GPU Implementation of the Offline Stage of DORA Algorithm for SDN

AbdulRahman AlHamali
Department of Electrical and Computer Engineering
American University of Beirut
Beirut, Lebanon
aaa149@mail.aub.edu

***Abstract*— several methods have been proposed to implement dynamic routing on the controller side in Software Defined Networks. However, this represents extra load on the controller that is already considered a performance bottleneck in the system. To mitigate this load, this research presents a GPU implementation of the offline stage on one of the proposed dynamic routing algorithms, the DORA algorithm.**

Keywords—GPU, SDN, Dynamic Routing, DORA

I. Introduction

In traditional networks, routing intelligence and decision making is carried out on the same plane on which data forwarding is done. This consumes the already-overloaded resources of the network, and limits the network's ability to make the best routing decision, due to the lack of a global view. In Software Defined Networks (SDN), intelligence is offloaded to a single, centralized controller that has a global view of the network. This allows the network resources to focus only on data forwarding, and allows us to make better routing decisions because of our global knowledge of the network. In addition, the continuous collection of network statistics, which the controller carries out, allows us to respond to changes and find new routes much faster than in traditional networks.

A. *Dynamic Routing on the Controller*

Several approaches have been proposed to implement dynamic routing on the SDN controller.

Tomovic, Lekic, Radusinovic, and Gardasevic (2016) compare 3 approaches of centralized routing to distributed Shortest Path First (SPF) algorithms [1]. In their implementations, they calculated routes based on the network topology and on statistics collected from the network, and proposed three algorithms: Shortest Widest Path (SWP), Minimum Interference Routing Algorithm (MIRA), and Dynamic Online Routing Algorithm (DORA). They simulated the traditional SPF algorithm as well as their proposed algorithms on different networks, and calculated the network throughput for each algorithm. They concluded that SPF algorithms gave the worst throughput, while DORA algorithm gave a significant improvement over traditional SPF.

Azzouni, Boutaba, and Pujolle (2017) proposed NeuRoute, a routing framework based on neural networks that predicts the future traffic matrix on the network based on previous statistics collected by the controller, and uses a neural network to calculate routes based on a baseline dynamic routing heuristic. The neural network achieves the same results as the heuristic, but in about 25% of the time [2].

B. *GPU Acceleration in SDN Controllers*

Centralizing the network intelligence in a single controller brings up the question of scalability. Indeed, having a single controller manage the whole network could represent a performance bottleneck. In order to address this, the industry has mostly headed in the direction of horizontal scaling, that is, designing the controller in a way that the load could be mitigated by adding more machines. However, some proposals have been made to use GPUs to scale the system vertically.

Renart, Zhang, and Nath (2015) propose a GPU-based SDN controller [3]. In this controller, packets are collected, sent as a batch to the GPU for processing, and then returned to the CPU. In order to limit control divergence which could result from differences in packet sizes, packets are sorted by size before they are sent to the GPU. The paper, however, does not address divergence that could result from differences in packet contents. In addition, it produces latency for single packets, because they have to wait until a large-enough batch has been received to be offloaded to the GPU.

In his thesis, Theobald (2014) presents Bonfire, a GPU-accelerated SDN controller [4]. In this controller, the datastore is stored on the memory of the GPU instead of the RAM, and queries are carried out on it, in a fashion similar to GPU databases. For routing, the controller uses a modified version of Dijkstra's algorithm to calculate the shortest paths between different nodes. His results show that GPU performs worse the CPU for smaller network, but becomes better by orders of magnitude for larger networks.

The Remainder of the paper is organized as follows: Background information about the problem, followed by the proposed GPU implementation to solve the problem, and finally, the experimental setup and the results.

II. Background

A. *DORA Algorithm*

Dynamic Online Routing Algorithm (DORA) was originally proposed by Boutaba, Szeto and Iraqi in 2002, for traffic engineering in MPLS networks [5]. The main purpose of DORA was to establish bandwidth-guaranteed paths between different sources and destinations, in a way that "[allows] more future paths to be accepted into the network and to balance the traffic load".

