPS: NVIDIA's latest Ampere GPU has Multi-Instance GPU(MIG), where the GPU resource partitioning goes beyond SMs and includes memory partitioning [7]. Ampere keeps the underlying CUDA platform same, so we expect optimization in GSLICE would be beneficial in Ampere GPU as well. In [34], authors identify the performance gap issues with GPU resource sharing (temporal and spatial), and propose a dynamic space-time multiplexed scheduling to optimize the GPU inference throughput. It also tries to prioritize predictability, but proposes to micro-manage the kernels that execute on GPU, monitor the latency for each kernel execution and control the eviction of degraded task. GSLICE is simpler, providing platform level optimization mechanisms that are non-intrusive (no need to micro-manage kernels), and readily leverage and deploy available IF frameworks.

ML Acceleration: Several works optimize and produce lightweight versions of the DNN models [25, 43, 62] to make DNNs fast and less compute intensive.SqueezeNet [32], has fewer parameters and a small model size. Others accelerate ML training and inference with binarized neural networks [20] or compress the DNN model[28]. There are hardware accelerators such as NVIDIA's tensor cores[8], Google's TPU[36] and Eyeriss[17] made to speedup DNN processing.

GPU accelerator functions: Many works leverage GPUs to accelerate packet processing. PacketShader[27] utilizes GPUs to process packet headers for switching and routing, and SSLShader[35] provides high throughput SSL processing in the GPU. Similarly, NBA[37] presents an adaptive load balancer to balance the workload of network functions (NFs) running on both CPU and GPU. APUNet [25] uses integrated GPU to process packets and eliminates the data transfer over PCIe bus. G-NET[61] is a scheduling and virtualization framework to share GPU resources across multiple NFs. G-NET uses Hyper-Q to spatially share the GPU, and provisions the GPU SMs for each of the NFs. In contrast, GSLICE utilizes MPS to provision GPU. Using MPS allows GSLICE to support low level ML libraries such as cuDNN[18] and cuBLAS[5] which do not expose thread blocks information and prevent the use of thread block counting technique to provision GPU.

6 CONCLUSIONS

We presented GSLICE, a platform supporting cloud-based low-latency inference applications. GSLICE supports a number of ML frameworks and IF models, and specifically addresses the challenges of efficiently utilizing the GPU while multiplexing several concurrently running streaming IFs. GSLICE builds on top of CUDA MPS to provide controlled spatial sharing of the GPU across multiple IF models to ensure performance guarantees. GSLICE's 'self-tuning' resource allocation scheme dynamically adjusts and apportions just the right amount of GPU resources for the IFs to meet their SLO and demand. GSLICE's 'Self-Learning Adaptive Batching' helps maximize GPU utilization and IF throughput without violating latency SLOs. Using a shadow IF and an efficient overlap technique, GSLICE masks the high startup costs (2-15s) and re-provisions GPU resources and instantiates new IFs with less than 100μs downtime. GSLICE also improves IF performance and reduces the overhead on CPU through data aggregation and efficient data transfer mechanisms. Parameter sharing in GSLICE improves scalability by reducing IF memory footprint and helps multiplex multiple instances of an IF. Overall, GSLICE dramatically improves GPU multiplexing (5-54%) and utilization efficacy (up to 800%) to achieve 2-13× overall IF throughput improvement.

Acknowledgements: We thank the anonymous reviewers and our shepherd, Dr. K. R. Jayaram for their valuable feedback and the US NSF for their generous support of this work through grant CNS-1763929.

REFERENCES

[1] 2014. Data plane development kit. http://dpdk.org/. [online].

[2] 2019. NVIDIA Container runtime for Docker. https://github.com/NVIDIA/nvidia-docker. (2019). [online].

[3] 2019. NVIDIA TensorRT Inference Server. https://github.com/NVIDIA/tensorrt-inference-server. [online].

[4] 2020. Clipper Github. https://github.com/ucbrise/clipper.

[5] 2020. CUBLAS LIBRARY. https://docs.nvidia.com/cuda/cublas/index.html. Accessed: 2020-02-19.

[6] 2020. Metal Documentation. https://developer.apple.com/documentation/metal. Accessed: 2020-04-25.

[7] 2020. NVIDIA Ampere MIG. https://www.nvidia.com/en-us/technologies/multi-instance-gpu.

[8] 2020. NVIDIA Tesla V100 GPU Architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: 2020-02-01.

