NetML: An NFV Platform with Efficient Support for Machine Learning Applications

Aditya Dhakal and K. K. Ramakrishnan
Dept. of Computer Science and Engineering, University of California, Riverside
adhak001@ucr.edu, kk@cs.ucr.edu

***Abstract*—Real-time applications such as autonomous and connected cars, surveillance, and online learning applications have to train on streaming data. They require low-latency, high throughput machine learning (ML) functions resident in the network and in the cloud to perform learning and inference. NFV on edge cloud platforms can provide support for these applications by having heterogeneous computing including GPUs and other accelerators to offload ML-related computation. GPUs provide the necessary speedup for performing learning and inference to meet the needs of these latency sensitive real-time applications.**

Supporting ML inference and learning efficiently for streaming data in NFV platforms has several challenges. In this paper, we present a framework, NetML, that runs existing ML applications on an heterogeneous NFV platform that includes both CPUs and GPUs. NetML efficiently transfers the appropriate packet payload to the GPU, minimizing overheads, avoiding locks, and avoiding CPU-based data copies. Additionally, NetML minimizes latency by maximizing overlap between the data movement and GPU computation. We evaluate the efficiency of our approach for training and inference using popular object detection algorithms on our platform. NetML reduces the latency for inferring images by more than 20% and increases the training throughput by 30% while reducing CPU utilization compared to other state-of-the-art alternatives.

I. Introduction

Real-time applications such as autonomous driving, connected cars, surveillance, multimedia delivery and even network management, etc. are increasingly dependent on machine learning capabilities. These require low-latency, high throughput processing of the machine learning (ML) functions. Given their computational complexity, they often depend on cloud computing facilities residing in the network. Edge cloud platforms supporting network resident functions, such as Network Function Virtualization (NFV), can provide support for these applications, processing the data as it streams in from the network. An important goal is to process data (either for learning or for inference) at high throughput and low latency.

ML algorithms such as Deep Neural Networks (DNN) have seen increasing use as they improve in accuracy and effectiveness [1]. However, ML models have become complex, requiring significant compute power, typically requiring specialized accelerators such as GPUs to speed up the processing [2], [3]. A number of useful applications, require offloading the task of learning and inference to an edge or cloud server rather than the end-devices that generate the information since these devices have insufficient compute and storage. [4] However, offloading any task to edge or cloud server comes with overheads which increases end-to-end latency. This includes network protocol stack cost, I/O, system calls etc., referred to as a ”datacenter tax” [5]. This datacenter tax will only get higher with addition of overheads incurred by GPU API calls and GPU I/O. In our experiments, we have observed the I/O to the GPU is about 20-40% of the overall task. Any reduction of the time for GPU's I/O will be extremely valuable for lowering ML inference latency on edge or cloud servers.

978-1-5386-9376-6/19/$31.00 ©2019 IEEE

ML applications often operate on streaming data. Popular ML libraries and frameworks such as PyTorch, TensorFlow, Caffe combine CPU cores with GPUs for processing streaming data. But these platforms come with a number of challenges, as they are not optimized for performing inference over streaming data on these CPU/GPU platforms. One of the key challenges with faster inference of streaming data is to perform efficient transfer of the data received from the network to the GPU subsystem. Typical system architectures have GPUs residing on the Peripheral Component Interconnect Express (PCIe) bus, with a Direct Memory Access (DMA) engine helping to transfer a large chunk of data from a contiguous region in the host's memory. However, transferring data received as packets stream in one after the other can result in poor performance, with substantial latency, because of the need to copy the data into a contiguous page-locked pinned CPU memory region and setting up the DMA. Since low latency is critical for real-time systems, we need to achieve multiple goals at the same time: deliver the streaming data to the GPU complex as the data arrives over the network, minimize overheads on both the CPU and GPU for performing communication and data movement tasks; and maximize the parallelism in the GPU for performing machine learning tasks. We observe that just running the existing popular libraries on a CPU/GPU system for streaming data from the network tends to perform poorly.

Further, the emerging field of distributed machine-learning systems [6] transmits data between distributed nodes for training, inference and result aggregation. Platforms like DAIET [7] perform in-network aggregation of results computed by various ML worker nodes. Other approaches like Branchynet [8] create a single DNN, dividing computational layers between the end-device, edge-server and cloud-server, using the network to transmit data between these computing devices. These approaches involve a lot of data transfers, and perform ML operations in multiple network-resident systems. Therefore, getting the network data to the GPU quickly and using the CPU and GPU effectively are valuable in all these domains.

There are several approaches possible for efficiently moving data to the GPU. NVIDIA's GPUDirect is one where data is directly transferred from a Fibrechannel NIC to the GPU. However, this method is limited to specialized NICs [9] and requires modified drivers that are not available for most other NICs. The goal of our work is to design mechanisms that can be more broadly useful for Ethernet NICs and common off-the-shelf (COTS) hardware with various GPUs.

There are other approaches to decrease inference latency of course, which involve compromises in the ML applications themselves, having to tradeoff application capability for throughput. DNN implementations in ML libraries have changed the DNN by either reducing the number of neural network layers to obtain faster inference. E.g., Yolo V3 [10] uses 106 layers in the neural network while a lightweight variant Tiny-Yolo only uses 23. Alternately, for object detection, the model may reduce the resolution of the input image to reduce the computation needed, to achieve speedup. Other techniques, such as binarized neural networks [11], do model compression and model pruning to have the DNN models take less memory and reduce computation for each inference. Hardware approaches such as NVIDIA GPUs with specialized Tensor cores provide speedup on inference. Eyeriss [12], which increases the re-usability of the data in the neural network, is another option. Most of these options are beneficial when all of data exists in memory, which implies significant latency because they have to wait for buffering all of the data. The options of modifying ML applications to tradeoff accuracy for throughput and performance and using special-purpose hardware are not entirely the most desirable, and have limited applicability for use with streaming data.

