ECML: Improving Efficiency of Machine Learning in Edge Clouds

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

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

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

2020 IEEE 9th International Conference on Cloud Networking (CloudNet) | 978-1-7281-9486-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/CLOUDNET51028.2020.9335804

***Abstract*—Edge cloud data centers (Edge) are deployed to provide responsive services to the end-users. Edge can host more powerful CPUs and DNN accelerators such as GPUs and may be used for offloading tasks from end-user devices that require more significant compute capabilities. But Edge resources may also be limited and must be shared across multiple applications that process requests concurrently from several clients. However, multiplexing GPUs across applications is challenging. With edge cloud servers needing to process a lot of streaming and the advent of multi-GPU systems, getting that data from the network to the GPU can be a bottleneck, limiting the amount of work the GPU cluster can do. The lack of prompt notification of job completion from the GPU can also result in poor GPU utilization.**

We build on our recent work on controlled spatial sharing of a single GPU to expand to support multi-GPU systems and propose a framework that addresses these challenges. Unlike the state-of-the-art uncontrolled spatial sharing currently available with systems such as CUDA-MPS, our controlled spatial sharing approach uses each of the GPU in the cluster efficiently by removing interference between applications, resulting in much better, predictable, inference latency We also use each of the cluster GPU's DMA engines to offload data transfers to the GPU complex, thereby preventing the CPU from being the bottleneck. Finally, our framework uses the CUDA event library to give timely, low overhead GPU notifications. Our evaluations show we can achieve low DNN inference latency and improve DNN inference throughput by at least a factor of 2.

***Index Terms*—Machine Learning, Inference, Edge Clouds, General Purpose Graphics Processing Unit (GPGPU)**

I. Introduction

A large number of emerging applications, such as speech recognition (e.g., Amazon Alexa, Apple Siri), image recognition, vehicular safety, augmented reality/virtual reality (AR/VR), *etc.* need low latency to satisfy the user's quality of experience (QoE) requirements. Frequently, these core services also require accurate Inference and Machine learning (I&ML) capabilities. These, I&ML applications often use compute intensive Deep Neural Networks (DNNs) which require accelerators such as GPUs for low latency inference and learning. These services typically depend on cloud resources to offload compute intensive tasks. Locating the cloud facilities close to users is highly desirable to ensure low latency and to guarantee QoE. In this regard, the Edge Cloud (such as at the end of the first hop link from the user, whether wired or wireless) or the Edge are often the most suitable to provide 'cloud' services or act as an intermediary to more centralized cloud services. Thus, the usage of Edge for I&ML is also becoming more attractive. However, unlike cloud environments that seek to provide 'almost infinite scalability' of resources, edge resources are likely to be constrained. Hence, the edge needs to judiciously utilize resources by allowing multiplexing of multiple services.

978-1-7281-9486-8/20/$31.00 ©2020 IEEE

Even in an Edge that includes systems with multiple GPUs (e.g., NVIDIA's Tesla T4 system), it is important to utilize the GPUs efficiently and maximize throughput. Multiplexing several applications on the GPU can help support a large demand from I&ML applications. However, multiplexing within the GPU is non-trivial and can be a major challenge, especially to achieve low latency. Traditional approaches, such as temporal multiplexing time-share the GPU among different tasks, but in each time-slice they dedicate a complete GPU (all of the compute engines or the Streaming Multiprocessors (SMs)) to a single DNN task. Tasks with a lighter compute requirement can result in under-utilization of the GPU resources. Temporal multiplexing also has the consequence of increasing inference delays. As an alternative, spatial multiplexing shares a portion of GPU and run different tasks on a GPU simultaneously. Controlled spatial multiplexing (CSM) that carefully manages this sharing is able to control latency for inference and improve throughput [1], [2]. Here, we further complement CSM on a single GPU with techniques to scale to larger number of I&ML applications running on an Edge with the use of a cluster of GPUs as found in multi-GPU systems.

