Machine Learning at the Edge: Efficient Utilization of Limited CPU/GPU Resources by Multiplexing

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

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

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

***Abstract*—Edge clouds can provide very responsive services for end-user devices that require more significant compute capabilities than they have. But edge cloud resources such as CPUs and accelerators such as GPUs are limited and must be shared across multiple concurrently running clients. However, multiplexing GPUs across applications is challenging. Further, edge servers are likely to require considerable amounts of streaming data to be processed. Getting that data from the network stream to the GPU can be a bottleneck, limiting the amount of work GPUs do. Finally, the lack of prompt notification of job completion from GPU also results in ineffective GPU utilization. We propose a framework that addresses these challenges in the following manner. We utilize spatial sharing of GPUs to multiplex the GPU more efficiently. While spatial sharing of GPU can increase GPU utilization, the uncontrolled spatial sharing currently available with state-of-the-art systems such as CUDA-MPS can cause interference between applications, resulting in unpredictable latency. Our framework utilizes controlled spatial sharing of GPU, which limits the interference across applications. Our framework uses the GPU DMA engine to offload data transfer to GPU, therefore preventing CPU from being bottleneck while transferring data from the network to GPU. Our framework uses the CUDA event library to have timely, low overhead GPU notifications. Preliminary experiments show that we can achieve low DNN inference latency and improve DNN inference throughput by a factor of ~ 1.4.**

***Index Terms*—GPU, Machine Learning, Deep Neural Networks, Inference**

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 GPU 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) are often the most suitable to provide 'cloud' services or act as an intermediary to more centralized cloud services. Thus, the usage of edge resources for I&ML is also becoming more attractive.

978-1-7281-6992-7/20/$31.00 ©2020 IEEE

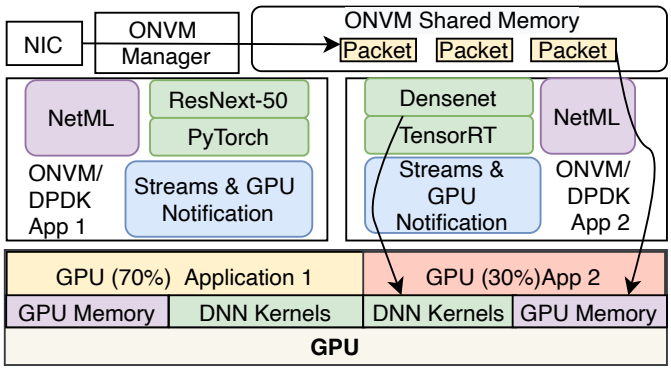


Fig. 1. I&ML framework for edge cloud

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.

Multiplexing several applications on GPU is critical to both support large demand of I&ML applications and to improve utilization of Edge cloud. However, GPU multiplexing is non-trivial and is a major challenge. First, the GPUs have only supported time-shared (temporal) multiplexing *i.e.,* each task is assigned a specific quantum such that multiple tasks appear to execute concurrently on a GPU. As a task with lighter compute requirement is given same amount of GPU time as heavier task. This results in under-utilization of the GPU resources, and increase in latency due to time-sharing between multiple applications. Second, as edge cloud hosts multiple I&ML applications concurrently, lot of inference and learning data would be streaming to the edge. GPU runtimes such as CUDA and OpenCL also requires CPU to perform the heavy lifting of transferring data from network to GPU. CPU has to copy the data from network packets to a contiguous buffer and push the data to GPU using runtime APIs causing CPU to become bottleneck in the process. Third, GPU being a different subsystem, does not have effective task completion notification. Callback methods present in current GPU runtime are limited in functionality and stall execution in the GPU until callback is processed. This can cause unnecessary delay and GPU idle time.

In this paper, we consider above challenge and describe and evaluate different GPU multiplexing options. We then make the case for controlled spatial sharing of the GPU with proper resource allocation, so as to provide the DNN applications

with the right amount of GPU resources to meet their low latency requirement. We use NetML [1], a cut-through approach, to transfer data to the GPU using the GPU's DMA engines. This alleviates the overhead on the CPU and directly transfers data from network packets to the GPU using the GPU's DMA engine. Finally, we use a lightweight CUDA event API to know when the execution of certain application have ended. Use of event API avoids the issue of stalling GPU execution thus, prevents GPU from being idle.

