Title	GPUNFV: a GPU-Accelerated NFV System
Author(s)	Yi, X; Duan, J; Wu, C
Citation	The 1st Asia-Pacific Workshop on Networking (APNet'17), Hong Kong, 3-4 August 2017
Issued Date	2017
URL	http://hdl.handle.net/10722/243236
Rights	Proceedings of APNet'17. Copyright © Association for Computing Machinery.; ©ACM, YYYY. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in PUBLICATION, {VOL#, ISS#, (DATE)} http://doi.acm.org/10.1145/nnnnnn.nnnnnn; This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

GPUNFV: a GPU-Accelerated NFV System

Xiaodong Yi
The University of Hong Kong
xdyi@cs.hku.hk

Jingpu Duan
The University of Hong Kong
jpduan@cs.hku.hk

Chuan Wu
The University of Hong Kong
cwu@cs.hku.hk

ABSTRACT

This paper presents GPUNFV, a high-performance NFV system providing flow-level micro services for stateful service chains with Graphics Processing Unit (GPU) acceleration. GPUNFV exploits the massively-parallel processing power of GPU to maximize the throughput of the NFV system. Combined with the customized flow handler, GPUNFV achieves a much better throughput than the existing NFV systems. With a carefully designed GPU-based virtualized network function framework, GPUNFV is able to efficiently support both stateful and stateless network functions. We have implemented a number of GPU-based network functions and a preliminary GPUNFV system to demonstrate the flexibility and potential of our design.

CCS CONCEPTS

• **Networks** → **Network design principles**; **Layering**;

KEYWORDS

Service chain; NFV; Micro service; GPU

ACM Reference format:
Xiaodong Yi, Jingpu Duan, and Chuan Wu. 2017. GPUNFV: a GPU-Accelerated NFV System. In *Proceedings of APNet'17, Hong Kong, China, August 03-04, 2017,* 7 pages.
https://doi.org/10.1145/3106989.3106990

1 INTRODUCTION

The recent trend of Network Function Virtualization (NFV) has been a driving force for network operators to discard customized hardware middleboxes and run various network functions (NFs) as software instances on virtualized environments in commodity servers. NFV significantly reduces the cost to deploy network services and enables high flexibility when processing fluctuating network traffic.

In the existing NFV systems [6, 7, 15, 20], virtualized network functions (VNFs) are mostly running on traditional CPU cores. Performance of VNFs when running on CPU cores may not be ideal: for NFs that perform intensive computation on the received packets, the average packet processing time achieved using a single CPU core may be several microseconds, limiting the maximum throughput to a few million packets per second. To achieve line-rate packet processing (e.g., 10Gbps), the existing VNFs distribute the input traffic flows to multiple CPU cores, to exploit parallelism. However, with the recent deployment of 40Gbps network interface cards (NICs) in datacenters, the total number of CPU cores that are available on a commodity server may not provide a high enough parallelism level to support the 40Gbps packet processing rate.

GPU based packet processing acceleration has been studied in the literature [9, 13, 14, 21, 22], for building software routers or implementing NFs such as stateless intrusion detection systems (IDSs). A single GPU chip consists of thousands of cores, providing massive parallelism. By delegating packet processing to GPU parallel processing, a 40Gbps throughput can be achieved [9]. Unlike a CPU, a GPU has thousands of uniform execution units called GPU threads. All the GPU threads can execute the same piece of program simultaneously, but over different input data. The typical design of using GPU to accelerate packet processing is to let the CPU prepare a batch of packets and submit these packets, together with the NF program, to the GPU. The GPU runs the NF program on each GPU thread, processing one of the packets contained in the packet batch. Stateless NF processing can be well handled in this way since the order in which packets are processed does not matter.

On the other hand, a stateful NF maintains a flow state for each flow and the packets belonging to the same flow must be processed according to the order that they are received. If the packet batch submitted to the GPU contains several flow packets belonging to the same flow, without a good design, consistency of the flow state may be compromised due to concurrent processing of those packets by potentially several GPU threads.