Managing such a multi-GPU cluster for handling I&ML tasks at the Edge poses several challenges. Since an Edge may host multiple I&ML applications concurrently, a lot of inference and learning data would be streaming to the Edge. GPU runtime environments, such as CUDA and OpenCL, require the CPU to perform the major task of extracting the streamed data from the network and transferring to the GPU *i.e.,* the CPU has to copy the payload from network packets to a contiguous buffer and push the data to the GPU using runtime APIs. This is especially concerning as we deploy multi-GPU systems, where the system includes a multi-core CPU and several, up to 8, GPUs. This can cause the CPU to become the bottleneck while performing the data movement task. We seek to alleviate this overhead in this work, building on our previous work on NetML [3]. We use NetML's cut-through approach to transfer data to the GPU using the GPU's DMA engines. This alleviates the overhead on the CPU as it transfers data from network packets to the GPU using the GPU's DMA engine and its scatter-gather capability.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on October 17,2021 at 16:51:31 UTC from IEEE Xplore. Restrictions apply.

Moreover, scheduling of the I&ML tasks across different GPUs, and load-balancing them among the cluster of GPUs, is critical to ensure timely completion of the jobs and proper utilization of the Edge resources. For this, the key is to monitor the task completion time and ensure that the application can be scaled appropriately to meet the desired latency requirements. Further, the GPU being a different subsystem, does not have an effective task completion notification mechanism. Callback methods in current GPU runtimes are limited in functionality and stall the execution in the GPU until the callback is processed. This can cause delay, and idle the GPU resources. We develop a method, exploiting a lightweight CUDA event API, to know immediately that the execution of certain application has completed. The use of the event API avoids the issue of stalling the GPU execution, and prevents GPU from being idle.

In this paper, we first present the challenges associated with the existing GPU multiplexing modes. We then make the case for a framework that can manage multiple-GPUs and provide controlled spatial sharing of the GPUs with proper resource allocation, so as to provide DNN applications with the right amount of GPU resources to meet their low latency requirement. Our preliminary framework provides the foundation to support more complex scheduling, load balancing and auto-scaling mechanisms in the near future. Our framework can efficiently utilize multiple GPUs of the cluster. At the time of initiation of the I&ML function (IF), we check the number of GPUs in the system and choose an available GPU to load the DNN model, create the GPU contexts, and the necessary memory buffers.

Overall, our Edge inference framework reduces the CPU overhead of transferring high volume of streaming data from network to multiple GPU in a GPU cluster and multiplexes applications by spatially sharing GPUs in the cluster. Edge clouds can facilitate low latency ML inference. However, due to resource constraints, they do not offer the same economies of scale that a centralized cloud may provide. Nonetheless, the proposed techniques and optimizations of our framework can also improve the utilization of centralized cloud. Multiplexing applications across, and within individual resources can have much larger impact in the utility and economics of relatively resource-scarce edge clouds. Multiplexing within resources might not be as critical in a large centralized cloud due to its inherent economies of scale. We demonstrate that our approach helps with getting overall higher throughput than providing single GPU to each DNN. Moreover, our approach keeps latency low and predictable.

II. Background and Related Work

A. NVIDIA CUDA Support for GPU Multiplexing

NVIDIA GPUs along with the CUDA runtime provide support for sharing the GPU, both temporally and spatially. The state-of-the-art CUDA MPS (multi-process service) [4] allows to spatially share the GPU across multiple, different processes. However, when GPU resources are insufficient, multiple application contend with each other for the same GPU resources, often resulting in interference with each other's kernel's execution. This interference often results in increased and unpredictable latency while performing inference

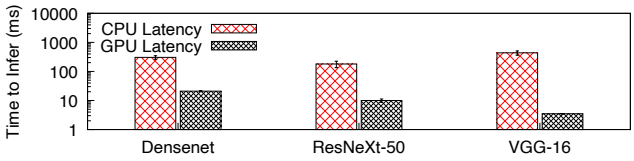


Fig. 1. Latency of inferring one image with CPU vs. a GPU

in conjunction with other inference or machine learning tasks. By default, CUDA MPS does have the means to limit the amount of GPU resources used by an application, but it needs to be managed carefully, which we describe in this paper.