There are two ways spatial sharing is used in NVIDIA GPUs. First being 'CUDA Streams' a software abstraction that represents a sequence of commands (execution kernels (functions running in GPU), data copies, and other commands), that can be launched and executed in order within a process. Streams enable spatial sharing and allow for coarse-grained parallelism. However, when GPU resources are insufficient, multiple streams contend with each other for the same GPU resources, often resulting in interference between the different kernel's execution.

The state-of-the-art CUDA MPS (multi-process service) [2] is another method to spatially share the GPU. MPS goes beyond the constraints of Streams by allowing spatial multiplexing of the GPU across multiple, different processes. However, in a manner similar to Streams, MPS also can engender interference between these applications, resulting in increased, unpredictable latency. CUDA MPS does have the means to limit the amount of GPU resources used by an application. This is achieved by defining a GPU% while starting a GPU application. This limits the application to using at most the amount of GPU resources (Streaming Multiprocessors (SMs)) specified by the GPU%. We take advantage of this scheme to provide controlled spatial sharing of the GPU in a manner that not only increases the GPU utilization, but also avoids the interference. Our approach achieves good performance isolation through proper GPU resource separation among the different GPU applications.

Unlike a standalone node, the edge I&ML platforms also need to handle large amounts of streaming data. This requires the data from the network (e.g., 40/100 Gbits/sec. Ethernet links) to be transferred and assembled at the CPU and then again that assembled data be transferred to the GPU to perform inference and/or ML training operations. This approach incurs significant amount of CPU cycles to process the streaming data and additionally demands judicious co-ordination with the GPU to perform the inference operation. We show that NetML is very useful in scenarios where a huge quantity of streaming data is being inferred upon in the edge server. Using the CPU to copy and arrange the data will unnecessarily increase latency and decrease the throughput of inference.

We propose an I&ML framework for an edge as seen in the Fig. 1. The framework consists of applications running in the OpenNetVM environment [3] which aids in low latency high throughput packet processing. Each I&ML application in the environment will have access to shared memory where packets are forwarded by the NIC. We further present more efficient ways of CPU-GPU interaction that efficiently utilize the CPU resources by avoiding unnecessary wastage of the CPU for data transfer, and idling of the CPU and GPU due to the blocking of the GPU execution during the callback notification operation.

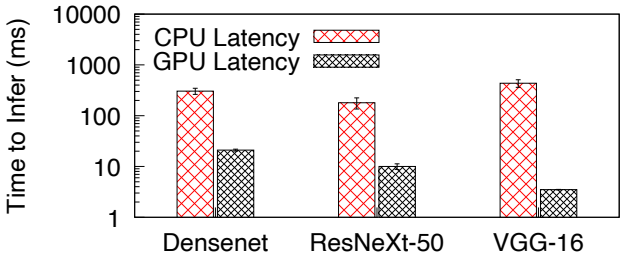


Fig. 2. Latency of inferring one image with CPU vs. a GPU

The I&ML application environment consists of three main modules: 1) A NetML module that helps transmits the data from packet payload to GPU memory with help of GPU DMA engine. 2) The ML platform module hosts ML platform libraries such as PyTorch, TensorRT and Tensorflow and is able to run the DNN written for such platforms. 3) A Streams & Notification module that helps infer the task in GPU as well as monitor the GPU such that the application knows when a task has ended.

II. Background and Related Work

A. Edge/Cloud GPU Inference Platforms

Edge Computing 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 [4]. Edge clouds are often used to offload processing from end-devices [5]–[7]. 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 [8]. This makes it necessary to adapt the I&ML support to better match current workload.

Inference and training of DNN models are suitable candidates for offloading to the edge. 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 [9]. 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 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

TABLE I
DIFFERENT INFERENCE PLATFORMS AND THEIR CHARACTERISTICS.

ML Platform	Programming model(s)	CUDA Streams	Batching	Startup Time (sec) (ResNet-50 model)
CNTK [11]	CUDA	No	Yes	2.2
Darknet [14]	CUDA	No	No	5.7
MxNet [15]	CUDA/OpenCL	Yes	Yes	5.53
PyTorch [10]	CUDA/OpenCL	Yes	Yes	5.03
TensorFlow [13]	CUDA/OpenCL	Yes	Yes	9.2
TensorRT [12]	CUDA	Yes	Yes	3.2

in Fig. 2. 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 [10], Microsoft CNTK [11], NVIDIA TensorRT [12], Tensorflow [13] *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. Some frameworks, such as TensorRT load the DNN model from disk to GPU directly and offer no CPU side inference, while, PyTorch, TensorFlow, CNTK and MxNet load the model in host memory first and only transfer the model data to the GPU when the first inference/training instance is executed. Similarly, the GPU memory requirement for different frameworks also vary. TensorRT has a static memory occupancy in GPU, while TensorFlow and PyTorch takes a portion of GPU memory and uses a custom allocator to provide GPU memory to the ML programs running in Tensorflow.