In order to achieve that, DORA has two stages, an offline stage that is carried out in the beginning, or upon link failures, etc. and online stage that is carried out whenever a request to establish a path between a source-destination (SD) pair is received.

In the offline stage, DORA creates a Path Potential Value (PPV) array for each SD pair. This PPV array contains a value for each link in the network. This value represents how possible it is for this link to be used by other SD pairs in the future. In order to calculate the PPV, for an SD pair, we calculate all disjoint paths between *S* and *D*. This could be done, for example, by running Breadth-First Search (BFS) to find the shortest path from *S* to *D*, then saving all the links of this path in a side array, and removing them from the graph, then repeating again until *D* is not reachable from *S* anymore. Finally, we calculate PPV for each SD-pair as follows

- For a given link *L*, and an SD-pair *X*, the value of *PPV[L]* in pair *X* starts at 0
- If *X* uses *L*, we decrement *1* from *PPV[L]*
- For each other SD-pair that uses *L*, we increment *PPV[L]* by *1*
- After calculating the PPV for all the links, normalize the value between 0 and 100

This ends the offline stage. The online stage is carried out whenever we receive a request to establish a link between an SD pair. When this request is received, we assign a weight to each link in the network:

$$Weight[L] = PPV[L] * (1 - BWP) + RB[L] * BWP$$

Where RB is the Residual Bandwidth of the link. That is, the bandwidth of the link that remains unconsumed by some other connection (RB is also normalized between 0 and 100). The BWP is the weight that we want to assign for RB; the bigger BWP is, the more we care about the current RB, the smaller it is, the more we care about the future (represented by PPV).

After assigning the weights, we run a Dijkstra algorithm, and we assign the shortest path that has been found by the algorithm.

The offline calculation is a very expensive process, because it runs BFS for every SD pair several times. However, this is highly parallelizable on the SD pair level, because each SD pair runs independently from the rest.

III. Proposed Solution

In order to address the performance problem of the offline stage of DORA, we implemented a GPU version of the algorithm that exploits SD-pair-level parallelism to solve the problem

A. One SD Pair per block

The implementation runs two kernels, the first calculates intermediary arrays *usesLink*, and *globalPPV* that are used in the second kernel to calculate the final PPV arrays.

1) Kernel 1

For this kernel, we instantiate one kernel for each SD-pair. The number of threads in the kernel does not really matter, it could be set to the number of edges in the network, or, if the network is too big, to the maximum num of threads per block, which 1024. The kernel, demonstrated in Figure 1, copies the full network description to its own shared memory, then runs BFS multiple times on this network to find a path from the source *S* to the destination *D*. Each time it finds a path, it removes this path from its local network (stored in the shared memory), and repeats BFS to look for another path, and keeps repeating the operation until *D* is no more reachable from *S*. Each link in each found path is added to an array in the GPU's global memory called *usesLink*. This array contains *E* (number of links) entries for each SD-pair, and it specifies whether this SD-pair uses this link. Finally, for each link, if the SD-pair uses this link, we increment its corresponding entry in the *globalPPV* by 1. The *globalPPV* also contains *E* entries, and represents the total number of SD-pairs that use each link.

Note that the BFS implementation is run in parallel among the threads of the block, and is loosely based on Harish and Narayanan's (2007) paper [6]. The difference between our implementation and that of [6], is that ours runs the algorithm inside each block in a single kernel invocation, while theirs runs across the whole GPU, and requires multiple kernel invocations.

