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 flows/s in the worst case scenario using just 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. We wanted to created a new SDN controller architecture to be able to scale the controller vertically and make it possible by using off-the-shelf 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.

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] that is sent to the controller. 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 (Open FLow 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. Next, consider Layer 3 ICMP packets such as ping. Here a host can send an ICMP echo reply that is received by the switch which results in a 70B frame + 18B Openflow frame

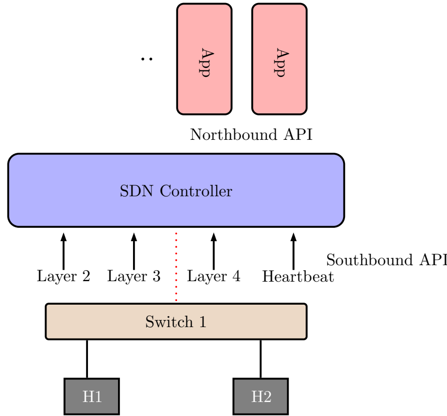

Fig. 1. Representation of an SDN architecture.

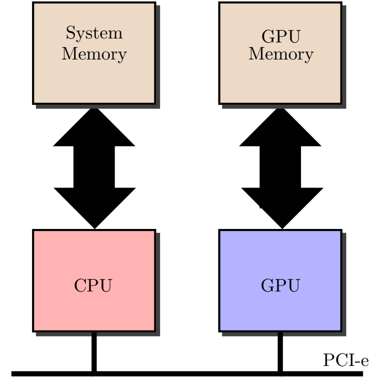

Fig. 2. Simulation of the CPU-GPU architecture.

sent to the controller. At the next level, Layer 4 packets such as TCP and UDP have multiple sizes for the Packet_in. For example, a TCP packet can have a TCP frame of 74B + 18B of OpenFlow frame. In addition, we should also consider heart beat messages such as the Echo Request and Echo Replay that Openflow implements with a size of 8B. Openflow architecture results in packet size heterogeneity with the messages from the switch to the controller (Packet_in) and from the controller to the switch (Flow_mod and Packet_out). In addition, the controller workload varies depending on whether it is packet_ out or a packet_out and flow modification, or a heartbeat message.

Thus, we need to consider offloading heterogeneous workload to a GPU as opposed to offloading homogeneous workload to a GPU offloading 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 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 offloading issues as it relates to a GPU based SDN controller.

III. Basic GPU Implementation

We have implemented a basic version of the GPU SDN Controller, 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 can produce as many packets per second as we want. It allows us to fully utilize the CPU and 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 SDN controller is heterogeneous in terms of both packet size and computational cost. In this paper we will limit the discussion to packet size heterogeneity. The input queue to the SDN controller consists of a set of packets that vary in size. 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 machine 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 machine is equipped with four Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 32 GB memory, and one NVIDIA Tesla K40c (2880 cores, 12GB RAM). The machine has installed Red Hat 4.4.7-3.

C. Controller Structure:

The GPU Controller is a software framework that consists of code running on both CPU and GPU. Figure 2 illustrates the workflow. The controller divides the CPU threads in two: the producer threads and the consumer threads. The producer threads are responsible for checking if the connected clients 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 back to the clients the packet_out or flow_mod that the GPU has produced. 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 time point. The number of threads that are active depends on the number of clients 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 transfered from CPU memory to GPU memory. The GPU kernel is invoked to process all packets in parallel. The GPU output is stored 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 clients.

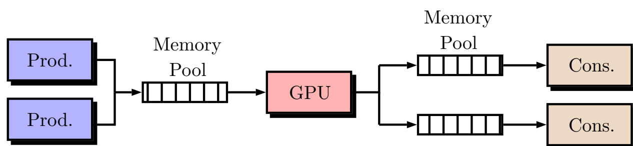

Fig. 3. GPU SDN Controller workflow.

D. Workflow:

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

- **Producer-CPU:** Each producer fetches and classifies the incoming packets of the clients and store them into the processing memory pool for later inspection by the GPU. By classifying the packets before they are stored in the memory pool, we reduce control divergence in the GPU execution.
- **Inspection-GPU:** The producer memory pool is transfered 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:

In the basic implementation, we let every GPU thread process one variable size packet. And we do not perform any optimization. Figure 4 depicts the performance of the kernel 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. And loop iterations are divided evenly among CPU threads. We measured the performance of the eight-core CPU and the two distinct GPUs using 82B size packets. 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 times. We expect to see an even more impressive performance improvement by implementing all the potential improvements described in section IV.