We have built an NFV platform, NetML, for supporting ML applications on a COTS hardware platform. Our contribution is to develop a fundamentally different approach to data movement from the host to the GPU and efficiently transfer data from network packets. The current CPU and GPU systems have evolved in a way that the process of transferring data from the host (CPU) to GPU requires the data to be "pushed" to GPU. With the GPU on the other end of a PCIe BUS, the host has to communicate the address, data size and other information to set up the GPU's DMA engine by executing multiple CUDA API calls and driver functions. This process of "pushing" data from CPU is expensive and is incurred for each packet scattered around in host memory. NetML, avoids this unnecessary overhead and latency of processing CUDA API and the driver function repeatedly by initiating the data transfer from GPU subsystem itself. NetML"pulls" the data by starting the data transfer by providing the GPU threads the address of multiple network packets and the data transfer of network packets is initiated from the GPU subsystem itself. Thus, expensive CUDA API and driver functions are invoked just once. Moreover, we exploit the shared memory huge-pages of the NFV platform [13] to eliminate data movement by the CPU, and pin this shared memory to allow the data to be accessed by the GPU subsystem as it streams in, with low latency and minimal overhead.

NetML also maximizes the overlap between GPU execution and the DMA-based data transfer to the GPU complex for subsequent ML stream data processing. In addition, NetML seeks to maximize the overlap of CPU NF processing with the GPU execution. Thus, we reduce latency and improve throughput substantially by eliminating CPU data movement, streaming data to the GPU subsystem and maximizing overlap of CPU, DMA and GPU functions.

With NetML, efficient data transfer to the GPU cuts the task completion time for inference in neural networks. In the experiments using image detection ML libraries, we use NetML for inference of a large image that is received by the edge server in multiple packets. NetML helps achieve maximum pipelined parallelism across all components, including the network link carrying the packets, DMA setup and transfer to GPU, and initiation of the inference engine. NetML thus reduces the time to infer a single image by at least 20% compared to a traditional implementation of the same libraries without NetML's optimizations. The NetML platform, runs on COTS hardware and allows plugging in of existing ML libraries and models while providing speedup both for learning and inference for streaming data, without requiring modifications to those applications.

II. Motivation

There are a number of challenges associated with performing ML with streaming data, both for learning and inference, in an edge server. First, the ML models are complex and compute-intensive. The popular Convolutional DNN models such as AlexNet and ResNet50 require very high number of multiply and accumulate (MAC) operations (666 million and 3.9 billion MACs, respectively) [2]. It is well understood that even with modern high-end servers, CPU-based processing is just not sufficiently fast. In contrast, the GPUs can spawn thousands of lightweight threads, which can perform these MAC operations in parallel more effectively, thus enable to achieve very high speedup.

More importantly, as data streams through the edge-server at high bandwidth, the time available at the edge server to process the data is limited. Furthermore, storing the streaming data for learning or inference subsequently might not be feasible due to significant storage requirement. Thus, it is necessary that we learn and perform inference as the data is streamed through the edge server, without incurring large latencies from buffering the data. Using GPUs to perform ML is likely the only option at the edge server.

To reiterate this point, we performed experiments in an edge server to observe the time required by typical ML application to perform object detection on an image data being forwarded through it. We utilized two alternatives to perform inference in the data. First, in **CPU-Only** approach, we extract the streaming data and process it in ML applications such as PyTorch and Darknet (Tiny-Yolo) running in the CPU itself. Alternatively, we use the **CPU-GPU** based approach, where we DMA the data received from streaming packets (one packet at a time) to the GPU and process the image data in a DNN model running in a GPU.

We used an Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz processor to conduct inference by a CPU and the NVIDIA Tesla P100 GPU to perform inference while using a GPU. We used VGG-11 in PyTorch and Tiny-Yolo in Darknet to get the time for inference on one image. In Table I, we can observe that, CPU-Only approach is highly inefficient to perform the ML computations, while the CPU-GPU approach is nearly ***100x*** times more efficient than CPU-only approach. Of course, this is well-known.

TABLE I
TIME FOR OBJECT DETECTION ML APPLICATION IN CPU AND GPU

ML Application	CPU-Only (ms)	CPU-GPU (A#1) (ms)	GPU-Only (ms)
PyTorch (VGG-11)	636	8.6	3.95
Darknet (Tiny-Yolo)	1067	11.3	4.68

However, to understand the overhead and complexity of processing the streaming data, we compare the time taken to run the model on the GPU when all the data is locally available in GPU (GPU-Only). We believe understanding the tradeoff is useful when we want to run ML applications on edge- servers. We observe that in comparison to the processing in the GPU-Only (i.e., all data available at the GPU) case, the CPU-GPU (A#1) approach is is still considerably worse. There is almost a ***2-3x*** increase in inference time with A#1. The additional delay with A#1 is caused by having to wait for the data to arrive over the network to the CPU and then eventually having to transfer to the GPU from the host memory.

Thus, getting the required data to the GPU quickly and efficiently is essential. In NetML we tackle this problem by efficiently transferring the streaming data to the GPU. Thereby, we significantly reduce the overall task completion time.

III. System Architecture

NetML is a capability built on top of a DPDK [14]-based OpenNetVM [13] NFV platform. OpenNetVM utilizes shared memory in user space, with the Ethernet NIC directly DMAing the packet into the shared memory, bypassing the kernel network stack. The architecture of NetML is shown in Figure 1. We first describe a number of key components in the system and then how they fit together in NetML to support efficient ML processing on the system.

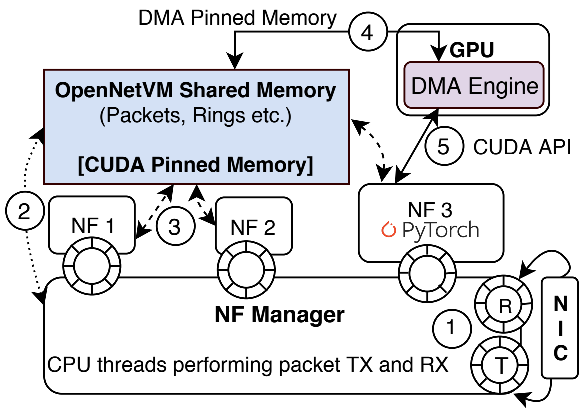

Fig. 1. **NetML Architecture**: 1.NF manager polls NIC's Rx and Tx rings for packet descriptors. 2.NF manager can access the packets in OpenNetVM's shared memory. 3. NFs have zero copy access to the packets in shared memory. 4. Shared memory is "Pinned" (sec.III-D1), allowing GPU zero-copy access. 5. ML NFs can use GPU using CUDA API

A. NF Manager

The NF manager is the primary process that manages the NFV platform. It reserves a shared memory buffer in the Linux Hugepage memory for buffering the incoming packets from NIC. The NF manager also creates the transmit and receive ring buffers, which are necessary for packet processing and communication between the NFs. It also initializes multiple CPU threads for packet processing at the start-up. The major threads include i) *Rx threads*: poll the NIC rings to obtain the descriptor entry of the incoming packets and de-multiplex the packets to the NFs by checking the flow table entry for each packet. ii) *Tx threads*: poll the NFs transmit rings and place the packet descriptor in the NIC's Tx ring for transmitting out the packet, and iii) *NF manager main thread*: performs NFV platform initialization, bookkeeping of the registered NFs, and updates statistics.

