Towards a GPU SDN Controller

Eduard G. Renart Eddy Z. Zhang Badri Nath
Department of Computer Science
Rutgers University
New Brunswick, New Jersey
egr33@scarletmail.rutgers.edu, {eddy.zhengzhang, badri}@cs.rutgers.edu

***Abstract*—The SDN concept of separating and centralizing the control plane from the data plane has provided more flexibility and programmability to the deployment of the networks. On the other hand, the separation of the planes has raised some scalability and performance questions, being that the SDN controller is the bottleneck. In this paper we present an implementation of a GPU SDN controller. The goal of this paper is to mitigate the scalability problem of the SDN controller by offloading all the packet inspection and creation to the GPU. Experimental evaluation shows that the controller is able to process 17 Million packets/s using off-the-shelf GPU's.**

I. Introduction

New emerging technologies such as software-defined networking are an attempt to solve the traditional switching/routing deployment bottleneck. Software-defined networking is based on three simple principals:

- Separation of the control plane from the data plane
- A centralized controller and a centralized view of the network
- Programmability of the network

SDN raises a certain number of questions regarding the scalability of the control plane. The two most popular concerns are (1) how fast can the controller respond to data plane requests?; and (2) how many data plane requests can it handle per second? [13]

These two questions have led to extensive research on different types of SDN controller architectures such as: Elastic distributed SDN controller [12], Beacon OpenFlow controller [11], Nox [3] and Pox OpenFlow controller [4] etc. But the problem that all these architectures present is that all of them scale horizontally, making it difficult to add more nodes to the network due to synchronization overhead. Vertical scaling is another design that needs to be explored [21]. Even though, the throughput requirements of a control plane are less than a data plane, scalability of the controller as a function of the number of switches connected to it needs evaluation. We wanted to created a new SDN controller architecture to be able to scale the controller vertically and make it possible by using commercial off-the-shelf (COTS) hardware, one of which is the available GPUs.

Extensive prior work exists that shows that the performance of many network workloads can be improved using the GPU, such as pattern matching, network coding, IP table lookup and cryptography. Until now, no research has focused on SDN controller workloads and the issues of using off-the-shelf GPUs in a SDN controller.

978-1-4799-5804-7/15/$31.00

This paper is organized as follows: section II presents the difference between software routers vs SDN controllers, section III describes the GPU implementation, section IV presents all the potential GPU improvements, and finally section V presents the conclusions.

II. Software Router VS SDN Controller

Inspite of all the advances in technology, there is an ever increasing demand on the packet processing rate due the increase in number of hosts and the number of sessions in the Internet. Such high packet transmission rates require fast and more reliable packet processing. For this reason, the architecture of a new generation of routers seems to be changing; now the trend is to implement software routers using general purpose hardware and still overcome the route lookup bottleneck problem. In this section, we compare and contrast the similarities and differences between a software router and an SDN controller. We show that a SDN controller is different from a traditional software router.

A. Software Router:

Software routers are responsible for performing IP lookups and packet forwarding. The role of the router is to receive the packets arriving at all the ports that the router has and process them by either using the CPU, GPU or both, and finally, to transfer each incoming packet from its input port to the desired output port which is typically determined by processing the packet's IP headers.

B. SDN Controller:

The role of a Controller in a software-defined network (SDN) is to be the "brain of the network and be responsible for making decisions for every traffic flow at all layers of the protocol stack. Typically, if a rule for packet forwarding exists in the forwarding plane, packet forwarding on the required interface happens without the need to involve the controller figure 1. However, if there is a miss in the flow table, the forwarding plane generates a packet_in [1] event and it is sent to the controller. If the switch has sufficient memory to buffer all packets that are being sent to the controller, the packet-in event contains some fraction of the packet header (by default up to 128 bytes) and a buffer ID to be used by the controller when it is ready for the switch to forward the packet. Switches that do not support internal buffering (or have

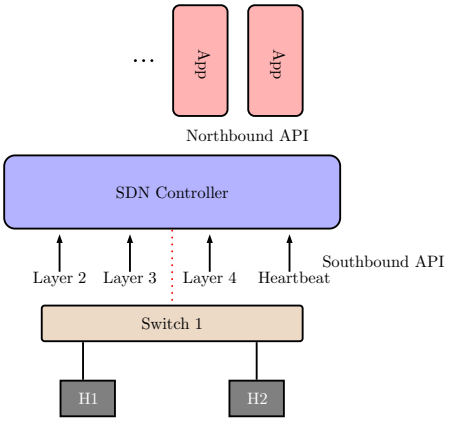

Fig. 1. Representation of an SDN architecture.