F. Bottleneck Analysis:

After thoroughly analyzing our application performance, we found that the major bottleneck is the transfer time for the batch processed packets. Batch processing of numerous packets significantly lowers per-packet processing overhead. However, it increases transfer time and thus leads to performance trade-offs. By transferring large batch of packets

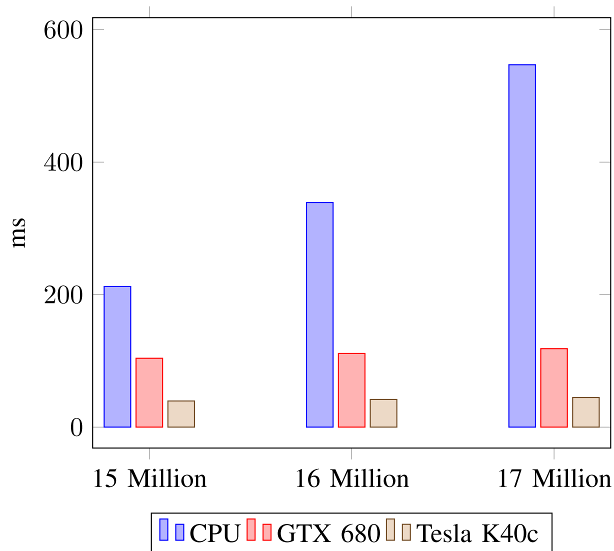

Fig. 4. CPU kernel time vs GPU kernel time using 82B size packets.

from the CPU memory to GPU memory, the PCI-e and the CPU memory bandwidth then become major bottlenecks. The current experiment configuration allows us to transfer 13 GB/s for reading from CPU memory and 12 GB/s for transferring over the PCI-e bus. In the worst case where we transfer 2GB of data to the GPU, it costs around 320ms from CPU memory to GPU memory as illustrated in figure 2, and another 320ms from GPU memory to CPU memory. In total it ends up with a transfer time of 640ms, 6 times slower than the kernel execution. Besides the memory transfer bottlenecks, we also that the CPU can not keep up with the GPU processing, which has become another bottleneck.

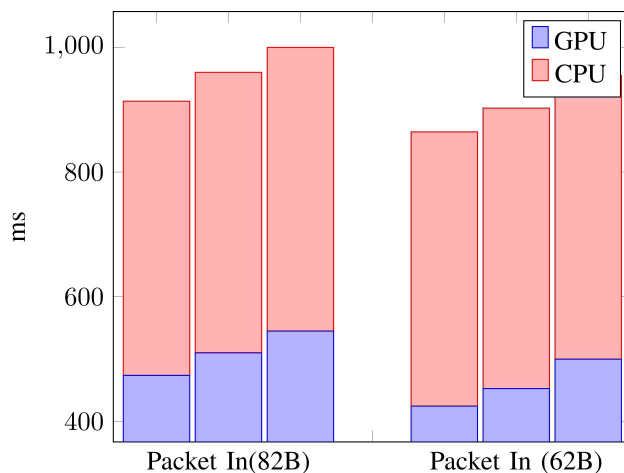

Fig. 5. CPU and GPU processing time.

Figure 5 shows the GPU + CPU performance of the controller processing 15 Million flows/s. As can be seen from figure 5, the CPU in this situation is the bottleneck because the CPU needs to perform 2 times the total amount of packets the GPU processes (it needs to send and receive the packets) while

CPU's computation horsepower is less than GPU's. However, even with two major performances bottlenecks, our controller yields 20% to 30% of improvement with one GPU in the worst case scenario when compared with the most popular controllers such as Nox [3], Pox [4], OpenDayLight [5] and etc. We expect to see an improvement around 45% to 60% if we add one extra GPU to the current configuration.

