Harnessing GPU Computing Power to Improve Performance of SDN Controller

Muhammad Imran[†] **and Muhammad Shamim Baig**[††],

Department of ECE, Center for Advance Studies in Engineering, Islamabad, Pakistan

Summary
Software Defined Network (SDN) has shown substantial benefits over the legacy network and fueled the implementation of a variety of innovative and intelligent applications on SDN Controller. However, these applications put the performance of SDN controller under question; since most of these applications excessively demand computing resources of SDN controller resulting in increased flow processing delay, and consequently performance of the controller reduces. Accordingly, in this paper, we investigate the potential of Graphics Processing Unit (GPU) to address this performance issue by accelerating the computationally/memory intensive tasks of SDN applications on GPU. More specifically, in this paper, we are considering SDN based traffic load balancing application in a large scale Data Center Network (DCN) as a case study to see how GPU based approach can improve performance of SDN controller. We offload computations of traffic load balancing application on GPU and analyze the performance gains in terms of throughput, latency and speedup. The preliminary performance evaluation results show that GPU has an impressive capability to improve performance of SDN controller.

Key words:
SDN, OpenFlow, Controller, GPU, GPGPU Computing, CUDA

1. Introduction

In legacy networking paradigm, data plane and control plane are tightly coupled and physically located in packet forwarding devices such as routers and switches, posing many challenges regarding utilization, reliability, security, cost etc, of the networks [1]. To address these challenges, a novel paradigm "Software Defined Network" (SDN) has emerged in recent years [2]. In this paradigm, the entire control plane functionalities are pulled out from forwarding devices and shifted to a centralized machine called SDN controller. SDN controller maintains global view of entire network and provides its abstract view to the applications running on the controller. Furthermore, SDN controller facilitates programmability for forwarding devices through an open standard interface– Openflow [2]. The vision of unified control plane with global network view and network programmability provide an opportunity to gain fine-grained centralized control and visibility over both the traffic flows and network resources. Consequently, SDN has gained a considerable attention from both the research community and industry [3]; that have been spending earth-shattering efforts for the development of sophisticated and intelligent applications around SDN. Examples are QoS Aware routing [4-7], Traffic Load Balancing [8-21], Attacks Detection and Mitigation [22, 23] and many others [3]. The results presented in the above referred papers have shown that SDN has a great potential to improve security, reliability, energy efficiency, resource utilization etc, of the networks. However, most of these applications employ compute or memory intensive algorithms which overload the controller and drag down its performance. For instance, SDN based traffic load balancing application, which is one of the main SDN applications gaining considerable attention in challenging environments such as Data Center Networks (DCNs), typically employs algorithms which consider current status of the network and regulate traffic flows to optimize certain network performance criterion such as maximum link utilization, power consumption, delay, congestion etc [8-12]. These algorithms excessively demand computing resources and create sever processing bottlenecks resulting in lessening the performance of controller. This, in turn, raises an urgent call for high processing power at SDN controller.

Contemporary approaches— coarse-grained centralized flow control [8, 11, 12, 21, 24], multi-core controllers [25-29] and distributed controllers [30-33], do not address this issue adequately. One of the promising directions in this regard is to use GPU as a co-processor in SDN controller to accelerate execution of the compute/memory intensive algorithms of SDN applications, and thus improve performance of the controller. The motivation comes from the fact that GPUs have recently evolved into massively multithreaded parallel architectures which have an immense number of high clock-rate cores along with tremendous memory resources [34]. In addition, GPUs are becoming progressively more programmable [35] and showing impressive speedups for a variety of non-visual, General Purpose computations known as GPGPU Computing [36, 37].

Considering the growing demand of high processing power at SDN controller and inspired by the momentous computing power of GPUs, we study the use of GPU as an alternative computing resource in SDN controller. For this

Manuscript received May 5, 2016
Manuscript revised May 20, 2016

purpose, we have considered SDN based traffic load balancing application, hereafter Adaptive flow scheduling application, as a case study and elaborated the potential of GPU to accelerate the adaptive flow scheduling process. Note that this paper is not about adaptive flow scheduling, but the novelty of this paper is on the concept of exploitation of GPU computing power to accelerate flow processing at SDN controller and consequently, to improve controller's performance. We have designed a GPU based adaptive flow scheduling application by parallelization of the main tasks of flow scheduling process and implement it on a machine equipped with NIDIA's GPU. We have compared the results of our GPU based adaptive flow scheduling application (named GPU based Controller) with that of the CPU based version (named CPU based controller). Our results showed that GPU based controller outperforms CPU based controller. GPU based controller showed five to ten times reduction in execution time of adaptive flow scheduling process on large scale DCNs. The results validate the efficacy of the use of GPU to address the performance challenges of SDN controller.

In the rest of this paper, we explain background information and related work, our strategies to parallelize adaptive flow scheduling process at SDN controller, performance evaluation methodology and results. Lastly we conclude the paper along with future directions.

2. Background and Related Work

In this section, we first demonstrate the performance challenges of an SDN controller in the context of adaptive flow scheduling process by providing a high-level overview of adaptive flow scheduling application. We then describe shortcoming of current approaches proposed to address these challenges and discuss about GPGPU computing briefly.

2.1 Traffic Load Balancing using SDN (Adaptive Flow Scheduling)

Traffic load balancing plays an indispensable role to optimize performance of an operational network at both the traffic and resource level. Recently, for that reason, a flurry of SDN based traffic load balancing applications (Adaptive flow scheduling) have been developed which optimize performance of network from different aspects [6-19] such as minimization of maximum link utilization, minimization of power consumption etc. For our discussion, we use the most popular optimization goal, minimization of maximum link utilization [9-11, 15, 17, 20, 21]. In this context, SDN controller periodically regulate traffic load on network links by dispersing traffic flows away from vastly utilized links towards less utilized links to accommodate demands of the traffic flows and minimize the maximum link utilization in the network. Demand of a flow is the bandwidth required by the flow to carry its data.