We argue that the absence of good design for stateful NF service chain processing on GPUs is due to the lack of an appropriate software abstraction to support flow-level packet

APNet'17, August 03-04, 2017, Hong Kong, China

ACM ISBN 978-1-4503-5244-4/17/08…$15.00
https://doi.org/10.1145/3106989.3106990

processing. Instead of processing one NF in one GPU and then chaining GPUs up to fulfill service chain processing, we advocate deploying the entire service chain in one GPU. This can potentially reduce the number of GPUs we need, decrease the times for data preparing and transferring between a CPU and a GPU, and facilitate state maintenance for individual flows. To enable correct processing of flows through the entire service chain on a GPU, packets of the same flow should be processed through the chain of stateful NF programs on the same GPU thread. We refer to the per flow service chain processing as a flow-level micro service chain service, which involves initiation, storage of per-flow NF states and buffering of flow packets to construct a flow-level packet stream. With the flow-level micro services enabled, each GPU thread can easily access packets belonging to the same flow for processing, and update the corresponding flow states, ensuring consistency of the flow states at all times.

Based on this design philosophy, we propose a novel framework, GPUNFV, that efficiently supports stateful NF service chain processing on GPUs. GPUNFV provides micro service chain services to individual flows using the actor model [1]. For each flow that needs to traverse a service chain, an actor is constructed to process the packets in the flow. The actor buffers flow packets to construct an in-order packet stream for the flow, and stores flow states associated with NFs in the service chain. The packet processing of each flow actor is then delegated to a GPU thread. Multiple service chains are supported by using one GPU to run each service chain.

We further address the following design challenges in GPUNFV. *First*, to port network functions to GPU, because of the many limitations on programs running in GPUs, the existing NF software mostly needs to be completely rewritten. We create a GPU-based VNF framework to facilitate this time-consuming task. *Second*, with stateful NFs, we must guarantee that every GPU thread processes packets of the same flow in order when the respective GPU kernel is executed. To this end, we use a carefully designed batcher to prepare the packet batch and the flow state batch, implemented with page-lock memory allocated when the system is initialized. The positions of packets in a packet batch are well organized to ensure that each GPU thread processes the same flow when the GPU kernel is launched every time. *Third*, CPU processing and GPU processing need to be efficiently synchronized, as otherwise, the CPU may always be waiting for completion of GPU processing, leading to serious performance degradation. Especially, given the extra work of preparing data that GPU needs, the performance may be even worse than processing using the CPU alone. We design a strategy for dynamic sizing of a packet batch, for best synchronization between CPU and GPU processing.

Our preliminary implementation and evaluation of GPUNFV show that compared with CPU only packet processing, GPUNFV can achieve a throughput more than 2 times higher. In addition, with our dynamic sizing of the packet batch, the waiting time of CPU can be reduced to nearly 0, and the optimal size of packet batch can be achieved in seconds.

2 BACKGROUND AND RELATED WORK

Network Function Virtualization. Much work on NFV have been focusing on performance improvement of VNFs to match that of the dedicated hardware network functions, e.g., ClickOS [16] and NetVM [11]. Others design NFV management systems to boost scalability of NFV service chains, e.g., Stratos [6], E2 [20]. All these work use CPUs for packet processing and the maximum throughput achieved by CPU based packet processing may not be up to the line speed of 40Gbps, due to the maximum number of CPU cores that are available on a commodity server.