G. Transfer time:

Figure 6 illustrates the time for transferring 17 Million packets from the CPU memory to the GPU memory (as illustrated in figure 2). For all the tests we used a heterogeneous mix of 62B and 82B packets. We find that by having heterogeneity in the packet data structure, we can improve the transfer time of our controller. Conventionally, a fixed-size data structure is for all packets. Thus the size of the data structure needs to be the same as the maximal packet size. However, the uniformity increases data transfer overhead. Even when there is a 50%-50% 82B/62B heterogeneity, the transfer time outperforms the 100% 82B by a factor of 1.1. In the best case where we have 90%-10% 62B/82B heterogeneity, the transfer time improves by a factor of 1.3. The performance improvement factor increases as the total number of packets increases. In this implementation we only consider the packet size heterogeneity; we are planning to benefit from the computational heterogeneity in future improvements.

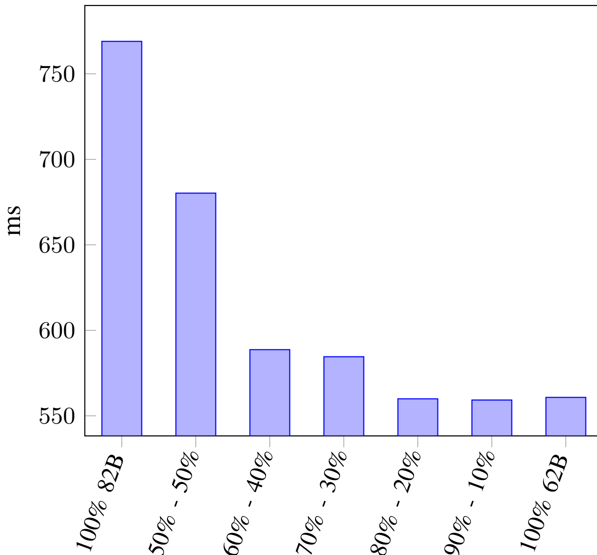

Fig. 6. Packet size heterogeneity: using two different sizes, 62B corresponds to the bottom percentage and 82B corresponds to the top percentage.

H. Hypothetical number of switches:

The main goal of this research is to mitigate the scalability problem that SDN presents by using GPUs. In the research of Theophilus et al, they mention that after analyzing 10 different data centers, for as few as 100 edge switches, an SDN Controller can see 10 new flows per μs or 10 million flows per second [10] in the worst case. With picks of 10 million flows per second, the current node configuration and a homogeneous workload, the maximum number of switches that the GPU controller can handle is 154. The number of switches that the controller can handle doubles if we add another GPU, which makes it 307 switches. By simply removing the CPU performance bottleneck that the CPU can't keep up with the GPU mentioned in section III, the GPU controller can handle 175 switches. By adding one extra GPU, the number of switches the controller can handle becomes 348.

IV. Potential GPU Improvements:

A. Unsorted vs Sorted:

A naive approach will be to take the k packets of the buffer and map it to k GPU threads without consideration of load valancing however this will lead to control flow divergence therefore the approach we take here is to sort the packets and then map it to k threads of the GPU. GPUs execute code in warps of 32 threads. Each thread in a warp should ideally follow the same execution path at runtime. The threads in one warp follow single instruction multiple data (SIMD) execution pattern, which means they run in lock-step. In the case where a branch in the code exists and only some threads follow that branch, these threads execute first while the renaming threads stay idle. For this reason branch divergence has a significant impact in the computational time of the GPU kernel. Our application presents that large amount of heterogeneous packets causes large amount of warp divergence because of the way OpenFlow is designed. By simply sorting the packets before they are uploaded to the GPU, we can assign the same size packets to consecutive threads, thus reducing the divergence and increasing the processing power of the GPU.

B. Heterogeneity:

Heterogeneity is the number one cause of warp divergence for GPUs. Most of the applications designed to run in the GPU try to avoid or minimize the heterogeneity, but in our situation we benefit from it. Large amount of heterogeneity allows us to transfer greater amount of packets through the PCIe bus if we compare it with the worst case scenario where we have a homogeneous workload of large size packets. As demonstrated in section III, the data transfer performance of the controller improves substantially even at a 50%-50% 82B/62B mix ratio.

C. 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.

V. Conclusion

We have presented a 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 flows/s that a regular controller can handle. By doing that we were also able to maximize the number of simultaneous switches a controller can handle. We believe that by efficiently implementing all the potential improvements described in section IV and other techniques such as packet compression and packet reconstruction, we will be able to improve the performance of our controller tremendously.

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