Actions taken by ML platforms while setting up for training or inference such as memory reservation, GPU initialization and DNN/ML application/data loading in the GPU contribute to the high startup time for getting a model to be ready for inference or training. We evaluated the time for a model to load in different platforms in our testbed with NVIDIA V100 GPU. We chose ResNet-50 as a representative model to load as it is popular and available for all the different platforms. It is a relatively small model with $\sim$104 megabytes of weights. We present our results in Table I. We can see that loading time of a model for the ResNet-50 model is in seconds. In the platforms that load the model from disk onto CPU and then to the GPU, such as Tensorflow, it is even higher. While the inference latency of ResNet-50 is less than 10 ms for all these platform, a high startup time is a concern when rapid change in demand for a certain application requires additional instances of the DNN model. To facilitate running a variety of I&ML applications, it is necessary for an edge cloud platform to support resource allocation and adjustment of ML platforms 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 [16], Torchserve [17] can host models in the cloud for inference and learning. However, these cloud based platform do not support spatial multiplexing with fixed GPU% to ensure resource isolation. Similarly, there is work for improving edge based systems [18], [19]. However, most of these focus on improving the DNN algorithm and in achieving 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.

III. GPU MULTIPLEXING FOR ML APPLICATIONS

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 and 16 GB of memory.

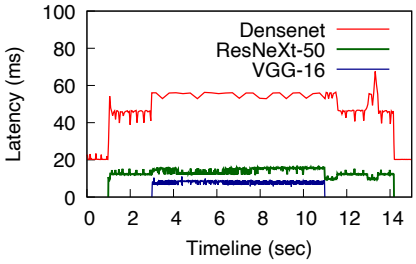
We use PyTorch and TensorRT platform for our evaluations. Our DNN workload consists of color images of resolution 224×224. For the experiments using NetML and data transfer we transmit each image through network as 588 UDP packets where each packet has a payload of 1 kilobytes. We use Moongen and TCPreplay 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 or training and eliminate the additional network-related latency contributed by network protocols, including HTTP, TCP or UDP.

B. Different Modes of GPU Multiplexing

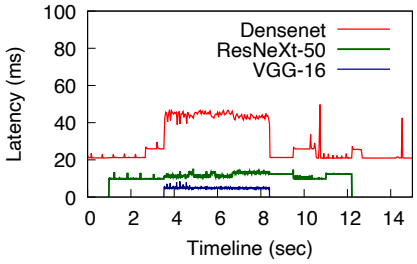
In order to improve GPU utilization and performance, several approaches such as multiplexed processing on GPUs, multiple streams, batched execution *etc.* have been proposed. We briefly describe the key multiplexing approaches and our preliminary evaluation and observations.

1) Temporal GPU Multiplexing: Like multi-programming in Unix/Linux, both CUDA and OpenCL support GPU kernels from different processes to time-share the GPU. However, it does not let them use the GPU concurrently. This can leave the GPUs underutilized, as a single process usually fails to fully utilize all the GPU resources.

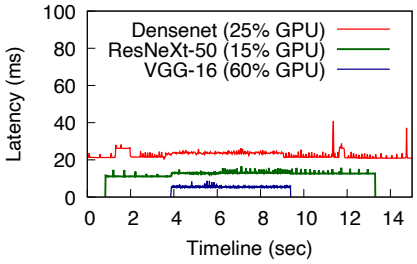
We run an experiment to show the effect on latency of inferring different images by different models which are temporally sharing the GPU. We run three different models, DenseNet, ResNeXt-50 and VGG-16 concurrently using PyTorch platform. We start the DNNs such that Densenet start first and starts inferring images as the ResNeXt-50 and VGG-19 models get loaded into the GPU to begin inference. The DNN models finish their inferences in same order they came up. We can see from Fig. 3a that the latency of Densenet is low, $\sim$ 20 ms per inference, when it is the only model running in the GPU. Once the ResNeXt model starts inferring, the latency of Densenet increases more than $2\times$. Starting VGG-16 further increases the latency of both the DenseNet and ResNeXt models. Completing the execution of VGG-16 and ResNeXt models lowers the per inference latency of DenseNet back down to $\sim$20 ms. Thus, increasing the number of concurrently



(a) Temporal Sharing



(b) Spatial sharing with Default MPS



(c) Spatial sharing of GPU with fixed GPU%.