GPU and GPU based Packet Processing. A GPU [3] typically has several graphics processing clusters which consist of multiple streaming multiprocessors (SMs). Each SM consists of several stream processors (SPs). All threads running on all SPs share the same program named "kernel". An SM works as an independent SIMT (Single Instruction Multiple Threads) processor. The basic execution unit of an SM is a *warp*, a group of threads, which share the same instruction pointer. One SM consists of multiple warps. All threads running the same kernel in a warp should follow the same code path. The threads always fetch data at some specific positions in the device global memory. Those positions are always relevant to their thread identity numbers. For a NVIDIA Titan X Pascal GPU that we use in our evaluation, it has 6 graphics processing clusters and 30 SMs, each of which consists of 128 SPs, resulting in 3584 SPs in total.

There have been a few studies exploiting GPUs to accelerate packet processing. PacketShader [9] is a high-performance software router running on GPUs, powered by a new packet I/O engine. Snap [21] is another GPU based software router framework, integrated with the Click modular router [15]. Kargus [12] is a high-performance IDS and employs GPUs for pattern matching. These designs can be easily extended to enable GPU-accelerated stateless NFs, but do not support stateful NFs.

The Actor Programming Model [1, 2, 4] has been widely used to construct reliable distributed systems [17, 18]. It employs a basic execution unit, called *actor*, for message handling and passing. Each actor is configured with several message handlers and a mailbox. To communicate with another actor, the actor sends a message to the mailbox of the other actor. The received messages are processed with the respective message handlers. A scheduler is used to schedule different actors in an operating system, so that each actor runs

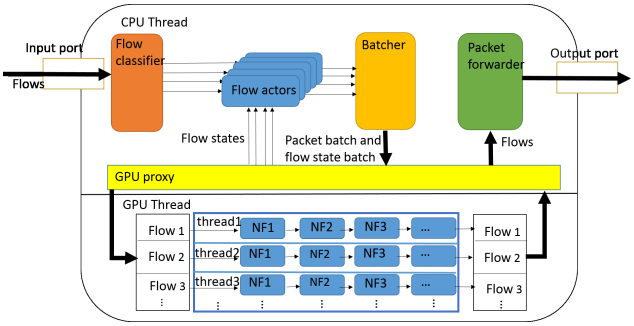

Figure 1: Architecture of GPUNFV

as if it ran in its own thread. We have identified that the actor programming model is naturally suitable to enable flow-level micro service chain service, and hence design our system on the actor model.

3 THE GPUNFV FRAMEWORK

The GPUNFV system consists of two threads: (i) the CPU management thread and (ii) the GPU computation thread. The two threads collaborate to accomplish packet processing jobs, which are periodically submitted by the CPU management thread to the GPU computation thread and executed by the GPU computation thread. On a modern GPU server, a GPU usually functions as an asynchronous device that communicates with CPU through the PCIe bus. Once a processing job is submitted to the GPU, the CPU needs to wait for a completion signal from the GPU before submitting the next job. Therefore, we need a CPU management thread for receiving/sending packets and submitting packet processing job to the GPU, and a GPU computation thread that handles the actual packet processing job. Fig. 1 illustrates the architecture of GPUNFV.

In the CPU management thread, data packets are actively polled from the input port using DPDK [5]. The received data packets are immediately classified into different flows according to the traditional flow 5-tuples (source/destination IP addresses, source/destination ports, application layer protocol type) using a *flow classifier*. Each flow is then delegated to a unique *flow actor* (to be discussed in Sec. 3.1), which is responsible for storing the flow states of NFs and temporarily buffering the packets for GPU processing of the flow. Later, the flow actors forward buffered packets and the current flow states to the *batcher*, which encapsulates them in a packet processing job and further submits the job to the GPU through the *GPU proxy*. The data is exchanged through a page-lock memory (Sec. 3.2).

The GPU computation thread carries out stateful NF service chain processing for each flow of the submitted job on a dedicated GPU thread. When the packet processing job is finished, a completion notification of GPU is delivered to the GPU proxy, together with updated flow states. The GPU proxy then passes the updated flow states to respective flow actors and replaces the out-dated flow states in the flow packet batch with the corresponding updated flow states (see Sec. 3.3). All the processed packets are sent out by the *packet forwarder* upon each completion notification.