Typically, an SDN controller that performs adaptive flow scheduling comprises of three applications — Topology, Monitoring and Adaptive flow scheduling. Topology application periodically sends queries to network switches to maintain network topology graph up-to-date. Monitoring application periodically gathers link counters from network switches and then consolidates the received counters to maintain up-to-date Residual Capacities (RCs) on all links in the network.

Adaptive flow scheduling application, which is the focus of this paper, optimizes traffic load in the network after a specific time period (termed as flow scheduling period). In every flow scheduling period, adaptive flow scheduling application takes up-to-date topology graph from topology application, most recent RCs on network links from monitoring application and demands of the flows to be scheduled (Scheduling flows) which are communicated by the flow's sources or some other servers in the network [9, 11], and then computes an optimal path for each of the scheduling flows. An optimal path of a scheduling flow is the path which can carry demand of the flow while not exceeding capacity of the links along the path and minimizes the maximum link utilization. In general, computing optimal paths of scheduling flows is proven to be NP-complete in a large scale network [17], leading to the proposal of many heuristics [9-13, 15, 17, 18, 20, 21]. Among the proposed heuristics, Worst-Fit heuristic [10, 17, 20] is widely used as it distributes traffic load more evenly in the network than the other heuristics such as First-Fit and Best-Fit [17]. In Worst-Fit heuristic approach, for each of the flows to be scheduled within a scheduling period, adaptive flow scheduling application first takes its set of equal hop paths, which are typically pre-computed, and then computes RCs on each path in the path set. RC of a path is the minimum of RCs on all links along the path. After that, flow scheduling application selects a path which has maximum RC and fulfills the flow's demand as its optimal path called Worst Fit Path. Finally, demand of the scheduling flow is subtracted from RCs of all links along the Worst Fit Path and flow entries are inserted in all switches along the Worst Fit Path. Algorithm 1 describes adaptive flow scheduling process with Worst-Fit heuristic approach whereas table 1summarizes the symbols used in algorithm 1.

Table 1: Symbols

Symbols	Meaning
V	Set of Switches in the Network
Ł	Set of Links in the Network
RC_ℓ	Residual Capacity on the Link ℓ
RC_β	Residual Capacity on the path β

$MaxRC_{path}$	Path which has Maximum RC
$f.src$	Source switch of the flow f
$f.dst$	Destination switch of the flow f
f_d	Demand of the flow f
$G^*(V, Ł)$	Network Topology Graph.
$\Phi_f = \{\beta_1, \beta_2, \ldots, \beta_P\}$	Set of paths of the flow f
$\varphi_\beta = \{ \ell_1, \ell_2, \ldots, \ell_Q \}$	Set of links along the path β
WF_{Path}	*Worst Fit Path.* A $MaxRC_{Path}$ which can carry flow demand
$F = \{ f_1, f_2, \ldots, f_N \}$	Set of Flows to be scheduled (referred as scheduling flows)

```
1:  For Each flow $f \in F$
2:      $\Phi_f \leftarrow$ GetPathSet ($f.src$, $f.dst$)
3:      For each path $\beta \in \Phi_f$
4:          $RC_\beta \leftarrow \min_{\forall \ell \in \varphi_\beta} RC_\ell$
5:      End For
6:      $MaxRC \leftarrow \max_{\forall \beta \in \Phi_f} RC_\beta$
7:      $MaxRC_{path} \leftarrow$ Id of the path having $MaxRC$
8:      If $f_d > MaxRC$
9:      $WF_{path} \leftarrow MaxRC_{path}$
10:     $RC_\ell \leftarrow RC_\ell - fd$        $\forall \ell \in \varphi_{WFpath}$
11:     InsertFlowEntries($WF_{path}$)
12:     End If
13: End For
```

Algorithm 1: Adaptive Flow Scheduling process with Worst-Fit heuristic Approach

2.2 Controller's Performance challenges

In algorithm 1, the worst case complexity of scheduling a flow on its WFpath is O (Q*P+P). Since SDN controller has to schedule N flows, hence the overall time complexity of adaptive flow scheduling process within a scheduling period is O(N(Q*P+P)) indicating that SDN based adaptive flow scheduling becomes problematic in large scale dynamic networks where each flow has vast number of paths and SDN controller is supposed to schedule large number of flows in the network. One example of such networks is Data Center Network (DCN) whose numbers and sizes are growing exponentially[38]. For example, DCNs of Yahoo, Microsoft, and Google host hundreds of thousands of servers [39-41] in their DCNs. In DCN, a large number of servers are densely packed in a well-defined hierarchy to provide an ample number of paths between any pair of servers in the network [42]. Additionally, inter-flow arrival time in DCNs varies from 1 flow per 15 ms and 100 flows per ms at servers and Top-of-Rack switches, respectively. More importantly, DCN traffic is bursty in nature and follows a heavy-tailed distribution [43, 44] requiring frequent execution of flow scheduling process at SDN controller. These facts indicate that adaptive flow scheduling process in such networks consumes a great deal of computing resources of SDN controller resulting in increased processing delay and creates a computational bottleneck at the controller.

2.3 Related Work

Current approaches proposed in the literature to address the controller's performance challenges can be classified into three categories.

Coarse-grained centralized flow control [8,11, 12, 21, 24]: These approaches reduce processing load of SDN controller by scheduling only some specific flows such as elephant flows (whose size exceeds some pre-defined threshold value (for example 100 MB [8]). Pre-defined threshold value is a critical parameter in these approaches. When threshold value is kept large, number of scheduling flows reduces and consequently, the processing load of SDN controller reduces. However, network performance gain also reduces as reported in [45]. On the other hand, small threshold value puts the performance of controller under question, since the controller has to process a large number of flows.