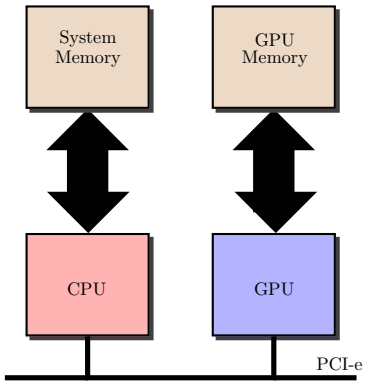

Fig. 2. Simulation of the CPU-GPU architecture.

run out of internal buffering) must send the full packet to the controller as part of the event. Then the controller, according to policy, processes the packet and sends a flow modification packet or a packet_out. This process occurs for packets at any layer. For example, consider Layer 2 ARP packets. Here a host sends an ARP Request to the switch which results in an ARP request miss in the flow table of the switch. This causes the switch to generate and send a OFP+ARP Packet_in (OpenFlow Packet) message to the controller with a size of 60B; 42B for the ARP frame and 18B for the OpenFlow frame. The controller then replies with a Broadcast Packet-Out message of size 24B. Since we want to implement a stand-alone GPU Controller where we only use the CPU to send and receive the networking packets from the nic card and we want to take into account that our controller will also work when switches run out of internal buffering, we need to consider offloading the entire packet_in event to the GPU. This will require us to offload heterogeneous workloads (in packet_in event sizes) to a GPU as opposed to offloading homogeneous workloads to a GPU, as in GPU Software Routers [7][8][9]. In GPU software routers, the primary task is an IP table lookup for all the incoming packets, which requires uploading 4 Bytes for every incoming packet to the GPU resulting in a very homogeneous and efficient memory transfer from the CPU to the GPU and vice versa. On the other hand, in a GPU SDN Controller the heterogeneous workload provides challenges to vertical scaling. In this paper, we explore GPU packet_in events offloading issues as they relate to a GPU-based SDN controller.

III. Basic GPU Implementation

We have implemented a mac learning switch GPU Controller using OpenFlow 1.0 from scratch, which offloads the entire packet_in event to the GPU, inspects the packet and generates a new packet_out event in the GPU, in which no GPU-related optimization is applied. In order to evaluate the performance of our GPU SDN Controller we have implemented a packet generator that simulates the behavior of the controller benchmarker, Cbench, which can produce different types of packet_in events in different granularity allowing us to test our controller in different situation. Our implementation of the controller benchmarker has allowed us to fully utilize the CPU and the GPU and understand what types of packet_in events are more suitable to be computed by the GPU. By doing this we are able to identify the strengths and weaknesses of our controller.

A. Workload Heterogeneity:

As explained in section II, the workload submitted to a GPU SDN controller is heterogeneous in terms of both packet_in size events and computational decisions. In this paper we will limit the discussion to packet_in size heterogeneity. We aim to deal with two main issues that arise due to heterogeneity: (1) the mapping of each variable size packet to each underlying GPU core, (2) the understanding of the performance bottleneck of the heterogeneous workloads.

B. Hardware Configuration:

For this research we used two different hardware configurations.

The first configuration is equipped with one Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 8 GB memory, and one NVIDIA GTX 680 (1536 cores, 4GB RAM). The machine has installed Suse Linux 12.1.

The second configuration is equipped with one Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 8 GB memory, and one NVIDIA Tesla K40c (2880 cores, 12GB RAM). The machine has installed Suse Linux 12.1.

C. Controller Structure:

The GPU Controller is a software framework that consists of code running on both CPU and GPU. Figure 3 illustrates the workflow of the GPU controller. The controller divides the CPU threads in two: the producer threads and the consumer threads. The producer threads are responsible for checking if the switches have sent data. If so, the producer threads store them in a memory pool that will be transferred to GPU for further processing. The consumer threads are responsible for sending the new packet_out or flow_mod events that the GPU

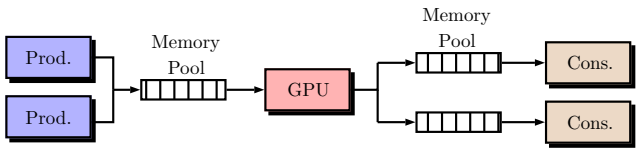

Fig. 3. GPU SDN Controller workflow.

has produced back to the switches. The thread assignment is the following: four consumer threads and four producer threads. Initially all eight threads are launched, but not all of them are active at every point in time. The number of threads that are active depends on the number of switches connected at that time. Once the memory pool reaches its maximum capacity or a sufficient amount of time has passed since the last GPU call, the memory pool is sorted by packet size and transferred from CPU memory to GPU memory. The GPU kernel is invoked to process all packets in parallel. The GPU outputs all the new packet_out or flow_mod and stores it into multiple output memory pools which only the consumer threads have access to. Finally, the consumer threads send all the processed packets back to the switches.