3.1 Flow Actor

Based on the actor model, we create one unique actor for each flow, i.e., the flow actor. A flow actor is created upon the arrival of the first packet of a flow and destroyed when the flow finishes. Upon creation, a flow actor initiates flow states associated with each NF on the service chain that the flow is to traverse, and stores the flow states. The flow actor does not process the flow packets, but only buffers them and submits them to the GPU through the batcher. Especially, when a flow actor receives a packet, it enqueues the packet in a queue. After the CPU management thread has polled the input port for several times (the current batch size of the batcher divided by the maximum number of packets that are polled from the input port), the batcher will retrieve enqueued packets from the flow actors, together with the flow states, to construct the next packet processing job. The flow actor communicates with the batcher and GPU proxy through customized message handlers.

In our prototype implementation of GPUNFV, we create a tailored actor library to implement flow actors, which provides basic message passing functionalities for flow actors to communicate with other modules in GPUNFV. We do not use the existing actor frameworks (e.g., [2, 4]) since they are not optimized for NFV systems. In addition, we aggressively remove the mailbox of each actor and implement message passing as direct function calls. This is possible because the CPU management thread does not share data with other threads and is contention-free.This optimization decreases overhead associated with message passing and greatly speeds up packet handling in GPUNFV.

3.2 Page-lock Memory

Since a GPU has its own memory and programs running on GPUs can not directly access the CPU memory, an efficient strategy for transferring data between the CPU memory and the GPU memory is important for correct data processing with GPUs. GPUNFV utilizes the zero-copy strategy provided by CUDA [19], which eliminates the need to copy data between the GPU memory and the CPU memory before launching the GPU kernel code.

Upon initialization, GPUNFV uses the cudaHostAlloc function [19] to allocate two sets of page-lock CPU memory pages

(a special kind of CPU memory pages that prevent themselves from being swapped to the disk) for each GPU managed by GPUNFV, to store packet batches and flow states, respectively. A special flag is set in the cudaHostAlloc function to map the allocated page-lock memory pages directly into a GPU device so that the GPU can directly access the mapped page-lock CPU memory pages as if they were GPU memory pages. The page-lock memory pages are used for data transfer between the batcher and the GPU proxy.

3.3 Batcher

In a GPU, one thread always fetches the data in the specific position of the device memory (mapped page-lock memory in our design), which is related to the thread's identity number [19]. To make full use of the parallel processing capabilities of a GPU and ensure that each GPU thread processes packets of the same flow in order, we carefully design a packet batching strategy. The batcher traverses the list of all flow actors whose queues have unprocessed packets, fetches the packets from the queues until a batch size is reached or a queue is empty, and places those packets into one batch. The positions of packets from the same flow in the batch follow a rule: if the position of the first packet from flow A is a, then the position of the second packet from the flow is $a + m$, the third packet is $a + 2 * m$, and so on. Here m is the number of flows whose packets appear in the batch. When the rates of flows are different, the batcher leaves some positions empty without a packet.

When the batch reaches a specific size (we design a dynamic sizing strategy in Sec. 3.5), the batcher copies the retrieved batch of packets and flow states of each flow to the page-lock CPU memory pages for flow packet batch and for flow states, respectively, which can then be directly accessed by the GPU. Since the positions of consecutive packets from the same flow are always apart by a constant m in the packet batch, a GPU thread can easily fetch the packets from the same flow easily, i.e., at a constant step of m. In this way, GPUNFV ensures that one GPU thread only processes one specific flow in each processing job.