Multi-core Controller approaches [25-29]:
These approaches use multi-core parallel architecture to increase the processing power of SDN controller. NOX-MT [28], Beacon [26], Maestro[27] and Macnett[29], are some examples of state of the art multi-core controllers. The reported performance of these controllers are high but simple L2 learning switch application is used for the performance evaluation of these controllers which is itself a question mark on recording performance as the CPU cores or memory bandwidth becomes bottleneck with compute or memory intensive algorithms as reported in [37]. Besides this, locking mechanism used for exclusive write in shared data structure reduces the performance of multi-core controller as the number of cores and/or writing workload of the cores increases [28, 29].

Distributed controller Approaches [30-33]:
In these approaches, multiple controllers are distributed across the network, each managing the flows initiated from a subset of switches, while global view of the entire network is kept consistent and synchronized across all the controllers by means of some distributed file systems. However, keeping the global network view consistent and synchronized causes high communication overheads and add additional latency in processing of the flows [46]. From the above discussion, we can conclude that current approaches either lose network performance gains and/or add additional overheads (e.g. communication, contention, synchronization etc), which itself can be a performance bottleneck as the number of cores in multi-core controller and/or number of distributed controllers increases. Thus,

there is a need of an alternative computing resource at SDN controller for the rapid processing of traffic flows. As GPUs have become powerful computing resource and are adopted increasingly for General-Purpose computations (GPGPU computing) [34-37] so they can be good candidates for this purpose. In next subsection, we briefly shed light on the hardware and software aspects of GPGPU computing considering the NVIDIA's GPUs.

2.4 GPGPU computing

Born as Graphics Processors, recently GPUs have evolved into massively multithreaded many-core architecture. Today's GPUs are comprised of hundreds or thousands of Streaming Processors (SPs) organized in a number of Streaming Multiprocessor (SMs). In addition, GPU have several types of off chip and on chip memories (e.g. global memory, constant memory, shared memory, registers) which differ with each other in terms of size, access latency, access type and scope [34].

GPUs are typically connected to the host by PCI Express bus (PCIe) and works as a co-processor or accelerator of the CPU. CUDA (Compute Unified Device Architecture) [35] is a parallel programming framework resealed by NVIDIA to simplify the use of GPU as an accelerator and to offload General Purpose computations on their GPUs. Applications developed in CUDA are comprised of at least one kernel function which defines the segment of CUDA code to be executed on GPU by a large number of threads. These threads are organized in 1D, 2D or 3D arrays of threads called thread blocks. Thread blocks are organized into 1D or 2D array of blocks called grid. To identify a thread within the grid, CUDA assigns a unique Id to each thread of a thread block in the form of three dimensional coordinates— (threadIdx.x, threadIdx.y, threadIdx.z), and assigns a unique two dimensional Id— (blockIdx.x, blockIdx.y) to each thread block in the grid. Number of threads in a block and number of thread blocks in a grid are defined when kernel is invoked.

At the hardware level, thread blocks are assigned to SMs which employ SIMT (Single Instruction Multiple Thread) execution model in which a group of coordinated threads called warp execute the same instruction in parallel to process different data (data level parallelism). If the instruction of any warp requires excessive cycles to complete, the SMs suspended its execution and select another active warp to execute its instructions.

3. Methodology

In this section, we explain our methodology used to improve the performance of SDN controller using GPU in the context of adaptive flow scheduling application described in section 2.1. We first describe Fat-tree DCN architecture to provide a foundation for latter discussion and then explain the data structure and parallelization strategies we have used to design the GPU based adaptive flow scheduling application. Finally, we explain workflow of flow scheduling process of GPU based adaptive flow scheduling application.

3.1 Fat-tree Data Center Network Architecture

Fat-tree DCN architecture is a prevailing architecture used to build a cost effective large scale DCN. An m-ary Fat-tree [42] is a multi-rooted tree like topology comprised of 5m2/4 identical switches, each of which has m numbers of bidirectional ports. These switches are structured into three tiers i.e. Core Tier, Aggregate Tier and Edge Tier as illustrated in Fig 1. The Core Tier is in the root of the tree and comprised of m2/4 switches called Core switches. The Aggregate Tier is in the middle of the tree and comprised of m2/2 switches called Aggregator switches. The Edge tier is at the leaves of the tree and comprised of m2/2 switches called Edge switches.

Edge and Aggregator switches are further organized into m Pods, each containing m/2 number of Edge switches as well as m/2 number of Aggregator switches. Each Edge switch in a Pod is connected to m/2 Servers through its m/2 ports and the remaining m/2 ports are connected to m/2 Aggregator switches in the Pod. Each Core switch has one port connected to one of the m Pods. The ith port of any Core switch is connected to Pod i such that Aggregator switches of each Pod are connected to Core switches on m/2 strides. There are m2/4 equal hop paths between any two servers in m-ary Fat-tree. Each of the paths between two servers belonging to two different Pods, called inter-Pod path, passes through a Core switch. Architecture of a 4-ary Fat-tree DCN is shown in Fig 1.