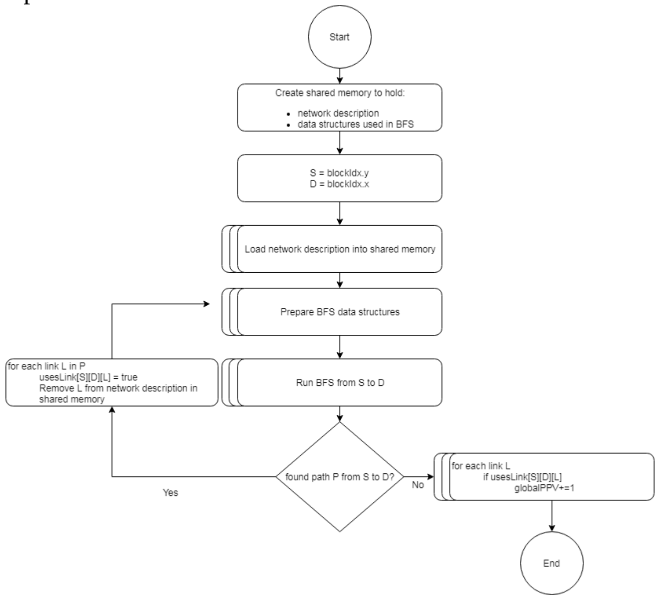
Start
Create shared memory to hold:
network description
data structures used in BFS
S = blockIdx.y
D = blockIdx.x
Load network description into shared memory
Prepare BFS data structures
Run BFS from S to D
for each link L in P
usesLink[S][D][L] = true
Remove L from network description in shared memory
found path P from S to D?
Yes
No
for each link L
if usesLink[S][D][L]
globalPPV+=1
End

Figure 1: Kernel 1 Flowchart

2) Kernel 2

The second kernel, described in Figure 2 also instantiates a block for each SD-pair, but is much simpler. It just checks whether its SD-pair uses each link *L*, if it does, *globalPPV[L] – 2* is stored into the corresponding PPV of the SD-pair, otherwise, the value of *globalPPV[L]* is simply copied there.

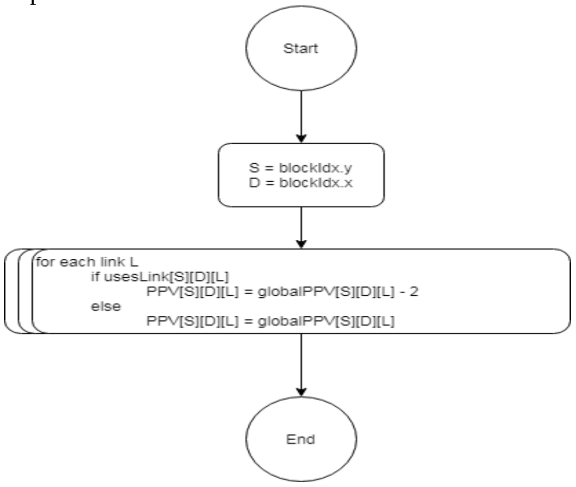

Figure 2: Kernel 2 Flowchart

B. *Network Representation*

The network graph is represented in a compact adjacency list representation. This representation is chosen for two main reasons:

- The network is copied into the shared memory of every block. Using the compact representation saves a lot of memory, especially that networks graphs tend to be sparse
- The representation is more suitable for the BFS implementation in [6]

After calculating the link usage of all the SD pairs, the results are consolidated in a single PPV array, that we will call the global PPV. The main difference between the SD-pair-wise PPV and the global PPV is that the global PPV adds 1 for each SD pair that uses the link, it does not subtract 1 at all.

After consolidating the results into the global PPV, we copy the global PPV and the link usage info back to the CPU memory, where we create the SD-pair PPVs simply by copying the global PPV, then checking if the SD pair uses a link, and if it does, subtracting 2 from the PPV of that link.

IV. Experimental Setup and Results

A. *Experimental Setup*

We tested the algorithm on 8 networks of sizes 8, 16, 32, 64, 128, 160, 176 and 192. All the networks were randomly generated with a sparsity of 70%, which means that whenever a decision was to be made on whether to have a connection between two nodes, 70% of the times the decision was not to have a link. The value of 70% is not special, but just denotes that networks are generally very sparse.

The tests were run on an Intel Xeon E5607 machine that has a clock cycle of 2.27 GHz, and 16 GB in RAM. The machine was equipped with a GeForce GTX Titan X GPU.