B. Edge Cloud GPU Inference Platforms

Computing at edge clouds promises to provide more responsive services requiring significant compute resources for functions not convenient to be supported on hand-held devices (e.g., smart phones) because of computation, power or cost limitations. Moreover, services often require fusing of multiple sources of data (e.g., sensory inputs) that require more computation, but also low latency [5]. Edge clouds are often used to offload processing from end-devices [6]–[8]. With central offices of traditional communication providers having space, power, and climate-conditioning, they are becoming prime candidates for housing edge cloud services. This is also true for wireless environments, such as cellular, with more distributed deployments of the packet core. While the edge clouds have more processing capacity and storage than a single system, they are still likely to be resource constrained relative to the aggregated demand from edge devices *e.g.,* smart phones, IoT devices, vehicles, *etc*). Hence, it is important to judiciously manage edge cloud resources. Further, the edge workload and traffic characteristics, akin to centralized clouds, can vary drastically over time [9]. This makes it necessary to adapt the I&ML support to better match current workload.

DNNs are compute heavy and performing inference with them in devices with limited computational capability (and limited battery power) such as a smartphone can take several seconds for even a single inference [10]. Offloading the computation to servers with more compute power can drastically bring down the inference time. However, even with additional computational resources of an edge server, the inference would be an order of magnitude slower if the platform only uses CPUs compared to one with even a small GPU [11]. DNNs, specially Convolutional Neural Network (CNNs) are often composed of multiple matrix operations, which can benefit from parallelization offered by the GPU. When compared to CPUs, GPUs with a number of compute engines and cores can accelerate DNN inference significantly (by 2-3 orders of magnitude). We have computed the average latency to infer one image with different models in Pytorch in one CPU core and one NVIDIA V100 GPU. We present our results in Fig. 1. We can see that a GPU can help infer the image $10\times$ faster than using a CPU. Therefore, it is necessary to use GPUs for I&ML workloads with real-time response requirements.

DNN and other ML applications are typically modeled, trained and deployed using one of a number of platforms, such as PyTorch [12], Microsoft CNTK [13], NVIDIA Ten-

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on October 17,2021 at 16:51:31 UTC from IEEE Xplore. Restrictions apply.

TABLE I
DIFFERENT INFERENCE PLATFORMS AND THEIR CHARACTERISTICS.

ML Platform	Programming model(s)	CUDA Streams	Batching	Startup Time (sec) (ResNet-50 model)
CNTK [13]	CUDA	No	Yes	2.2
Darknet [16]	CUDA	No	No	5.7
MxNet [17]	CUDA/OpenCL	Yes	Yes	5.53
PyTorch [12]	CUDA/OpenCL	Yes	Yes	5.03
TensorFlow [15]	CUDA/OpenCL	Yes	Yes	9.2
TensorRT [14]	CUDA	Yes	Yes	3.2

sorRT [14], Tensorflow [15] *etc.* Table I presents the key characteristics of the some of the most popular ML platforms.

All of these platforms support CUDA programming to interface with the GPU and are designed to support NVIDIA GPUs. There are differences in terms of their characteristics and support for different performance enhancing options, *e.g.,* CNTK and Darknet do not support CUDA streams; Darknet does not support batching DNN requests, etc. Another interesting aspect is that these frameworks have different model loading times. We evaluated the time for a ResNet-50 model to load in different platforms in our testbed with NVIDIA V100 GPU. We present our results in Table I. We can see that loading time of a model for the ResNet-50 model is in seconds. A high startup time is a concern when rapid change in demand for a certain application requires additional instances of the DNN model. Our prior work GSLICE [1] facilitates adjustment of ML platforms' GPU resources as well as gracefully dealing with the long start-up delays while adjusting resources.

Many ML platforms now offer a inference/learning serving interface that can be deployed in the cloud. Tensorflow serving [18], Torchserve [19] can host models in the cloud for inference. However, these cloud based platforms generally use the default CUDA runtime and are not suitable for edge cloud with scares resources. There is work for improving edge based systems [20], [21], focusing on improving the DNN algorithms to achieve a better balance between accuracy and inference latency. Our work is complementary to these efforts. We seek to maximize CPU and GPU utilization and lower latency, by improving the I&ML system instead.