B. Shared Memory

The NF manager creates a shared memory buffer during application startup by reserving Linux huge page memory. This hugepage memory buffer is used to create Mbufs to buffer incoming packets and create the NF's ring buffers. When new NFs are started, they map the shared memory to their address space so they can have zero copy access to the packets buffered there. In OpenNetVM, NICs directly DMA packets to the shared memory. It is important that the shared memory buffer not get swapped out while the NF manager and NFs are running. The NF manager locks the memory pages of the shared memory using the DPDK Environment Abstraction Library (EAL), guaranteeing that the shared memory buffer will remain in RAM and not get swapped out to disk.

C. Network Functions (NFs)

NFs in OpenNetVM can run as processes or containers for in-network applications (e.g., forwarding, IDS, encryption, etc.) that can process the packets in the shared memory. When an NF initializes, it maps the shared memory reserved by the NF manager, so it has zero-copy access to the packets. In OpenNetVM, each NF maintains a set of ring buffers, where the packet descriptors can be enqueued by the NF manager or an upstream NF. Once the descriptors are enqueued, the NF can then process the packet. In NetML, we run ML applications as NFs so the ML learning and inference could be performed on the stream of packets, as they arrive. Furthermore, ML NFs also *pin* shared memory by using the CUDA API to set up efficient data transfer to the GPU. The reason to pin the shared memory and the methods used to pin it are explained in section III-D.

D. CUDA Pinned Memory

In ways similar to NICs, GPU devices also utilize DMA to transfer data from the host's memory to device memory. CUDA runtime requires that data being transferred to the GPU reside in page-locked memory (also known as 'pinned memory') precluding it from being swapped out while the transfer is in progress. If a data transfer to the GPU is initiated for a non-pinned memory buffer, the CUDA runtime creates a new pinned buffer and the CPU copies the data into the newly created pinned buffer before initiating the DMA to transfer the data to the GPU. This extra copy incurs additional latency and CPU resources. Furthermore, it is recommended in CUDA's best practice guide [15] that pinned memory should be allocated when the application is started, as pinning the memory during the application runtime results in additional latency.

Thus, to transfer data received in packets efficiently, it is important to have the data reside in pinned memory before transferring the payload to the GPU. Note that incoming packets in shared memory are not guaranteed to be put in a contiguous space by the NIC driver and DPDK's libraries. We overcome this challenge by pinning the entire OpenNetVM's shared memory buffer, so every received packet reside in the pinned memory before we initiate the DMA transfer to GPU. The techniques we used to pin the shared memory used by NFs in NetML is explained in Sec. III-D1.

We should note that the shared memory buffer is already page-locked by the help of the DPDK library when the NF manager starts. *Pinning* that shared memory using CUDA API is still necessary as the CUDA runtime environment does not recognize if a memory buffer is page-locked by another application or not. Furthermore in CUDA, *pinning* a memory buffer goes beyond just page-locking the memory buffer. It also enables the Universal Virtual Addressing (UVA) feature, which we utilize extensively in NetML. Generally, the GPU subsystem has its own memory and memory address translation function. Thus, a pointer to the data residing in host's (CPU) memory cannot be used in CUDA kernels running in the GPU and vice versa. The exception is with GPUs using the UVA feature, where CUDA kernels can use the address of host memory region that is *pinned*. With UVA, if a GPU thread encounters a pinned host memory address, the GPU's DMA engine will initiate data transfer from the host memory location immediately and the GPU threads can continue processing the data. This feature is also marketed as ”Zero-copy” memory.

*1) **Pinning a Memory Buffer**:* We explored two options for pinning a memory buffer in CUDA and using it as OpenNetVM's (CPU) shared memory.

- **Dynamically Allocating and Pinning Memory:** In this approach, we modify the DPDK library by overriding DPDK's default method of reserving the hugepage memory and instead, we allocate the pinned memory buffers using the CUDA API function *cudaHostAlloc()*.We then use this dynamically allocated pinned memory as OpenNetVM's shared memory.
- **Pinning DPDK's hugepages memory:**In this approach, we pin the shared memory allocated by the DPDK (NF manager) process by using the *cudaHostRegister()* API function. This API function allows us to pin an existing pageable host memory buffer (e.g., buffer generated by *malloc()*) and provides the same features as pinned memory allocated by other CUDA API functions such as *cudaHostAlloc()*. When an NF starts, it accesses the address of every page of the shared memory allocated by NF Manager and uses *cudaHostRegister()* to pin all of the shared memory. Thus, all the packets received by the NFV platform and buffered in shared memory will reside in pinned memory as well. Because this operation works with hugepages memory, it does not incur any additional overhead for packet processing by NFs, as seen in the Table II. We adopt this approach for NetML. It also does not interfere with the poll mode drivers of DPDK or the NFV platform's ability to run multiple NFs.

To test the efficiency of these options, we dynamically allocated a memory buffer of 2 gigabytes using the CUDA API *cudaHostAlloc()* function and used it as shared memory for the OpenNetVM NFV platform. With this approach, we observed the packet processing throughput of OpenNetVM/DPDK applications decreased significantly (see Table II).

Next, we DPDK utilizes hugepages for shared memory, thus decreasing the number of distinct memory pages needed, thereby also reducing the number of translation lookaside buffers (TLB) entries needed. This speeds up the translation of virtual page address to physical page address. On the other hand, allocating shared memory with the CUDA API only allocates memory with standard 4-kilobyte page size, resulting in a huge number of entries in TLB and much higher TLB miss rate. It also impacts the poll mode drivers and libraries required to host multiple NFs in the NFV platform. This is why the approach used by NetML is preferred.