B. *Results*

The GPU version of the offline stage of the algorithm offered a significant speedup that reaches up to around 50x improvement over the CPU version, as demonstrated in Table 1. The speedup increases with the increase in the size of the network as shown in Figure 3. As we approach network size of 200, the simulation breaks as the GPU runs out of resources and issues an *InvalidArgument* error.

Table 1: Performance Comparison between CPU and GPU Implementations

Network Size	Number of SD-pairs	CPU Version	GPU Version	Speedup
8	64	0.001	0.002	0.5
16	256	0.004	0.002	1
32	1024	0.059	0.004	14.75
64	4096	1.073	0.048	22.354
128	16384	23.323	0.680	34.294
160	2560	66.553	1.555	42.799
176	30976	106.005	2.552	41.538
192	36864	155.204	3.13947	49.436

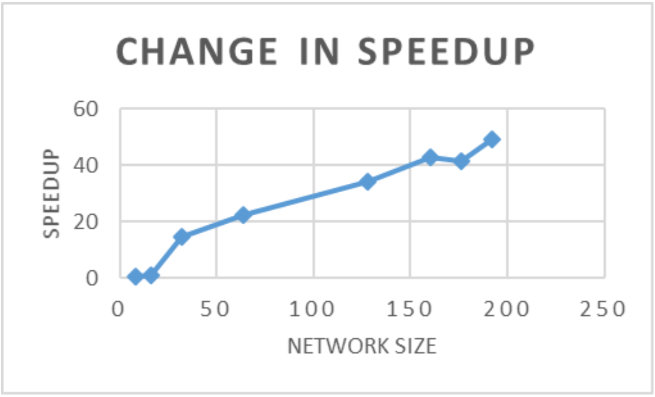

Figure 3: Change speedup with change in network size

V. Conclusion

In this research we presented a method to parallelize the offline stage of DORA algorithm on the GPU. This method offers significant improvement on the running time of the algorithm, decreasing, thus, the bottleneck problem of SDN controllers, and allowing us to run the offline stage more often. In future work, we will work on dividing the kernel into multiple streams, to allow it to run on even bigger networks. In addition, we will work on implementing the online stage of the algorithm on the GPU.

VI. Acknowledgment *(Heading 5)*

This research was carried out as a course project for EECE696: Applied Parallel Programming, in the American University of Beirut. We would like to thank Lecturer Manal Jalloul, who prepared and taught the course, and also helped with various discussions and insights on the project. In addition, we extend our gratitude to Nvidia Corporation for offering educational resources, and state-of-the-art GPUs to run our experiments on.

VII. References

[1] Tomovic, Slavica, et al. ”A new approach to dynamic routing in SDN networks.” Electrotechnical Conference (MELECON), 2016 18th Mediterranean. IEEE, 2016.

[2] Azzouni, Boutaba, and Pujolle “NeuRoute: Predictive Dynamic Routing for Software-Defined Networks” arXiv:1709.06002. Sep. 2017.

[3] Renart, Zhang, and Nath, “Towards a GPU SDN Controller,” in Networked Systems (NetSys), 2015 International Conference and Workshops on, pp. 1–5, IEEE.

[4] Theobald, “GPU Support for Software Defined Networking Controllers (Bachelor Thesis).” June, 2014. Retrieved from http://www.nt.uni-saarland.de/fileadmin/file_uploads/theses/bachelor/ThesisTobiasTheobald.pdf.

[5] Boutaba, Szeto, and Iraqi, "DORA: Efficient Routing for MPLS Traffic Engineering", in Network and Sys. Mgmt., vol. 10, no. 3, pp. 309-325, Sept. 2002.

[6] Harish, and Narayanan, “Accelerating Large Graph Algorithms on the GPU Using CUDA”, in Aluru S., Parashar M., Badrinath R., Prasanna V.K. (eds) High Performance Computing – HiPC 2007. HiPC 2007. Lecture Notes in Computer Science, vol 4873. Springer, Berlin, Heidelberg

[7] 2018. CUDA C Programming Guide, retrieved May 8, 2018, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html