III. SYSTEM DESIGN

We propose ECML, an I&ML framework for Edge clouds, as shown in Fig. 2. The framework consists of applications running on the OpenNetVM (ONVM) environment [22] which aids in low latency high throughput packet processing. Each I&ML application can load its DNN model into any GPU in the cluster and specify the GPU% such that the GPU can be shared with another I&ML's model. Each I&ML application in the environment will have access to shared memory where packets are forwarded by the NIC and can use NetML [3] to transfer application data, *e.g.,* images, to the GPU for inference.

The ECML application environment consists of three main modules: 1) An enhanced NetML module that helps move the data from packet payloads to and from GPU memory, utilizing the GPU's DMA engine(s); 2) The host CPU ML module that hosts ML platform libraries such as PyTorch, TensorRT, and Tensorflow, and is able to run DNNs developed for such platforms. 3) A Streams & Notification module that helps the inference task running in the GPU and monitor the GPU, such that the CPU-resident application is promptly informed when the inference task completes.

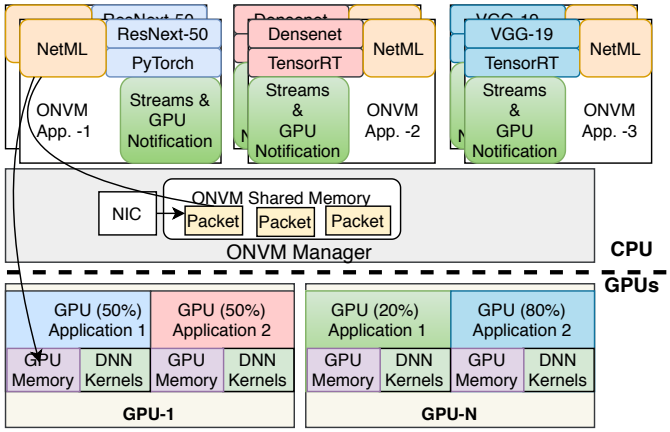


Fig. 2. ECML framework for edge cloud

A. NetML: Data Transfer to GPU

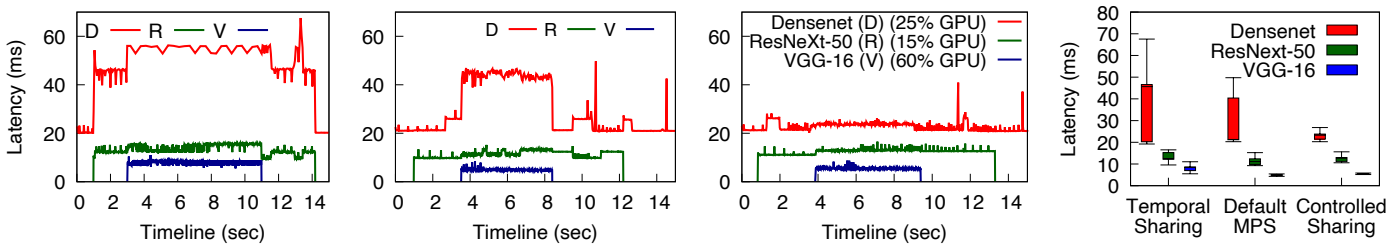
Unlike a standalone node, edge I&ML platforms with multiple GPUs and hosting multiple I&ML applications will need to handle a large amount of streaming data. A single GPU might easily get fully utilized by running 2 or 3 different I&ML applications. However, with a cluster of 8-16 GPUs in a server, a lot more I&ML applications can be run concurrently, therefore, it involves moving a lot more incoming data to the GPU to perform inference. We can see from Fig. 2 that the streaming data arrives from the NIC and is buffered in ONVM's shared memory. Copying the payload from each packet to a contiguous buffer and then to the GPU is expensive in terms of CPU cycles and adds latency.