TABLE II
FORWARDING THROUGHPUT OF OPENETVM USING PINNED MEMORY

Pinning Method /Packet Size	256 bytes	512 bytes
Dynamically Allocating Pinned Mem.	3.1 Gbps	5.1 Gbps
Pinning DPDK's Hugepages	10 Gbps	10 Gbps

E. Transferring the Data to the GPU

We explored three alternatives to transfer the streaming data received in OpenNetVM's shared memory from the NIC to GPU.In all three alternatives, multiple packets constituting an image are sent from a traffic generator. Associated with the image data packet payload is metadata containing the image ID, data offset, a packet ID and the number of packets for the image. The meta-data is used to reconstruct the image in the host and GPU memory. The image ID identifies the right image buffer, with the data offset helping transfer the image data appropriately into the image buffer. The CUDA API functions and CUDA kernel launches are asynchronous with respect to CPU. i.e., CPU is not notified when the CUDA function has finished processing. The packet ID and count help determine when an entire image is received so that the DNN image evaluation kernel is invoked after the entire image is in the GPU. The ordering of the execution of CUDA functions, i.e., the GPU executes CUDA kernels in the order they are launched. Three different data movement alternatives are described below.

*1) **Per-Packet cudaMemcpy (A#1)**:* In this approach, when an NF receives the packets carrying the data that is to be transferred to GPU, it calls a CUDA API function, *cudaMemcpyAsync()*, individually for each packet to initiate the transfer of the packet's data to GPU memory. The *cudaMemcpyAsync()* function is processed by a CPU thread and passes the necessary information, i.e., memory address and size of data, to CUDA runtime to initiate the DMA. CUDA runtime then schedules the GPU's DMA engine to asynchronously DMA the data. Figure

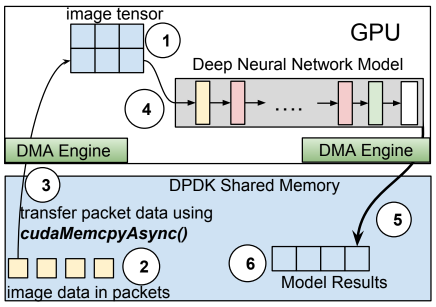

Fig. 2. Per Packet *cudaMemcpy* (A#1) **1.** ML application transfers the machine model and allocates a memory buffer in GPU. **2.** streaming data (packets) arrive in DPDK shared memory. **3.** Payload is transferred to GPU bound memory location asynchronously using *cudaMemcpyAsync()* **4.** When all the data has been received in GPU, it is inferred by object recognition model. **5.** Once the result is computed, it is transferred back to shared memory. **6.** CPU based ML application receives the results.

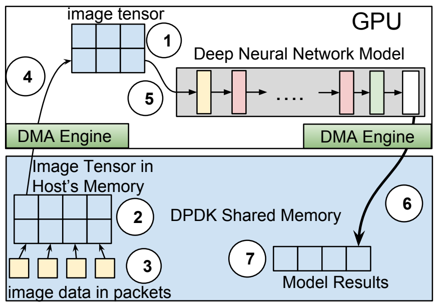

Fig. 3. CPU Copy and Batch *CudaMemcpy* (A#2) Unlike Per-Packet cudaMemcpy, CPU first copies the packet data into a separate buffer. **2.** A host memory bound tensor is created **3.** Payload is copied from packets to host (by CPU) bound tensor **4.** When the host bound tensor has received all the data, data is transferred to GPU bound tensor with single *cudaMemcpyAsync()* function call. **5.**The data is processed by a DNN in GPU and **6,7.**the results are DMA to host memory.

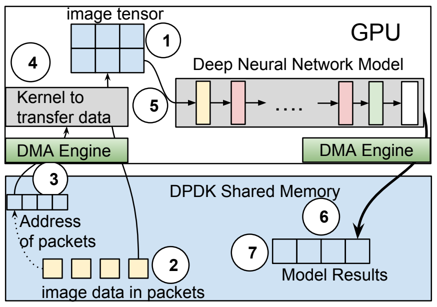

Fig. 4. NetML. NetML improves (A#1) by having the DMA transfer data from packet buffers as packets arrive, using UVA: **3.** Pointers to the packets are sent to a CUDA kernel. **4.** GPU threads use UVA "zero-copy" to write data to image tensor. **5.** When the image tensor has received all the data, it is inferred by object recognition model

2 explains the steps taken by an ML application to transfer and infer the data using this alternative. As the ML NF receives application data (eg. an Image), the image data is transferred from packets to GPU by using *cudaMemcpyasync()* and stored in a buffer called tensor before the DNN kernels infer it.

*2) **CPU Copy and Batch cudaMemcpy (A#2)**:* In this alternative, the ML NF (running in the CPU) creates a 'pinned' buffer in the host's memory. The payload data from packets are copied using CPU threads into this pinned buffer in contiguous memory. This CPU-based data copy can be a significant overhead. Once the entire application level data (e.g., an image) has arrived and copied into the buffer, the CPU-based NF launches the ML (e.g., PyTorch) application to run inference on the image as shown in Figure 3. This method uses the PyTorch API to transfer the data to GPU. With the help of NVIDIA's profiler [16], we verified that PyTorch uses a single *cudaMemcpyAsync()* function to transfer the image data to GPU before beginning inference on the image with the model in the GPU. We believe that this is the approach that most applications use, for inferring data using the GPU. I.e., wait for the entire application level data to be available before running the inference algorithm on the image using the GPU.

*3) **NetML**:* In NetML, we utilize a CUDA kernel to transfer the data to the GPU resident buffer. When packets arrive in the shared memory with data in the payload, we launch this data-transfer CUDA kernel with multiple GPU threads and provide the addresses of the packets as the arguments, which are obtained from OpenNetVM's receive rings. GPU threads use the memory address to read the packet's payload data. As the packet and the payload are in the host's memory, the GPU subsystem will use the Unified Virtual Addressing (UVA) feature and initiate the DMA to obtain the data (packet payload) and store it in the GPU's memory buffer. Once the entire application level data (e.g., an image) is transferred to GPU, the ML application (e.g., DNN kernels) in the GPU will process the data as shown in Figure 4.