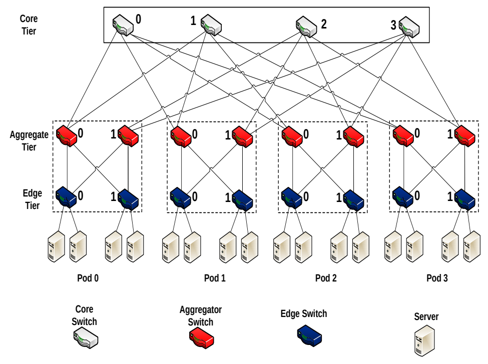

Fig 1: 4-ary Fat-Tree DCN Architecture

3.2 GPU based Adaptive Flow Scheduling Application

3.2.1 Data Structure

Let all Core switches, Pod, Edge and Aggregator switches of each Pod, are indexed with non-negative integers from left to right as shown in Fig 1. Core switches are indexed from 0 to $m^2/4 - 1$ and Pods are indexed from 0 to m. Both of the Edge switches and Aggregator switches of a Pod are indexed from 0 to $m/2 - 1$. Let the scheduling flows are also indexed from 0 to N and each flow f is identified by its source Edge switch *srcE,* source Pod *srcP,* destination Edge switch *dstE,* destination Pod *dstP*. We also index all the inter-Pod paths of a scheduling flow from 0 to $m^2/4 - 1$ such that the i^{th} path of a flow passes through i^{th} Core switch.
Based on the above indexing scheme, we used four 3D matrices— UL_{EA}, UL_{AC}, DL_{AC}, DL_{EA}, to stores RCs on the links in *m-ary* Fat-tree topology. We referred these matrices as *Link RC Matrices.* In addition, we also used three vectors— V_{Fd}, $V_{WFpathIds}$, V_{PRC} and a 2D matrix— M_{FA} to store the data of the scheduling flows. Definition and description of used matrices & vectors are summarized in table 2.

Table 2: Notations and Data Structures

Symbols	***Explanation***
m	Number of ports in a switch
srcE	source Edge switch
srcP	source Pod
dstE	destination Edge switch
dstP	destination Pod
fd	demand of the flow f
$V_{Fd}[f]_{1 \times N}$	*Flow demand vector*. $V_{Fd}[f]$ stores demand of the flow f.
$V_{WFpathIds}[f]_{1 \times N}$	*Worst Fit Path Ids Vector*. $V_{WFpathIds}[f]$ stores *Worst Fit Path Id* of the flow f
$V_{PRC}[i]_{1 \times m^2/4}$	*Path Residual Capacity Vector*. $V_{PRC}[i]$ stores RC of i th path.
$M_{FA}[i][j]_{N \times 4}$	*Flow Address Matrix.* $M_{fA}[i][0]$, $M_{fA}[i][1]$, $M_{fA}[i][2]$, $M_{fA}[i][3]$ store *srcE, srcP, dstE, dstP* of i th flow.
$UL_{EA}[k][i][j]_{m \times m/2 \times m/2}$	store RC of the UpLink connecting Edge switch i with Aggregator switch j in Pod k.
$UL_{AC}[k][i][j]_{m \times m/2 \times m/2}$	store RC of the UpLink connecting Aggregator switch i of Pod k with Core switch $i * m/2 + j$
$DL_{AC}[k][i][j]_{m \times m/2 \times m/2}$	store RC of the DownLink connecting Aggregator switch i of Pod k with Core switch $i * m/2 + j$.
$DL_{EA}[k][i][j]_{m \times m/2 \times m/2}$	store RC of the DownLink connecting Edge switch i with Aggregator switch j in Pod k.

3.2.2 Parallelization Strategies

GPU application encompasses multiple segments that are executed on either the CPU or the GPU. All segments that exhibit little or no data parallelism are executed on the CPU and if there is much data level parallelism in the segments, they are parallelized and executed on GPU as a kernel functions. Scheduling of each flow f described in algorithm 1(section 2.1) can be divided into two segments. The first segment is composed of line 3-5 where RC on each path of a scheduling flow is computed, we termed this segment as ComputePRC task. While the second segment is line 6-10 where Worst Fit Path (WFpath) of a scheduling flow is selected and links along the path are updated. We termed this segment as WFPathSelection task. Both of these tasks are analyzed in next subsections to achieve data parallelism in the tasks.

3.2.2.1 Parallelization of ComputePRC task

As stated earlier, there are $m^2/4$ paths between any two servers in m-ary Fat-tree DCN. This implies that the sequential process of computing RCs of all paths of a scheduling flow in m-ary Fat-tree involves $m^2/4$ iterations. These iterations are independent of each other's as the computation of a path RC in algorithm 1 (line- 4) does not involve RC of any other path. So these iterations exhibit high level of data parallelism, which makes *ComputePRC task* a good candidate to be implemented on GPU. RCs of all paths of a scheduling flow can be computed in parallel by executing the *ComputePRC task* with $m^2/4$ threads and assigning computation of RC of a path to one of the $m^2/4$ threads. The simplified version of *ComputePRC kernel* is explained in Fig 2.

```
Input: $UL_{EA}$, $UL_{AC}$, $DL_{AC}$, $DL_{EA}$, $M_{FA}$, $m$, $f$
Output: $V_{PRC}$,
    // Id of the path to be processed
1:  ꞵ ← threadIdx.x + blockIdx.x * blockDim.x
    // Id of Aggregator switch through which the path "ꞵ" passes
2:  $x$ ← ꞵ div ($m/2$)
    //Column index of the $UL_{AC}$ , $DL_{AC}$
3:  $y$ ← ꞵ mod ($m/2$)
    // source Edge switch & source Pod of the flow $f$
4: $srcE \leftarrow M_{FA}[f * 4]$; $srcP \leftarrow M_{FA}[f * 4 + 1]$;
    // destination Edge Switch & destination Pod of the flow $f$
5: $dstE \leftarrow M_{FA}[f * 4 + 2]$; $dstP \leftarrow M_{FA}[f * 4 + 3]$;
    //RC of the path ꞵ
6: $UL_{RC}$ ← MIN ($UL_{EA}[srcP * m^2/4 + srcE * m/2 + x]$ , $UL_{AC}[srcP * m^2/4 + x * m/2 + y]$ )
7: $DL_{RC}$ ← MIN ( $DL_{AC}[dstP * m^2/4 + x * m/2 + y]$ , $DL_{EA}[dstP * m^2/4 + dstE * m/2 + x]$ )
8: $V_{PRC}$[ꞵ] ← MIN ($UL_{RC}$ , $DL_{RC}$)
```

Fig 2: Simplified version of ComputePRC kernel. MIN (a, b) returns minimum of a and b.