We developed NetML [3] to avoid the extra copy by running a GPU kernel that uses the GPU's DMA engine to perform scatter-gather from host memory. This approach provides two key benefits: i) it is efficient and timely; ii) more importantly, it saves host CPU cycles. The original NetML design only infers a single image and start transferring data from packet payload to GPU as soon as first packet with image data arrives. However, to get the throughput improvement that can be gained from batching, we now modify the original NetML design to support batching. In NetML, if we miss a portion of data for an image due to network loss, we would have to wait for it or choose to not infer the image depending on user settings. However, to infer a batch of images, all of the batch's image data has to be present in contiguous memory buffer in the GPU. If we start building the batch in GPU from the first received data packet and subsequently miss some data for images in middle of the batch due to network loss, we would have to recreate a batch in GPU only with images that have all their data available, by copying data within GPU. To avoid this, we design a lightweight verification mechanism that verifies that the data for an entire image is present in received packets before transferring data to the GPU.

B. GPU Multiplexing

We describe 3 different methods of multiplexing I&ML applications in a GPU.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on October 17,2021 at 16:51:31 UTC from IEEE Xplore. Restrictions apply.



(a) Temporal Sharing (b) Default MPS (c) Fixed GPU%. (d) GPU sharing latency variation

Fig. 3. Latency of DNN models multiplexing GPU (a) temporally, (b) default MPS and (c) spatially with fixed GPU%.(d) box plot of inference latency

1) Temporal Sharing: Both CUDA and OpenCL support scheduling GPU kernels from different processes with regular time quanta, to temporally share the GPU. But, GPUs may be underutilized, as a single process usually fails to fully utilize all the GPU resources. We run an experiment with three different models, DenseNet, ResNeXt-50 and VGG-16 (start and finish in that order) concurrently on the PyTorch platform, temporally sharing the GPU. We see from Fig. 3a that initially the latency of Densenet is low, $\sim$ 20 ms per inference. With contention from the ResNeXt model, the latency of Densenet increases more than 2$\times$. VGG-16 then further increases the latency of both the DenseNet and ResNeXt models.

2) Spatial Multiplexing of the GPU: MPS allows spatially sharing of the GPU among multiple processes. We ran a similar experiment with default MPS (*i.e.,* spatial sharing of GPU). Fig. 3b shows that the latency of all three models are lower compared to temporal sharing. This is because MPS allows applications to use the spare GPU resources, if another concurrently running model is not fully utilizing the GPU. When ResNeXt comes up, the latency of Densenet does not increase as there are enough GPU resources to allow both to execute concurrently. But, when the third, VGG-16, model comes up, the latency of Densenet increases significantly, while ResNeXt's latency also increases by a small amount. But more importantly, the inference latency for each of the models is highly variable. While default MPS may improve GPU utilization, it leads to unpredictable latency for the concurrently running models.

3) MPS with resource usage limits: The recent NVIDIA CUDA architectures (i.e., Volta) provides resource provisioning limits in MPS, setting a fixed maximum proportion (GPU%) of the available threads (in units of # SMs) for individual clients. We leverage our work in [1] in ECML to provide controlled spatial sharing of the GPU. Our approach achieves good performance isolation through proper GPU resource separation among the different GPU applications.

To demonstrate the resource isolation and essentially eliminating interference, we run the 3 DNN models concurrently in the GPU with fixed GPU% for each. For this experiment, we provided Densenet with 25% GPU, ResNeXt-50 with 15% GPU and VGG-16 with 60% GPU. We see from Fig. 3c and Fig. 3d that the latency of the models remain about same for the entire experiment, with low variance (see boxplot), indicating almost no interference between concurrently running models. Since the latency of Densenet is the same throughout, we conclude that only 25% GPU is necessary for Densenet to run and still achieve the lowest latency across the different multiplexing options.

C. Inference deadline and adaptive batching

Batching multiple requests together helps amortize the cost of launching DNN functions and transferring data to GPU, thus, increasing inference throughput. However, larger batch sizes also increase the inference latency as bigger batch takes a longer time time to form, as well as take longer to complete inference in the GPU. Many tasks in the edge cloud can be latency sensitive and come with a deadline, or a service level objective (SLO). We have used self-learning adaptive batching (SLAB) as described in GSLICE [1] to benefit from both higher throughput from higher batch size, as well as to complete inference before the deadline. SLAB increases the batch size till total inference latency does not exceed the deadline.