Fig. 3. Latency of DNN models running concurrently and sharing GPU temporally, with default MPS and spatially with fixed GPU%

running models by temporally sharing the GPU increases latency for all the concurrently running GPU applications.

2) Spatial GPU Multiplexing: CUDA streams and MPS leverage Hyper-Q [20] to provide concurrent multiplexing of GPU kernel tasks within and across multiple processes respectively. We first focus on MPS that allows spatially sharing of the GPU among multiple processes. By default, an MPS client has all of the available threads usable (i.e., 100% of GPU resources). However, this results in GPU resource contention among concurrent kernels from different processes, resulting also in performance variation for the application at different time periods, and unpredictable latency.

We performed a similar experiment as the one above, by concurrently running three different DNN models while sharing the GPU using default MPS. We can see from the Fig. 3b that the latency of all three models are lower compared to temporally sharing the GPU by the DNN models. This is due to the fact that sharing GPU with CUDA MPS allows the applications to share the GPU spatially and use the spare GPU resources, if another concurrently running model is not fully utilizing the resources. From Fig. 3b we can also see that when ResNeXt model comes up the latency of Densenet does not increase as there are enough GPU resources to allow both to concurrently execute. However, when the VGG-16 model comes up, the latency of Densenet increases significantly, while ResNeXt's latency only increased a small amount. We observe that the amount of increase of latency, and which concurrently running model suffer the additional latency is not predictable, and depends on the sequence with which GPU resources are committed to the models. While default MPS may improve GPU utilization, it leads to unpredictable latency for concurrently running models.

3) MPS with resource usage limits : The recent CUDA architecture (i.e., Volta) has a version of MPS which supports resource provisioning limits, i.e., it allows us to limit individual MPS clients to use particular portion of the available threads (in units of # SMs). This enables us to reduce the GPU memory footprint and to achieve performance isolation. We argue and demonstrate that, while this feature is useful, it still requires more careful consideration and resource management to truly reap the benefits and full utilization of the GPU.

To demonstrate the resource isolation and essentially eliminating interference, we run three DNN models concurrently in the GPU with fixed GPU% for each of the model. For this experiment we provided Densenet with 25% GPU, ResNeXt-50 with 15% GPU and VGG-16 with 60% GPU. We can see from Fig. 3c that the latency of the models remain about same for the entire experiment. There is almost no interference between concurrently running models. We further observe that the latency of Densenet model is the same as when it is the only model running in the GPU. We conclude that only 25% GPU is necessary for Densenet to run and still achieve the lowest latency across the different multiplexing options.

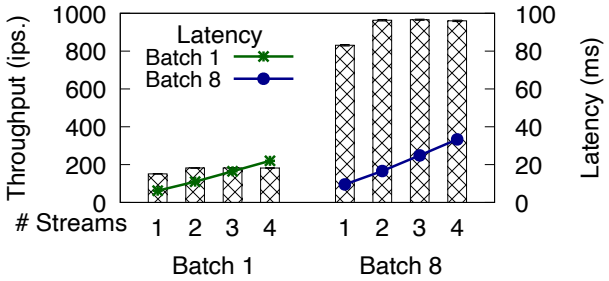


Fig. 4. Throughput & latency for different # of streams using ResNet-50

C. Use of CUDA Streams

Although, streams due to spatial sharing, enable to improve GPU utilization and overall throughput, they have adverse impact on latency. We experimented with different models for different batch sizes to see the impact of using multiple streams. We present the results for ResNet-50 model with TensortRT in Fig. 4. We can observe that beyond 2 streams (in both batch size of 1,8), there is barely any improvement in throughput, however the latency keeps increasing. We observed the same with VGG-19. Alexnet was an exception that showed improvement only for the small batch size (1). Our analysis is that spatial sharing is helpful only when there are sufficient resources that can take advantage of multiplexing, otherwise the tasks contend for the limited resources resulting in high execution overhead, reflected in the form of latency. Hence, streams are beneficial only for light weight models and lower batch sizes. However, we do see drawbacks of employing single stream for processing. We profiled for the single stream case, as shown in Fig. 5. We can observe that GPU remain idle for fairly large amount of time $\sim$700 micro-seconds between every execution. This interval corresponds to the time taken to notify the CPU of inference completion and 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 it the tasks were queued before. Note: Due to profiling the interval is extremely high; We observed that without the

profiling, callback has latency of around 40µs. Fig. 6 shows the profiler details with two streams. We can still observe the similar gaps in each of the streams, but since two streams have overlapped execution, the GPU utilization is higher in this case and is also able to mask the idle time issue that manifests with single stream. Hence, we restrict to using just two streams per DNN.