In this kernel, each thread first computes *Id* of the path assigned to it for computing RC of the path and finds

index of the Aggregator switch through which its assigned path passes. The key insight here is that in *m*-ary Fat-tree architecture, Aggregator switches in a Pod are connected with $m^2/4$ Core switches on a stride of $m/2$ [42]. In other word, i^{th} Aggregator switch in all Pods is connected with the $m/2$ Core switches which have *Ids* from $i * m/2$ to $i * m/2 + 1$.For example, in *4-ary* Fat-tree architecture shown in Fig 1, Aggregator switch 1of all Pods are connected with Core switches 2 and 3. Since we have assigned an *Id* "*i*" to a path passing through Core switch *i*, thus, the paths which have *Ids* from $i * m/2$ to $i * m/2 + 1$ pass through an i^{th} Aggregator switch. Hence, given the path *Id* "β", the *Id* of the corresponding Aggregator switch can be determined by dividing β with *m/2*.
At line 3, each thread computes column indices of UL_{AC} and DL_{AC}. Since $UL_{AC}[k][i][j]$ and $DL_{AC}[k][i][j]$ store RCs of the links connecting Aggregator switch *i* of Pod *k* with core switch $i * m/2 + j$ in upward and downward direction respectively, so given the path *Id* "β", which is the Core switch *Id* as well, the column index (value of *j*) of these matrices can be computed by taking the modulus of β with *m/2*. After determining the indices of the elements required to compute RC of the path, each thread reads RCs of the links along the assigned path of the flow *f* from *Link RC Matrices*, compared them and finally store minimum of them in V_{PRC} at index "β" (line 4-8).

3.2.2.2 Parallelization of *WFPathSelection task*

In *WFPathSelection task*, RCs of all paths of a scheduling flow *f* are sequentially explored to search maximum of RCs of the paths, which is intrinsically a reduction process. The only difference is that instead of only searching the maximum of the path RCs, *Id* of the respective path is also found. This process can be performed in $\log(m^2/4)$ steps using parallel reduction algorithm [47]. Various strategies for the implementation of parallel reduction on GPU are presented in [48] showing significant performance gain with millions of data elements. However, the numbers of paths in a network are not so huge resulting in low performance gain on GPU. For that matter, a possible approach is to compute path RCs on GPU and execute *WFPathSelection task* on CPU for effective utilization of computing resources. However, this strategy entails two expensive time consuming data transfers over PCIe; path RCs computed on GPU must be transferred to CPU and *Id* of *Worst Fit Path* selected by CPU must be transferred to GPU. To avoid these data transfers, we implemented the *WFPathSelection task* on GPU. Due to Space limitation, we did not present our *WFPathSelection kernel*. We used the same kernel 7 given in [48] with some modification. First, instead of using addition operator, we used maximum operator (>) for reduction process. Second, we used two shared memory arrays, one of the arrays is used to store RCs of the paths and the other is used to store *Ids* of the paths. Finally, one thread block is used to search maximum of path RCs and its index, (*MaxRC*, $MaxRC_{path}$), from V_{PRC} vector. After searching *MaxRC* and $MaxRC_{path}$, first thread of the thread block reads flow demand *fd* from flow demand vector V_d and compares it with *MaxRC*. If *MaxRC* is found to be greater than *fd*, it updates RCs of links along the $MaxRC_{path}$ and writes the value of $MaxRC_{path}$ in $V_{WFpathIds}$ vector at index *f* otherwise it writes -1 to communicate the CPU that *Worst Fit Path* is not found for the flow *f*.

3.3 Work flow

Having described the parallelization of *ComputePRC task* and *WFPathSelection task*, we now present the work flow of adaptive flow scheduling process on GPU
Processing of adaptive flow scheduling on GPU takes three steps:
In the first step, input data which includes *Link RC Matrices*, *Flow Address Matrix* and *Flow demand vector* (described in table 2), is copied on global memory of the GPU. In second step, *ComputePRC* and *WFPathSelection kernels* are invoked for each of RBS (Result Batched Size) number of flows, and their *Worst Fit Paths* are computed on GPU. In third step, the computed *Worst Fit Path Ids* are copied from GPU to CPU.
This process continues until all flows have been processed. Workflow of adaptive flow scheduling process on GPU is depicted in Fig 3.

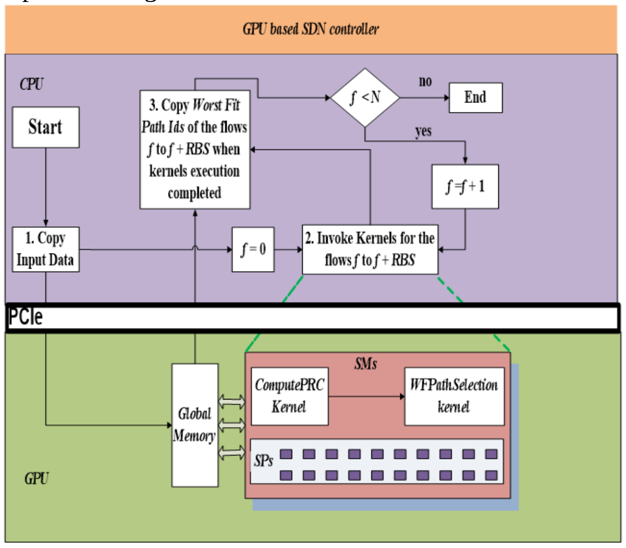

Fig 3: Adaptive Flow Scheduling process at SDN controller using GPU

The value of RBS plays an important role to achieve high performance gain from GPU, since Worst Fit Path Ids are copied to CPU after complete processing of RBS number of flows on GPU. Higher value of RBS leads to lower number of data transfers over PCIe bus, but the cost is extra delay resulted by waiting for the GPU to complete processing of a higher number of flows. This delay can be reduced by decreasing the value of RBS however, in this case a momentous time is spent in carrying out a higher number of data transfers (Worst Fit Path Ids) over low bandwidth PCIe bus. This issue can be solved by choosing an appropriate value of RBS which leads to an acceptable level of delay and number of data transfers over PCIe.