D. Inference using multiple GPUs: Notification and Scheduling

Utilizing multiple GPUs in an I&ML inference platform requires coordination between applications and the GPUs to perform inference efficiently. Depending on demand, a DNN may be running in a single GPU or can have multiple instances within the GPU cluster. Moreover, with spatial sharing within a GPU, multiple different DNNs may be concurrently using the GPU. Load balancing across multiple instances of the same DNN service and coordinating between multiple services within a GPU are necessary.

Managing multiple inference tasks on the CPU & GPU is especially challenging as the GPU computation is asynchronous with respect to the CPU. The asynchronous execution helps to free up the CPU after it submits the job to GPU and allows a single CPU thread to submit multiple tasks to the GPU without waiting for the tasks to be completed. However, there is no simple way for the CPU task to know when a submitted task has completed on the GPU. The CUDA API offers a 'callback' function *cudaLaunchHostFunction()* to notify the end of processing in the GPU by running a callback function on the CPU. Although the callback provides a notification of the end of a task, the current implementation of the callback does not help achieve effective GPU multiplexing. First, multiple concurrent callbacks are serialized, which halts the subsequent execution on the GPUs until the completion of the execution of the callback routine on the CPU. This results in the GPU idling. Second, the callback functions generated by the GPU prevent the CPU thread from running any CUDA API functions. Thus, this limitation demands an additional CPU context and signalling scheme to perform GPU related operation.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on October 17,2021 at 16:51:31 UTC from IEEE Xplore. Restrictions apply.

To overcome these limitations, we devise a lightweight method to check the GPU task completion status. We leverage CUDA's event-based API to record an 'event' (can be per GPU stream) when the DNN is executed. The CUDA event acts as a flag. When it is executed by the GPU, it records the time of the execution. The CUDA API function *cudaEventRecord()* allows us to put an event marker at the end of the DNN's execution. We then use another API function *cudaEventQuery()* to check if the event that we placed has completed execution or not.

Together with the lightweight notification scheme, we utilize a lightweight data structure to keep track of all active I&ML applications and utilization information of each GPU. With every task completion notification for an I&ML application, we compute the throughput and latency for inference. With the completion of every inference, we also change the status of I&ML application to becoming available for another inference. In contrast to running inference in a single GPU, where, a particular DNN, *e.g.,* ResNet-50, might only be served by 1 I&ML application, a multiple GPU cluster can host multiple ResNet-50 services. Therefore, it is necessary to maintain the availability state of each I&ML application.

With the throughput and latency statistics and the information about the arrival rate of DNN requests, we can autoscale, *i.e.,* start a new DNN instance, or scale down the DNN instance based on the request arrival rate. These statistics help us increase or decrease GPU resources for a particular I&ML application and load balance among multiple existing I&ML applications. Similarly, these statistics help design a scheduler for I&ML applications based on the demand. Enhancing our ECML framework with detailed auto-scaling, load-balancing and scheduler algorithms is part of our future work.

IV. Evaluation

A. Evaluation Testbed

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

We use PyTorch and TensorRT platform for our evaluations. Our DNN workload consists of color images of resolution 224×224. For the experiments using NetML and data transfer we transmit each image through network as 588 UDP packets where each packet has a payload of 1 kilobytes. We use Moongen and TCPreplay as traffic generator for transmitting images. With 10 GbE connection between traffic generator and the receiving server, we can transmit 1920 images per second. For all our experiments we only report the execution time of inference and eliminate the additional network-related latency contributed by network protocols, including HTTP, TCP or UDP.

B. Data Transfer To GPU

We measured the CPU utilization of CPU-Copy and NetML by measuring percentage of CPU cycles spent in transferring image data of different batch sizes to GPU out of all the CPU cycles used in one inference task. We present the utilization of CPU in Table. II. We observe that, the amount of CPU cycles as well as proportion of CPU cycles increases for CPU-Copy as the batch size increases. This is because the operations other than data copying i.e., calling GPU kernels for inference, CUDA API calls, etc. are amortized across a batch of images. But the data copy overhead itself sees no such amortization. But, NetML barely uses any CPU cycles for data transfer, with the maximum being 0.21% used to transfer batch of 32 images as NetML offloads the data transfer to GPU's DMA engine.