D. Workflow:

We divide the workflow of the GPU Controller into three stages: Producer-CPU, Inspection-GPU, Consumer-CPU.

- **Producer-CPU:** Each producer fetches the incoming packet_in events and stores them into the processing memory pool for later inspection by the GPU.
- **Inspection-GPU:** The producer memory pool is sorted (by packet_in size) and transferred from host CPU memory to GPU memory. The GPU kernel is launched to process the packets, and the output is saved into multiple output memory pools which can be accessed by the consumer threads.
- **Consumer-CPU:** Each consumer thread reads their own memory pool and sends the processed packets back to their respective clients.

E. Baseline Implementation:

For the evaluation of the workload performance of our GPU SDN Controller, we implemented two identical mac learning switch applications. The first one is a GPU-only application and the second one is a CPU-only application to evaluate the benefits of using GPU for processing networking workloads. Figure 4 depicts the performance of the GPU SDN Controller for three distinct hardware architectures: two GPUs and one CPU. For the CPU implementation, we let every iteration be mapped to one packet. Loop iterations are divided evenly among CPU threads. We measured the performance of the eight-core CPU and the two distinct GPUs using 82B and 62B packet_in events as the input and producing a flow_mod

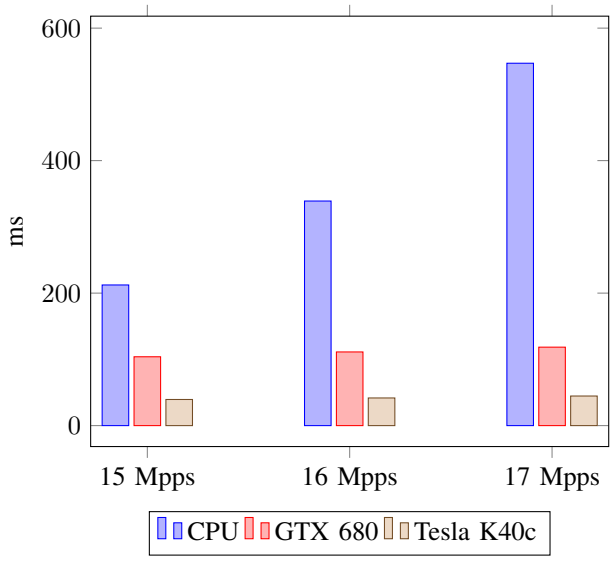

Fig. 4. CPU processing time vs GPU processing time using 82B and 62B packet_in event sizes.

event as the output. To mitigate the flow divergence between different packet_in events we simply sort them by size and group them in groups of 32 packets before they are uploaded to the GPU, by doing that we reduce the thread divergence and increasing the processing power of the GPU. The biggest trade off between different packet_in sizes is in the transfer time as explained in the following section. Figure 4 shows that Tesla K40c overperforms the GTX 680 by at least a factor of 2.6 and the CPU by at least a factor of 5.4 . We found that our naive kernel implementation of a GPU SDN controller significantly outperforms a regular CPU SDN controller by at least 5.4.

F. Transfer time:

For the evaluation of the CPU to GPU and GPU to CPU memory transfer time we programmed our controller benchmarker to generate two different types of packet_in events with different granularity. Figure 6 illustrates the time to transfer 17 Million packets (all in one batch) from the CPU memory to the GPU memory (as illustrated in figure 2). For all of the tests we used a heterogeneous mix of 62B and 82B packets. We find that by having packet_in size heterogeneity, we can improve the transfer time of our controller because if more small size packet_in events hit our controller, the transfer time from the CPU to GPU and vice versa is going to be smaller. This allows us to transfer even more packets and still achieve resolvable latency. Even when there is a 50%-50% 82B/62B heterogeneity, the transfer time outperforms the 100% 82B by a factor of 1.08. In the best case where we have 90%-10% 62B/82B heterogeneity, the transfer time improves by a factor of 1.17. The performance improvement factor increases as the total number of packets increases. With this test we have proven that the GPU can handle large packet_in OpenFlow batches even in the worst case where all the packets are big