3.4 GPU Proxy

When the batcher has finished filling the flow packet batch and flow states in the page-lock memory, the GPU proxy takes control. The GPU proxy checks whether the GPU has completed the previous processing job. If not, the GPU thread blocks until the GPU processing job is finished. When a GPU processing job is done, the GPU proxy sends the processed packets to the packet forwarder for output. If the states of a flow have been updated in the job, the GPU proxy checks whether there are already flow states of the flow in the page-lock memory pages which are out-dated. If so, it replaces the old flow states in the memory by the updated ones. Then it pushes all the updated flow states back to the corresponding flow actor. The GPU proxy then notifies the GPU about the positions of the page-lock memory pages storing the flow packet batch and flow states (this is a compulsory step in CUDA [19]), before launching the GPU kernel code to start the next GPU processing job. The GPU kernel code contains core processing logic of VNFs on the service chain (Sec. 3.6).

3.5 Dynamic Sizing of Packet Batch

After the CPU management thread has prepared the next flow packet batch for processing in the page-lock memory, the GPU processing thread may block if the previous GPU processing job is not done yet. If the GPU thread blocking lasts for long, incoming flow packets to the input port may be dropped, compromising system throughput. We design a dynamic batch sizing strategy to address this problem: we dynamically adjust the size of the packet batch prepared by the batcher in each round, such that the time the CPU management thread takes for preparing the batch, to be processed by the next GPU processing job, is slightly longer than the GPU processing time of the current job. In this way, the GPU thread will not block, but will fetch the batch for the next job immediately when the batch is ready in the page-lock memory.

To identify such an optimal batch size, we initialize the size of the packet batch to a small value (320 is used in our implementation). Each time when the GPU thread takes control and blocks, the blocking time is recorded. If the blocking time is larger than a threshold (0.1ms is used in our implementation), the size of the packet batch is increased by a small value (320 in our implementation).

3.6 GPUNFV API

Since we use the actor model to provide per-flow micro service chain services and process packets on GPUs, the NFs running in GPUNFV must be programmed according to the per-flow abstraction and the runtime environment of GPU. To facilitate programmers with this task, we provide a number of new APIs in GPUNFV, to easy developer's task in extracting and storing flow states in each flow actor, and for constructing the GPU kernel code. Especially, the core NF logic can be implemented in the function __device__ nf.process_pkt(input_pkt, fs), which takes as input a packet and the current flow state, and processes the input packet using the current flow state. This is a CUDA function, invoked inside a GPU.

The GPU kernel code runs uniformly on each GPU thread. It uses the identifier of the GPU thread to retrieve flow packets and flow states of a flow from the batches in the page-lock memory, and then sequentially calls __device__ nf. process_pkt(input_pkt, fs) for each NF along the service chain, for stateful NF service chain processing of the packets. Those