TABLE II
CPU Usage in different modes for data transfer to GPU

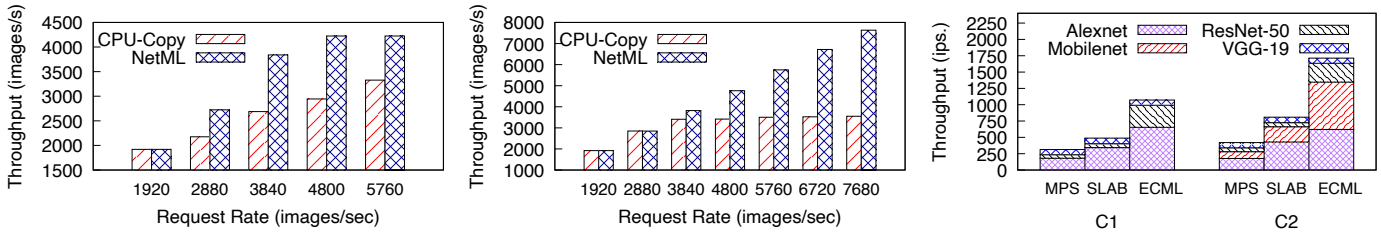
Batch Size	Data Size (MB)	CPU-Copy (%)	NetML (%)
1	0.57	10.6	0.12
4	2.29	29.2	0.15
8	4.59	35.99	0.16
16	9.19	43.8	0.20
32	18.37	45.73	0.21

We evaluate NetML using (i) single V100 GPU and (ii) cluster of 8 Tesla T4 GPUs. We use a fixed batch size of 8 for inferring incoming requests in both experiments. In Fig. 4a we show the throughput achieved by Alexnet in a single V100 GPU with increasing requests. Inference with NetML provides higher throughput than CPU-Copy as the request rate exceeds 2000 images per second. We also see the inference with NetML peaks at about 4000 images per second. This is when the GPU utilization is maximized and becomes the bottleneck. Inference with CPU-Copy remains lower, as the CPU becomes the bottleneck due to the data transfer overhead. In Fig. 4b we evaluate the inference with NetML and CPU-Copy in a 8 GPU cluster. Here, we see that with NetML we can infer virtually all requests coming in from 40GbE bandwidth (7680 images per second). However, CPU-Copy hits the peak inference at about 3500 images per second due to CPU data transfer being the bottleneck.

C. Inference in Multiple GPU

Figure 4c (left) shows the performance running 3 different type of IFs (Alexnet, ResNet-50 and VGG-19) in each of the 4 GPUs concurrently (total of 12 IF instances). Fig. 4c (right) shows the performance of running 4 different type of IFs (Alexnet, Mobilenet, ResNet-50 and VGG-19) concurrently in each of 4 GPUs (16 IF instances). We set the SLO to 50 ms. We compare the throughput of MPS+SLAB and ECML for both setups. We should note that SLAB provides batching but no NetML, while ECML provides both. For the aggregate throughput of all models, ECML provides more than 2× that of MPS+SLAB. ECML eliminates the interference caused by the model with high computational requirement (VGG-19), thus allowing lighter models (computationally) to achieve higher throughput. However, ECML only marginally reduces the throughput of the heavy model (for VGG-19, 84 inferences/sec with MPS+SLAB vs. 79 with ECML). We recognize that a cloud provider may choose to have a GPU cluster to provide additional spatial multiplexing across GPUs, given the limitations of CUDA-MPS. But, with ECML, multiple DNN models can run concurrently on each GPU in a multiple GPU system, thus further improving multiplexing capability.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on October 17,2021 at 16:51:31 UTC from IEEE Xplore. Restrictions apply.