It should be noted that the two alternatives, **Per-Packet *cudaMemcpy()*** and **CPU Copy and Batch *cudaMemcpy()***) involve CPU threads invoking CUDA API functions and initiating the DMA. On the other hand, in NetML, the DMA is initiated by the GPU subsystem when the GPU threads in the CUDA kernel access the host's memory address. Furthermore, in second alternative, **CPU Copy and Batch *cudaMemcpy()***, the data from the packets is copied to a host memory buffer using a CPU thread before performing a single cudaMemcpy() to transfer the batch of data to GPU. While in NetML, the GPU subsystem initiates the DMA as soon as the packets arrive in the host CPU's memory.

IV. Experimental Results

Our experimental testbed has Dell PowerEdge R730 servers with Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz processors, each with 256GB RAM, running Ubuntu SMP Linux kernel 4.4.0-119-generic. Each system has a NVIDIA Pascal P100 GPU with 12 GB device memory, 56 SMs, 3584 FP32 CUDA cores and dual DMA engines. For these experiments, nodes were connected back-to-back with dual-port 10Gbps DPDK compatible NICs to avoid any overheads related to switches. We make use of DPDK based NF to generate line rate traffic.

We conducted a number of experiments to evaluate NetML's performance. The first set of experiments show the improvement in the data transfer throughput, moving data from network packet payload to the GPU, with NetML. The second set of experiments demonstrate the improvement in throughput for an image detection ML application running in NetML and also to show the improvement in throughput for training a MNIST [17] model for recognition of handwritten digits.

A. Data Transfer to GPU

To test the data transfer time taken by each alternative (A#1, A#2 and NetML), we sent data streams of various lengths to the server hosting the ML application over a 10 Gbps ethernet link. We used the tcpreplay application to generate the UDP packets containing 1 kilobyte chunks of raw image data as payload and transmitted the packets at 10 Gbps. We repeated the experiment across multiple data sizes, evaluating the time spent to transfer the entire data stream, i.e., time between first packet entering the shared memory to last packet's payload being transferred to GPU. We also profiled the GPU and created execution profiles using NVIDIA profiler, while transferring an image of 500 kilobytes.

Figures 6, 7 and 8 show the results from the NVIDIA profiler on the three alternatives. They show the timeline of CUDA calls made for transferring a 500 kilobyte of image data as 500 packets of size 1 kilobyte each and processing it by DNN. The profiling clock on the X-Axis shows time since the application has started. We highlight the time taken for transferring data from host memory to the GPU, i.e., time period between arrival of the first packet and the instant all of the image data is transferred to GPU and the first DNN kernel starts. This data transfer time is shown in green tab highlighted on the X-Axis in the profiling diagram.

We can see in Figure 5 that **per Packet cudaMemcpy (A#1)** is the slowest approach, with rapidly increasing latency as the image size increases. The other approaches also have their latency increase as the data transfer size increases, although they remain much lower than **(A#1)**. Observing the profiling timeline in Figure 6, we see that the time taken to transfer data is more than 4 milliseconds. The cause of additional latency with **(A#1)** is the overhead of *cudaMemcpyAsync()* performed on each and every packet. The average time taken for launching *cudaMemcpyAsync())* function is about 8 μseconds. When transferring data from a large number of packets, this overhead gets substantial. Moreover, the frequent *cudaMemcpyAsync()* calls also occupies the GPU DMA engine. This precludes other applications from utilizing the DMA for moving their data.

We can also see from Figure 5 that NetML is the fastest among other alternatives for all data sizes. This is due to the fact that in NetML, we initiate data transfer kernel to DMA the data from packet payload as soon as the packet is in OpenNetVM's shared memory. It does not need to wait for a certain size buffer to fill up before transferring data to GPU as in **CPU Copy and Batch cudaMemcpy (A#2)**.

The profiling timeline in Figure 7 shows that **(A#2)** takes more than 1.37 milliseconds to transfer the data. This latency is caused by **(A#2)** waiting for all 500 kilobytes of data to get to the OpenNetVM's shared memory and copying it to the host side buffer as well as setting up *cudaMemcpyAsync()* call to finally transfer the data to GPU.

Finally, Figure 8 shows that NetML cuts the data transfer time by more than half of **(A#2)**, to about 630 μseconds. After receiving a minimum sized batch of packets, NetML launches a 'Data Transfer Kernel', which initiates the DMA transfer of packet data to the GPU. The runtime of this data transfer kernel is very short, so it does not occupy the DMA engine for long, unlike **(A#1)**. Moreover, unlike **(A#2)**, NetML does not have to wait for the packets to be copied to a contiguous host buffer and setting up an expensive *cudaMemcpy()* call. As a result, it is faster than both the other alternatives. Furthermore, NetML is faster than **(A#2)** by a similar amount of time across all data sizes. This is because, limited by the 10 Gbps link bandwidth for receiving the data, **(A#2)** has to wait for the packets of the image to be delivered from the wire.

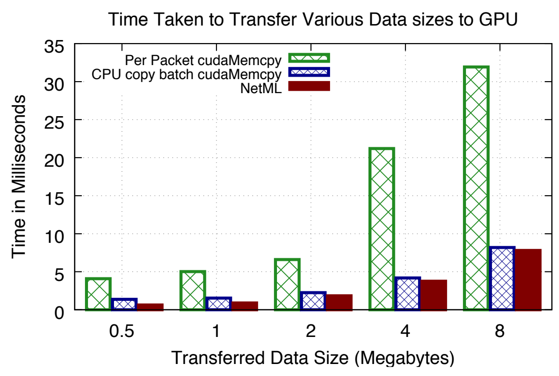

Fig. 5. Data transfer latency with NetML compared to cudaMemcpy(), receiving data at 10 Gbps