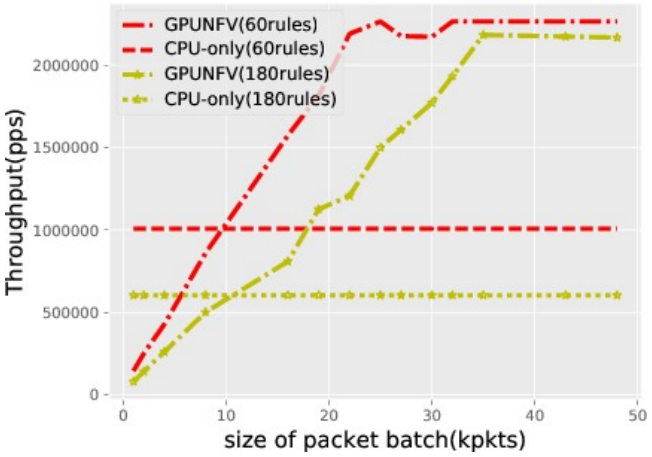

Figure 2: Throughput: GPUNFV vs. CPU-only processing

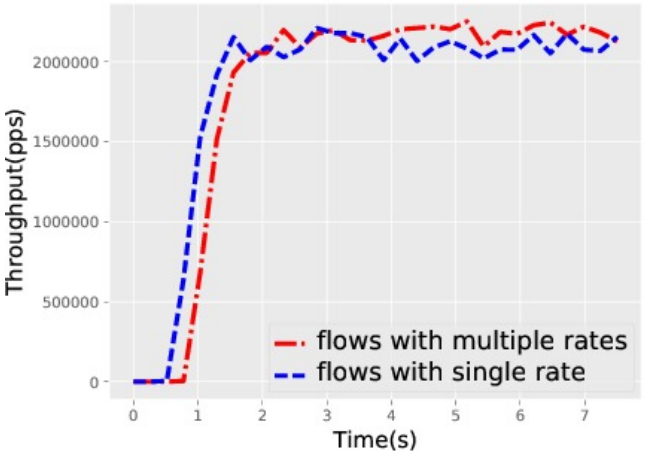

Figure 3: Throughput of GPUNFV with multiple flow rates

threads are automatically allocated to SPs belonging to different SMs by CUDA [19].

4 EXPERIMENTS

We implement GPUNFV in a runtime which is a high-performance user-space program enabled to use both DPDK and NVIDIA GPU. We have implemented a number of NFs, including firewall (410 LOC), load balancer (63 LOC), and flow monitor (159 LOC). The firewall maintains a large number of rules such as blocking a certain source IP address, and checks each received packet against the rules: if the packet violates any of the rules, a tag in the flow state is flipped and later packets are automatically dropped. The firewall also records the TCP connection status of the flow in the flow state. The flow monitor updates an internal counter whenever it receives a packet. The load balancer allocates the flows to different destinations and stores those destinations as flow states. Our following experiments are carried out in a Mac Pro server equipped with two 2.4GHz 6-Core Intel Xeon E5645 processors and one NVIDIA Titan X Pascal GPU.

4.1 Packet Processing Throughput

In this set of experiments, we use the flow generator module in BESS [8] to generate 50000 flows at the rate about 3Mpps(packets per second). We deploy one service chain "flow monitor(FB)->firewall(FW)->load balancer(LB)" to process these flows. The size of the packet batch is fixed to different values, i.e., without dynamic sizing.

In Fig. 2, we compare GPUNFV with CPU-only processing, i.e., flows processed by the service chain implemented in each flow actor, instead of being sent to the GPU for processing. We vary the fixed batch size ('kpkts' in Fig. 2 stands for '10^3 packets'), and observe that when the size of packet batch is small, GPUNFV does not perform as well as CPU-only processing, but picks up quickly when the batch size increases. This is because when the batch is very small, GPU's powerful parallel processing capabilities are poorly utilized, as the CPU thread always needs to wait for GPU processing to complete. We also vary the number of rules enabled in the firewall, to change processing complexity of the service chain. Fig. 2 shows that when the processing logic is more complicated, the advantage of GPUNFV becomes more obvious. The curves of CPU-only processing are flat, since there is no packet batching nor copying to page-lock memory in such scenarios.

We further compare the throughput when the packet rates of flows are all around 3Mpps and when they differ within the range of 20-100 pps. Fig. 3 shows that the performance in these two cases is nearly the same, where the batch size is fixed to 40k packets.

4.2 Processing Time

We next evaluate the time taken by the GPU thread, by the CPU thread for packet handling and by the CPU thread due to waiting for GPU processing completion, recorded during a 20-second run of the system (roughly 1000-2000 processing jobs are completed), and averaged over multiple runs. We deploy the service chain "FM->FW(180rules)->LB" in this set of experiments.

Fig. 4 shows that when the batch size is small, the CPU processing time is relatively small; even though the GPU thread is working all the time, a large amount of CPU time is wasted by waiting for the GPU thread. With the increase of batch size, the waiting time of the CPU thread drops and CPU processing time increases.