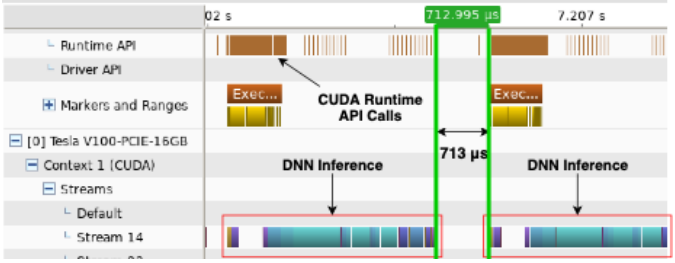


Fig. 5. Alexnet inference with single CUDA stream

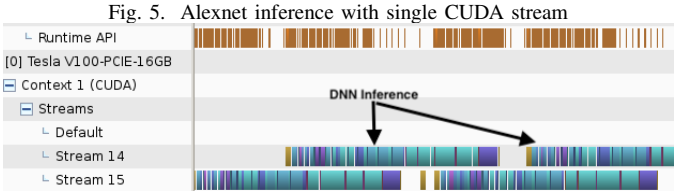


Fig. 6. Alexnet inference with two CUDA streams

IV. A First Step to Improving Multiplexing

A. Data Transfer To GPU

DPDK enables the Ethernet NIC to transfer packets to shared hugepage buffers that can be accessed with zero-copy overhead. Copying the payload from each packet separately to the GPU (using CUDA API calls such as cudaMemcpy) is very expensive. A better method is to first copy and aggregate data from packet payloads into a contiguous buffer and then DMA that buffer to the GPU. We name this method 'CPU-Copy'. However, even this method still requires the CPU to copy data from packet payload to a contiguous buffer and use the CUDA API to move the data to GPU. CPU-Copy taxes both CPU and memory to copy and store data in a contiguous buffer.

We developed NetML [1], to avoid the extra copy by running a GPU kernel that uses the GPU's DMA engine to perform scatter-gather from the host CPU memory. NetML utilizes NVIDIA's Unified Virtual Addressing (UVA) [21]. During application initialization, NetML pins the DPDK's packet memory buffer pool (mbuf pool). This ensures that all network packets reside in the pinned memory region. The time to pin 1GB memory is ∼20ms. Once the OpenNetVM application on the host side receives all the packets, a CUDA kernel is launched to gather data from these packets into a GPU buffer.

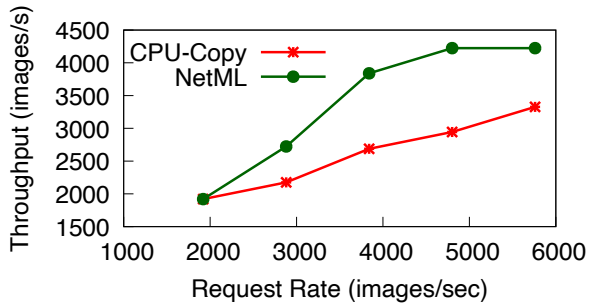


Fig. 7. Inference throughput of Alexnet (TensorRT) with V100 GPU

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

This approach provides two key benefits: i) it is efficient and timely; ii) more importantly, it saves host CPU cycles. Here, we modified the original NetML design to facilitate batching. Originally, NetML was designed for inference of a single image. To infer a batch of images, all the data in the batch has to be present in contiguous memory buffer in the GPU. As NetML starts building the batch in the GPU starting from the first received data packet, any interim packet loss would stall the progress and subsequently require us to restart the creation of the batch due to the lack of end-end (with the GPU memory being the destination) reliable, in-sequence delivery. To avoid this, we use a lightweight verification mechanism to ensure data for an entire image is present in the CPU from the payload of the received packets, before transferring data to the GPU. We continue transferring newly arrived, complete, images to the GPU until the desired batch size is reached. The CPU application then individually launches the DNN kernels to run on the GPU for inferring on the entire batch of images. We measured the CPU utilization of CPU-Copy and NetML by measuring percentage of CPU cycles spent in transferring data to GPU out of all the CPU cycles used in one inference task. We present the utilization of CPU in Table. II. We vary the batch size, *i.e.,* number of images transferred to GPU and inferred at once. We observe that, while inferring only one image, more than 10% of CPU cycles used for inference is used in data transfer with the CPU-Copy method. The proportion of CPU cycles expended for data transfer increases 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. Thus, it contributes a higher percentage of the total work for larger batch sizes. With a batch size of 32, nearly half of the CPU cycles are used just to move the data from network to GPU. But with NetML, we barely use any CPU cycles for data transfer. The maximum CPU cycles used is 0.21% to transfer batch of 32 images. Thus, NetML effectively offloads the data transfer to the GPU's DMA engine.