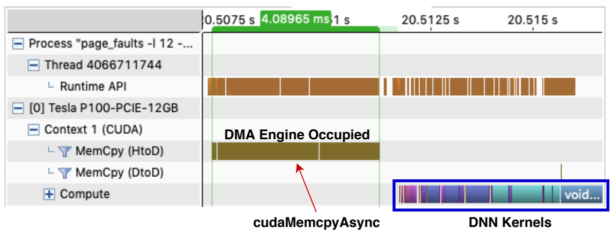

Fig. 6. GPU profile of Per Packet cudaMemcpy (A#1)

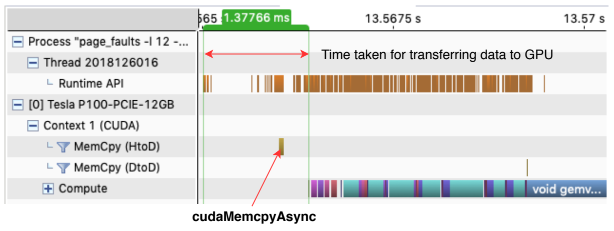

Fig. 7. GPU Profile of CPU copy and Batch cudaMemcpy (A#2)

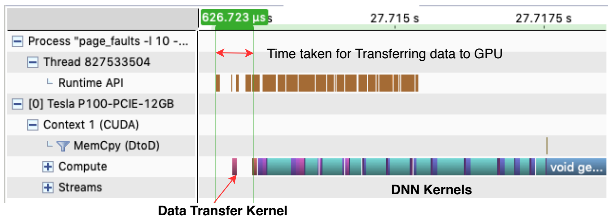

Fig. 8. GPU Profile of NetML

B. CPU consumption for transferring data

Although the operations performed in the GPU are asynchronous, CPU cycles are consumed for calling CUDA API functions as well as launching CUDA kernels in GPU. We measure how much CPU cycles are consumed for transferring data. We counted the CPU cycles taken to call *cudaMemcpyAsync* for **(A#1)**, the time taken for copying the data from packets into a host side buffer for **(A#2)**, and time taken to launch the GPU kernel to transfer packet for NetML.

In our experiments we found that it took an average of 25 μseconds to launch a CUDA kernel in GPU. However, in NetML we can initiate the data transfer for multiple packets concurrently by utilizing multiple threads in GPU. The result of CPU consumption for transferring an image of various size

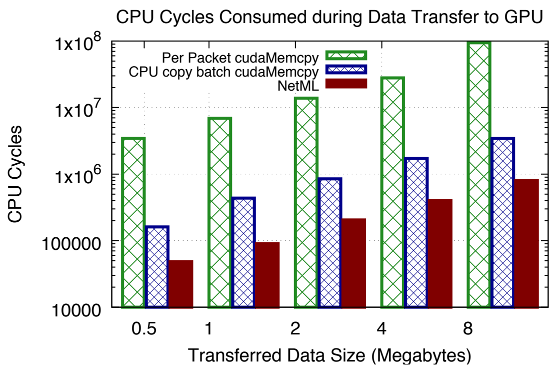

Fig. 9. CPU cycles consumed by NF for transferring a single image of various sizes to GPU

is shown in Figure 9. Since we initiate the DMA by running a kernel in GPU in NetML, there is very little overhead for the CPU to transfer the packets. However, in **(A#2)**, the CPU is used to copy the data from packet payload to an application's contiguous buffer before DMAing to the GPU, thus, consuming a lot of CPU cycles. **(A#1)** performs worse than both the other alternatives because of the high CPU cycle consumption for making expensive *cudaMemcpyasync()* calls for each of the received packets.

C. Object detection with streaming image data

For experiments evaluating the throughput of an ML application performing object detection on streaming image data, we use two applications. First was PyTorch 0.4.0 [18] with a pre-trained VGG-11 [19] object detection model obtained from PyTorch's TorchVision library. The second was the Tiny-Yolo object detection and bounding algorithm based on the Darknet library. We run PyTorch and Tiny-Yolo each as an NF in the CPU in NetML. The CPU based NF is essentially an NF on OpenNetVM with the optimizations we discussed here for enabling data movement. We briefly describe the two libraries, for completeness.

PyTorch is an open source ML library based on the Torch ML library. Torch provides abstractions for NVidia's CUDA and the cuDNN neural network library. Torch's primary data structure is a multi-dimensional matrix, a tensor, which can be allocated in both the host's memory as well as the GPU's memory. PyTorch also has the TorchVision library, which provides multiple pre-trained object detection models trained on Imagenet dataset like AlexNet, VGG, ResNet, and DenseNet.

Tiny-Yolo is an object detection application based on the Darknet library. It performs object recognition and creates a bounding box around the detected object. Tiny-Yolo uses single pass object detection algorithm, where the image data only passes through the neural network once and the objects in the image are detected and bounded by a box.

Image data is generated separately on another server and is carried over UDP packets on a 10Gbps link to the server running NetML. The image data primed for inference in PyTorch is generated by converting a PNG image of resolution 224×224 to a raw image array composed of floating point values for red, green and blue (RGB) values of each pixel. The resulting image data array is of size $224\times224\times3$. This array is transported over 2352 packets, each being 256 bytes, 1176 packets of 512 bytes and 784 packets of size 768 bytes to the server performing inference. The packet payload includes meta-data describing which section of the image array the packet payload belongs to, the sequence number of the packet and additional information such as a fileID so that the inference server can direct the data to the appropriate buffers. A similar operation is performed for an image intended for Tiny-Yolo. However, a larger image, with a resolution of 416×416 is used. We experimented using the alternative approaches for data transfer of streaming image data, and measure the latency of inference for each image.

D. Inference of a Single Image

We see from Figure 10 that NetML yields the fastest inference at 4.3 msecs. This time is relatively independent of packet size. With **CPU copy and Batch cudaMemcpy (A#2)**, the inference time is higher, at 5.5 msec. The time improves a little as the packet size increases. The improvement for larger packet sizes is due to improved efficiency of the CPU copying a larger amount of data each time, as the packet size increases. For **Per Packet cudaMemcpy (A#1)**, which uses *cudaMemcpyAsync()* to transfer data to GPU, latency is much higher 12.3 msec. The latency for inferring decreases significantly as the packet size increases as there are fewer CUDA calls being made. However, it is still higher than NetML across all packet sizes. The image detection performed in Tiny-Yolo also yields similar results as shown in Figure 11, where NetML infers an image at 4.7 msecs while **(A#2)** has an inference time of 5.9 msec (for the largest packet size). **(A#1)** is the slowest at 8.1 msec per image, even at the largest packet size.