4. Performance Evaluation

4.1 Experimental Methodology

Experiments setup: We implemented GPU based Adaptive flow scheduling application (named GPU based controller) and its CPU based version (named CPU based controller) on a commodity machine equipped with Intel Core i5 CPU (4 physical Cores operating with 2.8 GHz clock frequency), 4GB RAM and one GPU (NVIDIA GeForce GT 640). The machine run an Ubuntu Desktop 15.04 (64 bits) OS. The entire GPU based adaptive flow scheduling application was developed using CUDA SDK 7.5.

Input Data sets: We consider a set of 10K flows to be scheduled on *64-ary* Fat-tree DCN, *128-ary* Fat-tree DCN, and *256-ary* Fat-tree DCN. For brevity, we referred *64-ary* Fat-tree DCN, *128-ary* Fat-tree DCN, and *256-ary* Fat-tree DCN as “DCN64, DCN128, and DCN256” respectively. Input data sets are generated by following the randomized traffic pattern used in [45]: Residual Capacities (RC) of all links of a Fat-tree DCN under study are randomly selected between 0 and 1. Source Pods and destination Pods of the scheduling flows are randomly selected between 0 and *m/2* - 1, and between *m/2* and *m* – 1, respectively. Source and destination Edge switches are randomly selected between 0 and *m/2* - 1. Flow demands are chosen randomly between 0 and 1.

Performance metrics: We considered Flow Scheduling Throughput, Flow Scheduling Latency and SpeedUp as performance evaluation metrics. Brief description of these metrics is given below,

<u>Flow Scheduling Throughput (FST)</u>*:* It measure number of flows processed by SDN controller per second.

<u>Flow Scheduling Latency (FSL)</u>*:* It is the time taken by SDN controller to process one flow.

<u>SpeedUp (SU):</u> It measures reduction of overall processing time when we offload flow scheduling process on GPU.

Experiment Scenarios: We conducted two sets of experiments for performance evaluation of GPU based controller. In each experiment, we generated input data set and copied it to global memory of the GPU. In first set of experiment, we measured FST without considering the data transfer overhead. We did not copy *Worst Fit Path Ids* of the flows from GPU to CPU in these experiments.

In second set of experiments, we copied *Worst Fit Path Ids* from GPU to CPU as described in section 3.3(explained in Fig 3) and measured FST, FSL and SU. The values of RBS (Result Batch Size) are varied from 2^0 to 2^{20}.

For performance comparison, we also measured the performance of CPU based SDN controller as the base reference using the same input data sets used to evaluate performance of the GPU based controller.

Each experiment was repeated 100 times and the average is used to measure the performance metrics.

4.2 Results

Fig 4 depicts FST of GPU based controller and CPU based controller on DCNs under study. Results are generated through experiment set 1 i.e. (without copying *Worst Fit Path Ids* to CPU)

Fig 4 shows that GPU based controller outperforms CPU based controller. GPU leads to 34.21%, 79.56% and 90.19% increase in FST on DCN64, DCN128 and DCN256, respectively. As the GPU based controller computes RCs of all paths of a flow in parallel, and the CPU based controller does this work sequentially, so higher FST is seen for the GPU based controller compared to the CPU based controller.

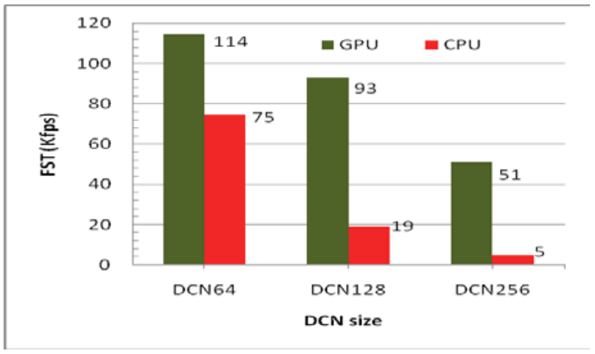

Fig 4: Flow Scheduling Throughput measured through experiment set 1

Fig 5 and 6 illustrate FST and FSL of GPU based controller on the DCNs under study for different values of RBS. Results are generated through experiment set 2 (i.e. time consumed in transferring *Worst Fit Path Ids* to CPU is also included).

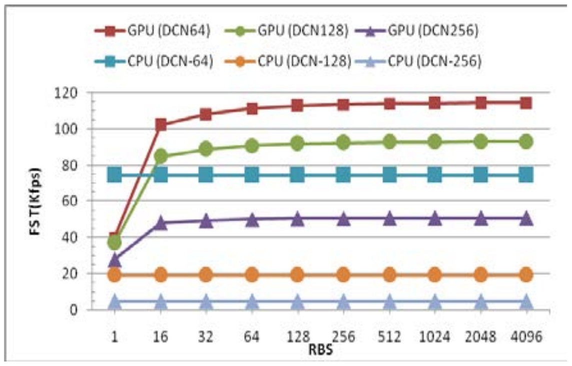

Fig 5: Flow Scheduling Throughput for different values of RBS (Result Batch Size). It is measured through experiment set 2.

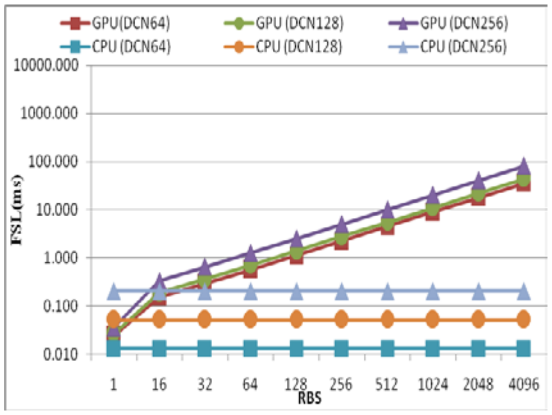

Fig 6: Flow Scheduling Latency for different values of RBS (Result Batch Size). The graph is ploted on a log scale with base 10. It is measured through experiment set 2.