We evaluate NetML using NVIDIA V100 GPU. We use a fixed batch size of 8. In Fig. 7 we show the throughput achieved by Alexnet in a 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.

B. Notification from the GPU

GPU operations can be executed asynchronously with respect to CPU. This helps to free up the CPU after submitting the job

to GPU and also allows a single CPU thread to submit multiple tasks to 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 *cudaStreamAddCallback()* to notify the end of processing in the GPU by running a callback function on the CPU. Although the callback helps notify the end of a particular task, the current implementation of the callback poses challenges for GPU multiplexing. First, the callback blocks all the subsequent execution on the GPU until the completion of the execution of the callback routine on the CPU resulting in the idling of the GPU. Second, callback functions generated by the GPU forbid the CPU thread to run any of the CUDA API functions. This limitation requires additional CPU context and signalling scheme to perform any GPU related operations.

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) where the DNN is being executed. The CUDA event acts as a flag, which, when executed by the GPU 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 can then use another API function *cudaEventQuery()* to check if the event that we placed has been executed or not. This allows us to not only avoid the idling of the GPU, but also provide the flexibility to perform different GPU tasks within the same CPU thread context. The event checking is lightweight, taking about 5 μseconds in our system. As the DNNs usually take a few milliseconds to perform an inference, we check the event with a coarse interval of 1 milli-second to find out when the DNN computation has ended, thus providing the flexibility for the CPU thread to check event completion status at a desired frequency.

V. Summary and Future Work

With the growing need for Machine Learning capabilities for latency-sensitive applications, having them performed at an edge cloud is very attractive. Because of the more limited amount of computational capacity at an edge cloud (compared to centralized clouds), there is a need to better utilize CPU and GPU resources. We proposed adding controlled spatial multiplexing of the GPU, in comparison to only pure temporal sharing or just uncontrolled spatial sharing (like what is provided with CUDA MPS) of the GPU. We showed that controlled spatial multiplexing gave much lower, predictable, inference latency and higher inference throughput than the alternatives, by improving GPU utilization. We also proposed that timely notification of GPU task completions to the CPU improves latency and GPU utilization. Finally, we showed that using a small kernel thread in the GPU to fully take advantage of the GPU-resident DMA (as in NetML) can substantially improve movement of streaming data to the GPU, relieve CPU load and improve inference throughput substantially. Our current work is to develop a framework to implement all these capabilities on a CPU/GPU system that can support a variety of ML frameworks with limited or no modifications to their existing algorithms.

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 and K. K. Ramakrishnan, "Netml: An nfv platform with efficient support for machine learning applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 396–404.

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

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

[4] M. Satyanarayanan, "Edge computing for situational awareness," in *Local and Metropolitan Area Networks (LANMAN), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1–6.

[5] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[6] L. Liu, H. Li, and M. Gruteser, "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.

[7] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "Deepdecision: A mobile deep learning framework for edge video analytics," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1421–1429.

[8] H. Chang, A. Hari, S. Mukherjee, and T. Lakshman, "Bringing the cloud to the edge," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2014, pp. 346–351.

[9] X. Zhang, Y. Wang, and W. Shi, "pcamp: Performance comparison of machine learning packages on the edges," in {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[10] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[11] 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, pp. 2135–2135.

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

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *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.

[14] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[15] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[16] "Tensorflow serving," https://www.tensorflow.org/tfx/guide/serving.

[17] "Torchserve," https://pytorch.org/serve, 2020.

[18] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *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.

[19] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput cnn inference on embedded arm big. little multi-core processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[20] "Nvidia hyper-q," http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2020, [ONLINE].

[21] "Nvidia universal virtual addressing," https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf, 2011, [ONLINE].