We also used the NVIDIA profiler to generate profiles during the inference of an image for each alternative. We also measured the time spent by the application for receiving the packet data at the NF running in CPU. With the profiler we measured the additional time it takes to move the data to GPU, and the time spent on processing the data via DNN kernels in the GPU. As we can see in the Figure 12, the time to receive the data from the network and the model execution time are about the same across alternatives. However, with NetML, the time to transfer data to GPU is smaller than with both **(A#1)** and **(A#2)**. In **(A#1)**, we can see the data transfer taking more than half of total time of processing an image. The reason for the extra latency in **(A#1)** is the overhead of calling individual *cudaMemcpy()* to initiate DMA for each and every packet. **(A#2)** has to wait for the entire image data to be available before transferring the data to the GPU. The additional latency is also due to time taken for copying the data from packets into a buffer as well as setting up a *cudaMemcpy()* call for a bigger chunk of data. NetML reduces the latency by starting the DMA by data transfer kernel in GPU and initiate the data transfer as a 'cut through' (not waiting for the entire image data).

We also conducted experiments to evaluate the improvement of throughput while training on the data. We created a convolutional neural network with two convolutional layers, one dropout layer and two fully connected layers and used 60,000 images from the MNIST database in various batch

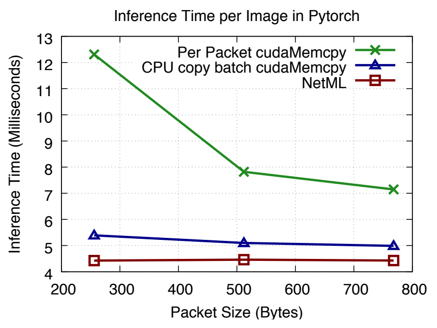

Fig. 10. Pytorch image inference time: data movement alternatives

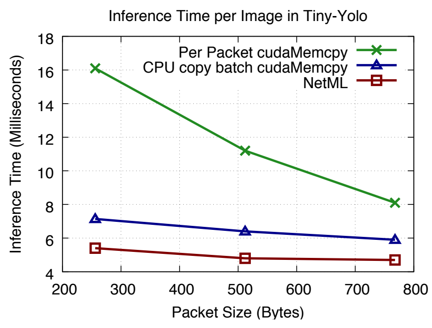

Fig. 11. Image Inference Time with Tiny-Yolo

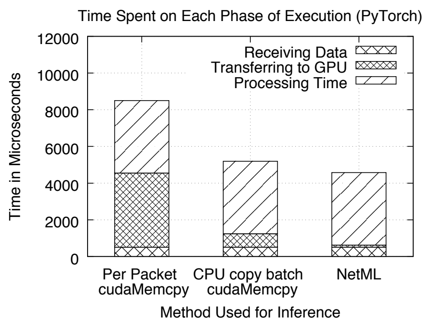

Fig. 12. Breakdown of Inference time using Pytorch

sizes to train the model. Each image is a monochrome handwritten digit of size 28×28. The batch size is the number of images propagated through the neural network before performing optimization. As we experiment with streaming data, we trained the model for only one epoch. We used 10,000 test images to verify the accuracy of the trained model.

We experimented with two data transfer alternatives, **(A#2)** and NetML to transfer the data from network packets to the GPU. The throughput, i.e. number of image files trained per second for NetML, is shown in the table III. We can see that increasing the batch size results in better throughput as the model takes similar time to train irrespective of batch size. However, the accuracy of the trained model decreases as the batch size increases. Moreover, we see that NetML has at least 30% higher throughput that **(A#2)** for all batch sizes. Furthermore, for experiments with batch size 100, we utilized a 20 Gbps link as we saturated the 10 Gbps link while sending data.

TABLE III
NO. OF FILES TRAINED PER SECOND FOR DIFFERENT BATCH SIZES

Alternatives/Batch Size	50	80	100
Throughput, **(A#2)**	22727	30005	41518
Throughput, **NetML**	29585	40113	56497
Improvement with NetML	30.1%	33.6%	36%
Accuracy with NetML	95%	93%	92 %

V. Related Work

GPU Based Packet Processing: A number of studies have evaluated using GPUs for accelerating packet processing. PacketShader [20] utilizes GPUs to process packet headers for switching and routing, and SSLShader [21] provides high throughput SSL processing in the GPU. Snap [22] built a packet router using GPU. GPUNFV is a NFV platform that [23] creates NFs in GPU for packet processing. GPUnet [24] creates a networking layer for GPUs and offers socket level abstractions for programs running in GPUs. NBA [25] utilizes an adaptive load balancer to balance the processing at NFs running on both the CPU and GPU. G-NET [26] creates a scheduling and virtualization framework to share GPU resources across multiple NFs running concurrently. All of these works employ batching of packets into a buffer before transferring them to the GPU. While they exploit the parallelism offered by having a large number of GPU cores, none of them effectively address the data movement problem, and essentially adopt the approach outlined in the alternative **(A#2)**. Unlike other works noted above, APUNet [27] uses integrated GPU to process packets and eliminates the data transfer over PCIe BUS. However, integrated GPUs have much fewer compute cores than discrete GPUs, thus, they would not be able to provide high speed-up for processing DNNs. Therefore, the data transfer problem remains relevant as processing DNNs require much powerful discrete GPUs.

Accelerating Training and Inference of ML Applications: Popular DNN algorithms such as Inception and ResNet require significant processing and storage, resulting in high latency for inference. Studies to produce light-weight versions of DNN which can run on edge devices and have lower compute cost and faster inference have been attempted. SqueezeNet [28], with fewer parameters and a small model size runs much faster. Similarly, SqueezeDet [29] is built upon SqueezeNet and has low computational overhead, and is able to achieve real-time object detection. Other approaches to accelerate ML training and inference use binarized neural network [11] or compress the DNN model [30]. Our approach in NetML complements these simplifications and optimizations of the ML algorithms. ML applications can further be accelerated by using specialized hardware such as the NVIDIA's tensor cores [31], Google's TPU [32], MIT's Eyeriss [12]. However, these hardware solutions come as a co-processor or subsystem that sits on the other side of the PCIe or serial bus. It is necessary to transfer the data and instructions to these hardware accelerators in a manner similar to a GPU. The optimization performed in NetML to transfer data could benefit these hardware accelerators as well.