The results reveal that performance gain of introducing GPU in SDN controller significantly depends on the value of RBS.

Fig 5 shows that for small values of RBS, the GPU based controller exhibits worse and slightly higher FST compared to the CPU based controller on DCN64 and DCN128, DCN256, respectively. On the other hand, for large values of RBS, the FST of GPU based controller is much higher than that of the CPU based controller. For RBS = 1, GPU leads to a 46 % decrease, 48% increase and 82% increase in FST of the controller on DCN64, DCN128, and DCN256, respectively. While the increase in FST for RBS = 1024 are 34%, 79.39% and 89.35% on DCN64, DCN128 and DCN256, respectively. The primary reason of low FST for small values of RBS is appeared to be due to a large number of time-consuming data transfers (*Worst Fit Path Ids)* over PCIe bus, as the increase in FST for RBS = 1024 is very close to the increase in FST observed in experiment set 1 (Fig 4).Thus, a large value of RBS is most appealing to use for achieving high FST of GPU based controller.

Unfortunately, large values of RBS lead to substantial increase in FSL of the GPU based controller as illustrated in Fig 6. For RBS = 1024, GPU leads to 99.85 % , 99.52% and 98.98% increase in FSL of the controller on DCN64 and DCN128, DCN256 respectively. The reason of this too much increase in FSL is the long waiting time of CPU to get *Worst Fit Path Id* of a flow from GPU; since the *Worst Fit Path Ids* are copied after complete processing of RBS number of flows on GPU. This indicates that small values of RBS are better to use instead of the large RBS values. But small values of RBS reduce FST significantly as illustrate in Fig 5.

Therefore, to get the better acceleration results from GPU, it is necessary to choose a suitable value of RBS that adapts to minimum reduction of FST with an acceptable level of FSL needs. From Fig 5, it can be observed that FST of GPU based controller increases slightly when the value of RBS goes beyond 64, 64 and 32 for DCN64, DCN128, and DCN256 respectively. Assuming additional latency of 1ms added by the batching mechanism is tolerable, FST achieved by the GPU based controller reaches about 112K fps, 91K fps, 50Kfps on DCN64, DCN128 and DCN256 for RBS = 64, 64 and 32 respectively.

Fig 7 shows acceleration results of GPU based controller for RBS= 64, 64, 32 on DC64, DCN128, and DCN256 respectively.

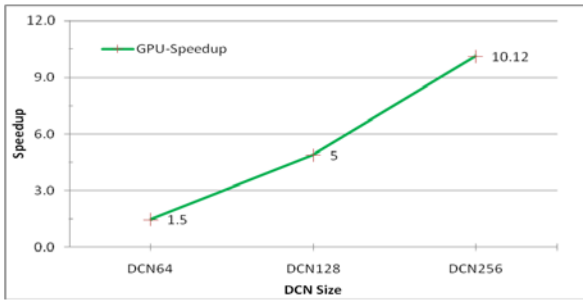

Fig 7 Speedup achieved by the GPU based controller for RBS =64, 64 and 32 on DCN64, DCN128 and DCN256 respectively.

Fig 7 shows that the execution time of flow scheduling process reduces when flow scheduling process is offloaded to GPU. GPU demonstrates 1.5X, 5X and 10.12X speed on DCN64, DCN128 and DCN256 respectively, compared to the CPU. The same speed ups on CPU based controller entails 2, 5 and 11 cores respectively. Furthermore, Fig 7 shows that an increase in DCN size yield higher speed up on GPU compared to the CPU which means a single GPU can perform work of more CPU cores. The results confirm our insight that the use of GPU in SDN controller leads to better acceleration of processing of flows and consequently improve its performance.

5. Conclusion

This paper strives to analyze the potential of GPUs to address the performance challenges of SDN controller by accelerating the compute/ memory intensive algorithms of SDN applications. We considered SDN based traffic load balancing (Adaptive flow scheduling) application in a large scale Fat-Tree Data Center Network and parallelized its two tasks: Computation of residual capacities on all path of a flow and selection of the path which has maximum RC. We offloaded these tasks on NVIDIA’s GPU and analyze the performance gain through detailed Experimentation.

The experimental results show that GPU bring significant increase in flow scheduling throughput of SDN controller, confirming the efficacy of the use of GPU for improving the performance of SDN controller. In future work, the memory hierarchy of GPU will be exploited to improve controller performance further.

Acknowledgments

The author acknowledges the enabling role of the Higher Education Commission Islamabad, Pakistan for research fellowship under HEC indigenous scholarship program.

References

[1] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, et al., "Rethinking enterprise network control," IEEE/ACM Transactions on Networking (TON), vol. 17, pp. 1270-1283, 2009.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, et al., "OpenFlow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol. 38, pp. 69-74, 2008.

[3] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," Communications Surveys & Tutorials, IEEE, vol. 16, pp. 493-512, 2014.

[4] F. Pop, C. Dobre, D. Comaneci, and J. Kolodziej, "Adaptive scheduling algorithm for media-optimized traffic management in software defined networks," Computing, vol. 98, pp. 147-168, 2016.

[5] M. S. Seddiki, M. Shahbaz, S. Donovan, S. Grover, M. Park, N. Feamster, et al., "FlowQoS: QoS for the rest of us," in Proceedings of the third workshop on Hot topics in software defined networking, 2014, pp. 207-208.

[6] A. Ghosh, S. Ha, E. Crabbe, and J. Rexford, "Scalable multi-class traffic management in data center backbone networks," Selected Areas in Communications, IEEE Journal on, vol. 31, pp. 2673-2684, 2013.

[7] A. Ishimori, F. Farias, E. Cerqueira, and A. Abelém, "Control of multiple packet schedulers for improving QoS on OpenFlow/SDN networking," in Software Defined Networks (EWSDN), 2013 Second European Workshop on, 2013, pp. 81-86.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in NSDI, 2010, pp. 19-19.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies, 2011, p. 8.