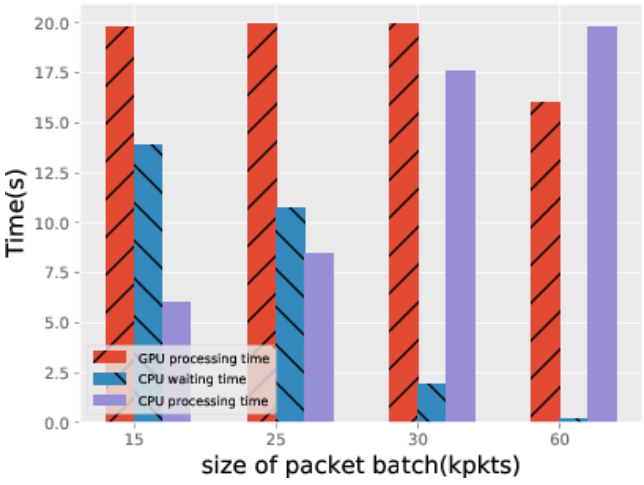

Figure 4: Time used by CPU and GPU threads

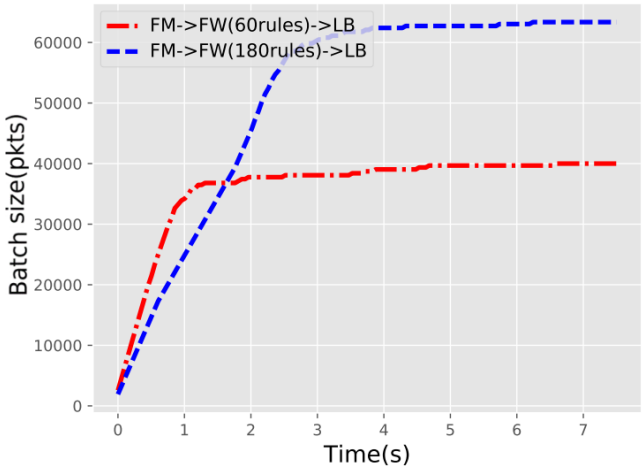

Figure 5: Batch sizes with dynamic batch sizing

4.3 Dynamic sizing of packet batch

We next evaluate the effectiveness of our dynamic batch sizing strategy in Sec. 3.5. We deploy service chains "FM->FW(60rules)->LB" on one runtime and service chain "FM->FW(180rules)->LB" on another and generate 50000 flows at the total rate of 400 Mpps. The initial size of the packet batch is 320 packets. Fig. 5 shows that the batch size increases linearly at first and then becomes stable. The size is relevant to the service chain: it takes longer to stabilize and the optimal size is larger if the service chain is more complicated. This is because the GPU processing time increases when the service chain is more complicated and a larger batch size is used for minimizing the wasted CPU waiting time. In Fig. 6, the throughput becomes relatively stable when the batch size becomes optimal.

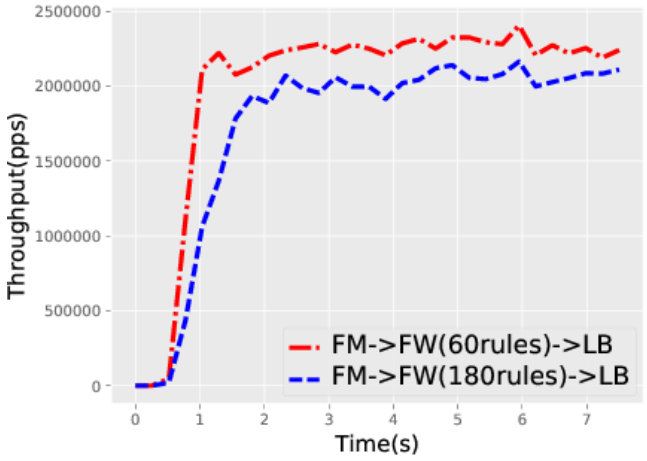

Figure 6: Throughput with dynamic batch sizing

5 CONCLUDING DISCUSSIONS

GPUNFV is a GPU-based NFV system which provides flow-level micro service for stateful service chain processing with GPU acceleration. It achieves high packet processing throughput by maximally exploiting parallel processing capabilities of GPUs. With the one-flow one-actor one-GPU thread based design, GPUNFV is able to easily maintain flow states for every single flow and enable stateful network functions to run on a GPU in parallel. We design strategies such as dynamic sizing of the packet batch and page-lock memory management, to achieve higher performance of the whole system.