[9] 2020. ROCm Github. https://github.com/RadeonOpenCompute/ROCml. Accessed: 2020-04-25.

[10] 2020. tcpreplay Github. https://github.com/appneta/tcpreplay.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[12] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. 2017. Real-time video analytics: The killer app for edge computing. *computer* 50, 10 (2017), 58–67.

[13] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–43.

[14] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. 2018. Benchmark Analysis of Representative Deep Neural Network Architectures. *IEEE Access* 6 (2018), 64270–64277. https://doi.org/10.1109/ACCESS.2018.2877890

[15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[17] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 367–379.

[18] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[19] R. Collobert, K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

[20] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[21] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 613–627.

[22] C Cuda. 2018. Best practice guide, 2018.

[23] Aditya Dhakal and K. K. Ramakrishnan. 2019. NetML: An NFV Platform with Efficient Support for Machine Learning Applications. In *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 396–404.

[24] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*. ACM, 275–287.

[25] Younghwan Go, Muhammad Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: revitalizing GPU as packet processing accelerator. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 83–96.

[26] Allison Gray, Chris Gottbrath, Ryan Olson, and Shashank Prasanna. 2017. Deploying deep neural networks with nvidia tensorrt. https://devblogs.nvidia.com/deploying-deep-learning-nvidia-tensorrt/.

[27] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, Vol. 40. ACM, 195–206.

[28] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[30] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.".

[31] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[32] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 0.5 mb model size. *arXiv preprint arXiv:1602.07360* (2016).

[33] IBM Corporation. 2018. PowerAI Vision Inference Server. https://www.ibm.com/support/knowledgecenter/SSRU69_1.1.2/base/vision_pdf.pdf?view=kc. Accessed:2019-12-01.

[34] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2018. Dynamic Space-Time Scheduling for GPU Inference. *arXiv preprint arXiv:1901.00041* (2018).

[35] Keon Jang, Sangjin Han, Seungyeop Han, Sue B Moon, and KyoungSoo Park. 2011. SSLShader: Cheap SSL Acceleration with Commodity Processors.. In *NSDI*.

[36] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 1–12.

[37] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. 2015. NBA (network balancing act): a high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 22.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[39] Jason Li, Vitaly Lavrukhin, Boris Ginsburg, Ryan Leary, Oleksii Kuchaiev, Jonathan M Cohen, Huyen Nguyen, and Ravi Teja Gadde. 2019. Jasper: An End-to-End Convolutional Neural Acoustic Model.

arXiv preprint arXiv:1904.03288 (2019).

[40] Peng Liu, Bozhao Qi, and Suman Banerjee. 2018. Edgeeye: An edge service framework for real-time intelligent video analytics. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*. ACM, 1–6.

[41] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient {GPU} Cluster Scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 289–304.

[42] Mellanox Inc. 2018. AI composabilitity and Virtualization: Mellanox Network attached GPUs. http://www.mellanox.com/related-docs/solutions/SB_ai_composability_virtualization.pdf. [online].

[43] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 1–15.

[44] NVIDIA. 2019. TensorRT Developer Guide. https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html. [online].

[45] NVIDIA, Tesla. 2017. V100 GPU architecture. The world's most advanced data center GPU. Version WP-08608-001_v1. 1. *NVIDIA. Aug* (2017), 108.

[46] NVIDIA, Tesla. 2019. MULTI-PROCESS SERVICE. (2019).

[47] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative preemption for multitasking on a shared GPU. *ACM SIGPLAN Notices* 50, 4 (2015), 593–606.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8024–8035.

[49] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[50] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[51] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2135–2135.

[52] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.

[53] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[54] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66.

[55] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. 2014. {GASPP}: A GPU-Accelerated Stateful Packet Processing Framework. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 321–332.

[56] Wikipedia Article. 2018. Diminishing returns. https://en.wikipedia.org/wiki/Diminishing_returns. [online].

[57] Piotr Wojciechowski, Purnendu Mukherjee, and Siddharth Sharma. 2018. How to Speed Up Deep Learning Inference Using TensorRT. https://devblogs.nvidia.com/speed-up-inference-tensorrt/.

[58] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[59] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 595–610.

[60] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.). Vol. 2. 98–111. https://proceedings.mlsys.org/paper/2020/file/f7177163c833dff4b38fc8d2872f1ec6-Paper.pdf

[61] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-NET: Effective {GPU} Sharing in {NFV} Systems. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 187–200.

[62] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.