[10] T. Cheocherngngarn, H. Jin, J. Andrian, D. Pan, and J. Liu, "Depth-First Worst-Fit Search based multipath routing for data center networks," in Global Communications Conference (GLOBECOM), 2012 IEEE, 2012, pp. 2821-2826.

[11] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in INFOCOM, 2011 Proceedings IEEE, 2011, pp. 1629-1637.

[12] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: scaling flow management for high-performance networks," in ACM SIGCOMM Computer Communication Review, 2011, pp. 254-265.

[13] E. d. B. e Silva, G. Pantuza, F. Sampaio, B. P. Santos, L. F. Vieira, M. A. Vieira, et al., "Enforcing Link Utilization with Traffic Engineering on SDN."

[14] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, et al., "ElasticTree: Saving Energy in Data Center Networks," in NSDI, 2010, pp. 249-264.

[15] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, et al., "Achieving high utilization with software-driven WAN," in ACM SIGCOMM Computer Communication Review, 2013, pp. 15-26.

[16] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, et al., "B4: Experience with a globally-deployed software defined WAN," in ACM SIGCOMM Computer Communication Review, 2013, pp. 3-14.

[17] E. Jo, D. Pan, J. Liu, and L. Butler, "A simulation and emulation study of SDN-based multipath routing for fat-tree data center networks," in Proceedings of the 2014 Winter Simulation Conference, 2014, pp. 3072-3083.

[18] A. Lester, Y. Tang, and T. Gyires, "Prioritized Adaptive Max-Min Fair Residual Bandwidth Allocation for Software-Defined Data Center Networks," ICN 2014, p. 209, 2014.

[19] R. Trestian, G.-M. Muntean, and K. Katrinis, "MiceTrap: Scalable traffic engineering of datacenter mice flows using OpenFlow," in Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on, 2013, pp. 904-907.

[20] F. P. Tso and D. P. Pezaros, "Baatdaat: Measurement-based flow scheduling for cloud data centers," in Computers and Communications (ISCC), 2013 IEEE Symposium on, 2013, pp. 000765-000770.

[21] P. Wette and H. Karl, "HybridTE: Traffic Engineering for Very Low-Cost Software-Defined Data-Center Networks," in Software Defined Networks (EWSDN), 2015 Fourth European Workshop on, 2015, pp. 31-36.

[22] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in Recent Advances in Intrusion Detection, 2011, pp. 161-180.

[23] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in Local Computer Networks (LCN), 2010 IEEE 35th Conference on, 2010, pp. 408-415.
[24] J. Liu, J. Li, G. Shou, Y. Hu, Z. Guo, and W. Dai, "SDN based load balancing mechanism for elephant flow in data center networks," in Wireless Personal Multimedia Communications (WPMC), 2014 International Symposium on, 2014, pp. 486-490.
[25] "Floodlight Available: http://www.projectfloodlight.org/."
[26] D. Erickson, "The beacon openflow controller," in Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, 2013, pp. 13-18.
[27] Z. A. C. T. EugeneNg, "Maestro: Balancing fairness, latency and throughput in the openflow control plane," Tech. rep., Rice University2011.
[28] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On Controller Performance in Software-Defined Networks," Hot-ICE, vol. 12, pp. 1-6, 2012.
[29] A. Voellmy and J. Wang, "Scalable software defined network controllers," in Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication, 2012, pp. 289-290.
[30] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, et al., "Onix: A Distributed Control Platform for Large-scale Production Networks," in OSDI, 2010, pp. 1-6.
[31] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in Proceedings of the first workshop on Hot topics in software defined networks, 2012, pp. 19-24.
[32] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," in Proceedings of the 2010 internet network management conference on Research on enterprise networking, 2010, pp. 3-3.
[33] A. S.-W. Tam, K. Xi, and H. J. Chao, "Use of devolved controllers in data center networks," in Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on, 2011, pp. 596-601.
[34] http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf, "NVIDIA Tesla Kepler-Family-Datasheet."
[35] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," Queue, vol. 6, pp. 40-53, 2008.
[36] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," ACM SIGCOMM Computer Communication Review, vol. 41, pp. 195-206, 2011.
[37] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "SSLShader: Cheap SSL Acceleration with Commodity Processors," in NSDI, 2011.
[38] A. Carter, "Do it green: Media interview with Michael Manos," Dec. 2007 [Online]. Available: htt p://edge. technet. com/Media/Doing-ITGreen, 2007.
[39] L. Rabbe, "Powering the Yahoo! network," Nov, 2006.
[40] J. Snyder, "Microsoft: datacenter growth defies Moore’s law," PC-World, 2007.
[41] S. E. Arnold, Google Version 2.0: The Calculating Predator: Infonortics, 2007.
[42] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," ACM SIGCOMM Computer Communication Review, vol. 38, pp. 63-74, 2008.
[43] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, et al., "VL2: a scalable and flexible data center network," in ACM SIGCOMM computer communication review, 2009, pp. 51-62.
[44] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference, 2009, pp. 202-208.
[45] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," ACM SIGCOMM Computer Communication Review, vol. 41, pp. 266-277, 2011.
[46] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in Proceedings of the first workshop on Hot topics in software defined networks, 2012, pp. 1-6.
[47] D. B. Kirk and W. H. Wen-mei, Programming massively parallel processors: a hands-on approach: Newnes, 2012.
[48] M. Harris, "Optimizing parallel reduction in CUDA," NVIDIA Developer Technology, vol. 2, 2007.

Muhammad Imran received the B.S. degrees in Computer Science from Bahauddine Zakariya University Multan, Pakistan in 2005, M.Sc. degree in Computer Engineering form Center of Advance Studies in Engineering, Islamabad, Pakistan in 2008. He is currently Ph.D. scholar at Center for Advance Studies in Engineering with research in the field of Software Defined Networks.