(a) Inference throughput with Single GPU (b) Inference throughput of Alexnet with 8 GPUs (c) Throughput with 8 GPU cluster, C1 = Alexnet, ResNet-50 and VGG-19. C2 = C1 + Mobilenet

Fig. 4. (a)(b) Inference throughput of Alexnet (TensorRT) with CPU-Copy and NetML. (c) Throughput of ECML and alternatives in 8 GPU cluster

TABLE III
LATENCY AND THPT. OF DNNS WITH ENTIRE GPU VS. SHARING GPUS

	Mobilenet		ResNet-50	
	Throughput	Latency	Throughput	Latency
1 DNN per GPU	1720 (images/s)	18.3 ms	480	66
DNNs Sharing GPU	1720	18.3	680	45

D. Benefits of Multiplexing models across multiple GPU

We evaluated scenarios where each individual model has an entire GPU, versus one where GPU is shared between models. We shared 2 Tesla T4 GPUs with Mobilenet and ResNet-50, where Mobilenet DNN gets 35%, and ResNet-50 gets 65% of each GPU. The request rate is 1920 images/sec and the maximum batch size is 32. We do not set an SLO to allow unconstrained batch formation. Average throughput shown in Table III. Controlled spatial sharing of the GPU increases ResNet-50 throughput by 40% and reduces inference latency, while keeping throughput for Mobilenet same as with dedicated GPU. In essence, ResNet-50 gets 130% of a GPU's capacity by sharing, while Mobilenet uses 70% GPU across 2 GPUs, and gets same throughput as dedicating a GPU for it.

V. SUMMARY AND FUTURE WORK

With the growing need for ML for latency-sensitive applications, executing them at an edge cloud is desirable. Because of limited compute capacity at an edge cloud (compared to centralized clouds), we need to better utilize CPU and GPU resources. To more effectively use multi-GPU machines, we seek to better utilize each of the GPUs by controlled spatial multiplexing of the GPU. Compared to pure temporal sharing or even the uncontrolled spatial sharing offered by CUDA MPS, controlled spatial multiplexing gives much lower, predictable, inference latency and higher inference throughput. We also need a timely notification of GPU task completions to the CPU to effectively use the GPUs and reduce latency. Having a small kernel thread in the GPU subsystem to fully take advantage of the GPU-resident DMA (as in NetML) substantially improves movement of streaming data to the GPU, relieving CPU load, a critical factor in multi-GPU systems to improve inference throughput. Our current work is to develop a framework to support a richer scheduling, load-balancing auto-scaling algorithms to support a variety of ML frameworks with limited or no modifications to their existing algorithms in multi-GPU systems.

VI. ACKNOWLEDGEMENT

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

REFERENCES

[1] A. Dhakal *et al.*, "Gslice: controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 492–506.
[2] A. Dhakal *et al.*, "Machine learning at the edge: Efficient utilization of limited cpu/gpu resources by multiplexing," in *AI towards Mission-Critical Communications and Computing at the Edge (AIMCOM2) workshop at ICNP 2020*, 2020.
[3] A. Dhakal and K. K. Ramakrishnan, "Netml: An nfv platform with efficient support for machine learning applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019.
[4] NVIDIA, Tesla, "Multi-process service," *NVIDIA. May*, p. 108, 2019.
[5] M. Satyanarayanan, "Edge computing for situational awareness," in *Local and Metropolitan Area Networks (LANMAN), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1–6.
[6] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.
[7] L. Liu *et al.*, "Edge assisted real-time object detection for mobile augmented reality," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16.
[8] X. Ran *et al.*, "Deepdecision: A mobile deep learning framework for edge video analytics," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1421–1429.
[9] H. Chang *et al.*, "Bringing the cloud to the edge," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2014, pp. 346–351.
[10] A. Ignatov *et al.*, "Ai benchmark: Running deep neural networks on android smartphones," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 0–0.
[11] X. Zhang *et al.*, "pcamp: Performance comparison of machine learning packages on the edges," in {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.
[12] R. Collobert *et al.*, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.
[13] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016.
[14] NVIDIA, "Tensorrt developer guide," https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html, 2019, [ONLINE].
[15] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.
[16] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.
[17] T. Chen *et al.*, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*.
[18] "Tensorflow serving," https://www.tensorflow.org/tfx/guide/serving.
[19] "Torchserve," https://pytorch.org/serve, 2020.
[20] C.-J. Wu *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.
[21] S. Wang *et al.*, "High-throughput cnn inference on embedded arm big.little multi-core processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
[22] W. Zhang *et al.*, "OpenNetVM: A Platform for High Performance Network Service Chains," in *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, Aug. 2016.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on October 17,2021 at 16:51:31 UTC from IEEE Xplore. Restrictions apply.