Processing Streaming Data: Apache Spark [33] and Storm [34] are popular platforms for stream data processing. However, they do not natively support GPU hardware. G-Storm [35] extends Apache Storm to process streaming data in the GPU. GPL [36] performs query processing on the GPU for streaming data. However, these do not optimize data transfer to the GPU and use the Linux kernel networking stack. NetML can provide the benefit of both DPDK-based OpenNetVM's faster zero-copy, poll-mode packet processing and the more efficient data transfer to GPU for an edge server running these stream data processing platforms.

Distributed ML algorithms: Distributed ML algorithms such as distributed DNN often use a network of compute servers to train large ML models. Distributed DNNs are usually trained by either partitioning data across several compute nodes running the same DNN model and periodically aggregating the results (data parallel) [37], or by splitting the DNN model over multiple compute nodes and feeding partial results computed by one node as input to another node (Model parallel) [8]. In both cases, a large number of model parameters or training data has to be copied from one compute node to another. This data transfer cost can be significant. The work [38] shows that the overall task-completion time can be reduced by better pipelining of the communication and processing tasks. Others, such as DAIET [7] perform distributed DNN training on multiple machines to reduce the amount of information exchanged between the compute servers with help of a data aggregating middle-box. NetML may be used in this context to speed up the data movement at each of the compute nodes.

VI. Conclusion

We presented NetML, a edge-server enhancement speedup for Machine Learning applications to process streaming data on OpenNetVM, a DPDK-based NFV platform. NetML runs on hybrid CPU and GPU based system architectures, with the GPU resources managed by the CUDA runtime environment. In NetML, we pin the DPDK process's shared memory to achieve low-latency, efficient transfer of streaming data to the GPU. NetML transfers packet data to GPU as soon as the incoming data arrives, without having to wait or buffer all the application data. It also avoids having the CPU to copy data into a contiguous buffer, and utilizes asynchronous GPU-resident functions to initiate the data transfer. Thus, NetML reduces CPU consumption and minimizes latency and seeks to minimize the idle time on the GPU. NetML is effective in improving throughput and latency both for learning and inference, without requiring fundamental changes to the ML libraries. NetML can benefit every application processing streaming data on GPU. We have demonstrated that NetML reduces the latency to infer an image for object detection using a neural network by 20% and increased throughput of training a neural network with streaming data by more than 30%.

VII. Acknowledgement

This work was supported by the US NSF grant 1763929 and a grant from Hewlett Packard Enterprise Co.

References

[1] NVIDIA, "Nvidia deep learning platform," https://developer.nvidia.com/deep-learning, 2019.

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[3] T. NVIDIA, "P100 white paper," *NVIDIA Corporation*, 2016.

[4] A. Dhakal and K. Ramakrishnan, "Machine learning at the network edge for automated home intrusion monitoring," in *Workshop on Machine Learning and Artificial Intelligence in Computer Network, ICNP 2017.*

[5] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015.

[6] T. Kraska, A. Talwalkar, and J. Duchi, "Mlbase: A distributed machine-learning system," 2013.

[7] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 2017, pp. 150–156.

[8] S. Teerapittayanon, B. McDanel, and H. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *ICPR*. IEEE, 2016, pp. 2464–2469.

[9] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of mellanox/nvidia gpudirect over infinibanda new model for gpu to gpu communications," *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 267–273, 2011.

[10] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv:1804.02767 [cs.CV]*, 2018.

[11] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv:1602.02830 [cs.LG]*, 2016.

[12] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[13] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, 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.

[14] D. Intel, "Data plane development kit," https://dpdk.org/, 2014.

[15] C. Cuda, "Best practice guide, 2018," 2018.

[16] "Nvidia visual profiler user guide," https://docs.nvidia.com/pdf/CUDA_Profiler_Users_Guide.pdf.

[17] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[18] A. Paszke *et al.*, "Automatic differentiation in pytorch," 2017.

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[20] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4. ACM, 2010, pp. 195–206.

[21] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "Sslshader: Cheap ssl acceleration with commodity processors." in *NSDI*, 2011.

[22] W. Sun and R. Ricci, "Fast and flexible: parallel packet processing with gpus and click," in *ANCS*, 2013, pp. 25–35.

[23] X. Yi, J. Duan, and C. Wu, "Gpunfv: a gpu-accelerated nfv system," in *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 2017, pp. 85–91.

[24] M. Silberstein *et al.*, "Gpunet: Networking abstractions for gpu programs," vol. 34, no. 3. ACM, 2016, p. 9.

[25] J. Kim *et al.*, "Nba (network balancing act): a high-performance packet processing framework for heterogeneous processors," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 22.

[26] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-net: Effective gpu sharing in nfv systems," in *NSDI*, 2018.

[27] Y. Go, M. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: revitalizing gpu as packet processing accelerator," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2017, pp. 83–96.

[28] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 0.5 mb model size," *arXiv:1602.07360 [cs.CV]*, 2016.

[29] B. Wu, F. N. Iandola, P. H. Jin, and K. Keutzer, "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving." *arXiv:1612.01051 [cs.CV]*, 2016.

[30] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv:1510.00149 [cs.CV]*, 2015.

[31] NVIDIA, "Nvidia tesla v100 gpu architecture."

[32] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*. IEEE, 2017.

[33] M. Zaharia *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, 2016.

[34] "Apache storm," http://storm.apache.org/, accessed:2018-12-01.

[35] Z. Chen, J. Xu, J. Tang, K. Kwiat, and C. Kamhoua, "G-storm: Gpu-enabled high-throughput online data processing in storm," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015.

[36] J. Paul, J. He, and B. He, "Gpl: A gpu-based pipelined query processing engine," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1935–1950.

[37] J. Dean *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012.

[38] H. Kim, J. Park, J. Jang, and S. Yoon, "Deepspark: A spark-based distributed deep learning framework for commodity clusters," *arXiv:1602.08191 [cs.LG]*, 2016.