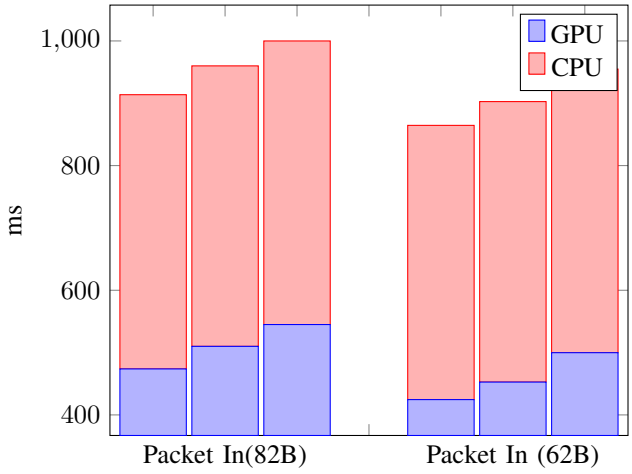

Fig. 5. CPU and GPU processing time.

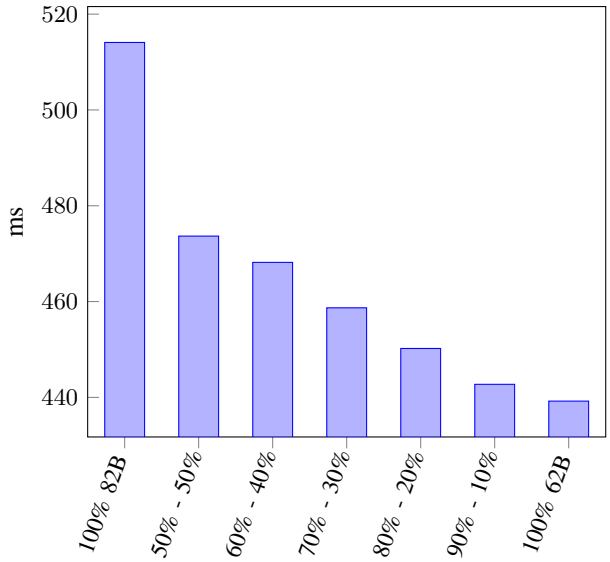

Fig. 6. Packet size heterogeneity: using two different sizes, 62B corresponds to the bottom percentage and 82B corresponds to the top percentage.

(82B), but we believe that our GPU SDN implementation will improve the performance when working in conjunction with existing CPU SDN Controllers as explained in future sections.

G. Bottleneck Analysis:

After thoroughly analyzing our application performance we found one major bottlenecks:

The first bottleneck is in the transfer of large packet_in event batches from the CPU memory to GPU memory. Batch processing of numerous packets significantly lowers per-packet processing overhead. However, it increases the latency of every single packet. By transferring large batches of packets from the CPU memory to GPU memory, the PCI-e and the CPU memory bandwidth then become major bottlenecks. Our research machines allow us to transfer data from the CPU to the GPU at a maximum rate of 48 Gb/s, and transfer from GPU to the CPU at a maximum rate of 44 Gb/s. If we assume that we have a homogeneous workload of 62B per packet and we want to process 17 million packets simultaneously, we end up needing to transfer 8.43Gb from the CPU to the GPU. It costs around 175ms to transfer from CPU memory to GPU memory as illustrated in figure 2, and another 260ms from GPU memory to CPU memory. In total, it ends up with a transfer time of around 435ms, around 4 times slower than the kernel execution.

The second major bottleneck we encountered is that the host TCP/IP stack has high CPU requirements. Some research has proven to experience up to 80% of CPU overhead associated with the TCP/IP implementation [20]. We believe that by making use of RDMA technology, TSO engines or adding more CPU power to overcome this problem will allow us to send and receive more packet/s. Figure 5 shows the GPU + CPU performance of the controller processing 15,16,17 Million packets/s. As can be seen from figure 5, the CPU in this situation is the bottleneck because the CPU needs to perform two times the total amount of packets as the GPU processes (it needs to send and receive the packets) while the CPU's computation horsepower is less than that of the GPU. However, even with two major performance bottlenecks, our controller yields 20% to 30% of improvement with one GPU when compared with the most popular controllers such as Nox [3], Pox [4], OpenDayLight [5], etc. We expect to see an improvement around 45% to 60% by simply adding one extra GPU and a better CPU to the current configuration.

IV. Potential Improvements:

A. Memory Overlap:

To be able to process the OpenFlow packets in the GPU, we need to transfer the data from the host (CPU) memory to the device (GPU) memory, process the packets on the device and transfer the results back to the host. If we perform this in a purely sequential manner, it leads to time periods where either the GPU or the CPU are inactive and results in loss of computation power. Instead, we can overlap three stages in a pipelined fashion. To perform more efficient computation/memory-transfer overlapping, we can make use of CUDA streams [17]. CUDA Streams are virtual queues on the GPU side that allow the GPU to execute a sequence of operations asynchronously with the CPU. We believe that by efficiently using streams in our GPU SDN Controller we will be able to overlap the data transfer time with the host/device computation time and further improve the performance of our controller.