Our current GPU-based NFV framework works best for stateful computation-intensive NFs with few branches in the processing logic. If there are many branches, the GPU parallel processing capabilities will be significantly degraded. In the future work, we plan to build a new framework which can convert NFs with many branches into ones with little branches while ensuring the correct logic [10] to improve generality of GPUNFV.

When evaluating our prototype system implementation, we notice that longer delay may be incurred by GPU based packet processing, as compared to CPU-only processing, when the traffic rate is low, since the batcher and the GPU proxy in the framework introduce additional delays. We plan to address the issue by using CPU-only processing for low latency under low traffic rates and exploiting GPU acceleration for high throughput under high traffic rates.

Because of the hardware limitation, we carry out evaluation using one server with one GPU. Our system can actually be readily running on a multi-server multi-GPU environment. We will carry out more extensive evaluation in a cluster of GPU servers in the near future.

6 ACKNOWLEDGEMENT

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17204715, 17225516, C7036-15G (CRF), a grant NSFC 61628209 and the HKU matching fund. The Titan X Pascal used for this research was donated by the NVIDIA Corporation.

REFERENCES

[1] 2010. Actor Modle. https://en.wikipedia.org/wiki/Actor_model. (2010).
[2] 2010. Erlang. https://www.erlang.org/. (2010).
[3] 2010. GEFORCE GTX 1080. https://www.nvidia.com/. (2010).
[4] 2010. Scala Akka. akka.io/. (2010).
[5] 2015. Intel Data Plane Development Kit. http://dpdk.org/. (2015).
[6] Aaron Gember, Robert Grandl, Ashok Anand, Theophilus Benson, and Aditya Akella. 2012. Stratos: Virtual middleboxes as first-class entities. *UW-Madison TR1771* (2012), 12.
[7] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2015. OpenNF: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 163–174.
[8] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A software NIC to augment hardware. *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155* (2015).
[9] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, Vol. 40. ACM, 195–206.
[10] Tianyi David Han and Tarek S Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 3.
[11] Jinho Hwang, KK Ramakrishnan, and Timothy Wood. 2015. NetVM: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management* 12, 1 (2015), 34–47.
[12] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. 2012. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 317–328.
[13] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G Andersen. 2015. Raising the Bar for Using GPUs in Software Packet Processing.. In *NSDI*. 409–423.
[14] Kang Kang and Yangdong Steve Deng. 2011. Scalable packet classification via GPU metaprogramming. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 1–4.
[15] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.
[16] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 459–473.
[17] Sanjeev Mohindra, Daniel Hook, Andrew Prout, Ai-Hoa Sanh, An Tran, and Charles Yee. 2013. Big Data Analysis using Distributed Actors Framework. In *Proc. of the 2013 IEEE High Performance Extreme Computing Conference (HPEC)*.
[18] Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. 2016. Optimizing Distributed Actor Systems for Dynamic Interactive Services. In *Proc. of the Eleventh European Conference on Computer Systems (EuroSys'16)*.
[19] CUDA Nvidia. 2011. C programming guide version 4.0. *Nvidia Corporation* (2011).
[20] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: a framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 121–136.
[21] Weibin Sun and Robert Ricci. 2013. Fast and flexible: Parallel packet processing with GPUs and Click. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*. IEEE Press, 25–36.
[22] Janet Tseng, Ren Wang, James Tsai, Saikrishna Edupuganti, Alexander W Min, Shinae Woo, Stephen Junkins, and Tsung-Yuan Charlie Tai. 2016. Exploiting integrated GPUs for network packet processing workloads. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE, 161–165.