B. Heterogeneous SDN Controller:

Our transfer times measurement in subsection F shows that transferring large packet_in events to the GPU and back to the CPU translates to higher latency in the packet processing. This issue can be addressed in multiple ways: first, we believe that the concept of a heterogeneous SDN Controller could be the solution that will improve the performance of current SDN Controllers. In a heterogeneous SDN Controller, we will

make use of existing CPU SDN implementations to process high-priority packet_in events and offload the packet_in events with lower priority to our stand-alone GPU Controller. The second option would be simply using the stand-alone GPU implementation to process packet_in events only when the inbound queue of the CPU controller reaches 80% to 90% of its total capacity, allowing it to reduce the buffer pressure on the inbound queues of the controller. The last option would be, instead of uploading the entire packet to the GPU, to only upload the required fields such as IP address or Mac address, this will reduce the overall size that needs to be transferred and allow the total amount of packets the can be processed by the GPU to increase tremendously. In addition, the use of the GPU in an SDN controller is not limited only for processing layer 2 or 3 packets, but it can also be used for traffic engineering, shortest path algorithm or any networking algorithm.

V. Conclusion

We have presented our stand-alone GPU Controller, an innovative framework for high performance SDN controller using commodity hardware that is able to scale vertically instead of horizontally like all the SDN controllers that exist nowadays. We maximized the number of packets/s that a regular controller can handle. In addition we also solved the problem that an SDN controller presents when the network scales and becomes bigger in terms of hosts and switches, a centralized controller requires bigger, and more costly systems with more powerful CPUs, RAM etc... Bottom line: the centralized approach can become expensive. By using GPUs as a processing engine and as a scalability element we can keep our centralized point of view and lower the CAPEX and OPEX because decent GPUs are less expensive, while every x86 server can cost similar or more money and achieve lower throughput compared to a GPU. We believe that the solution of our SDN approach keeps the centralized view, while providing vertical performance scalability. The key to maximal efficiency is the synergistic use of different components in a heterogeneous systems and how these components communicate with each other.

Acknowledgment

We would like to thank everyone who made this research possible and supported our work.

References

[1] OpenFlow Switch Specification, Version 1.0.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf
[2] OpenFlow Reference System. http://archive.openflow.org/wp/downloads/
[3] Nox SDN Controller. http://www.noxrepo.org/nox/about-nox/
[4] Pox SDN Controller. http://www.noxrepo.org/pox/about-pox/
[5] OpenDayLight SDN Controller. http://www.opendaylight.org/
[6] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chung, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. *RouteBricks: exploting parallelism to scale software routers*. In SOSP, 2009.
[7] S. Han, K. Jang, S. Moon and K. Park. *SSLShader: Cheap SSL Acceleration with Comodity Processors*. In NSDI, 2011.
[8] S. Han, K. Jang, K. Park and S. Moon. *PackeShader: A GPU-Accelerated Software Router*. In SIGCOMM, 2009.
[9] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. *Wire speed name lookup : a GPU based approach*. In NSDI, 2013.
[10] T. Benson, A. Akella and D.A. Maltz. *Network Traffic Characteristics of Data Centers in the Wild*. In IMC, 2010.
[11] D. Erickson. *The Beacon OpenFLow Controller*. In HotSDN, 2013.
[12] A. Dixit, F. Hao, S. Mukherjee, T.V. Lakshman and R. Kompella. *Towards and Elastic Distributed SDN Controller*. In HotSDN, 2013.
[13] A. Tootoonchian, S. Gorbunov and Y. Ganjali.*On Controller Performance in Software-Defined Networks*. In Hot-ICE, 2012.
[14] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam and C. Estan.*Evaluating GPUs for Network Packet Signature Matching*. In ISPASS, 2009.
[15] NVIDIA Corporation. NVIDIA CUDA Best Practices Guide, Version 3.0.
[16] NVIDIA Corporation. NVIDIA CUDA Programming Guide, Version 3.0 2009.
[17] CUDA C/C++ Streams and Concurrency. http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf
[18] NVIDIA GeForce GTX 680 Architecture http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680
[19] NVIDIA Tesla K40 Architecture http://www.nvidia.com/object/tesla-servers.html
[20] Pavan Balaji, Hemal V. Shah and D. K. Panda. *Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck.* In RAIT workshop 04.
[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. *The Google file system.* In SOSP '03. ACM, New York, NY, USA.