View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DigitalCommons@Florida International University

Florida International University
FIU Digital Commons

School of Computing and Information Sciences | College of Engineering and Computing

12-2018

Parallel Sampling-Pipeline for Indefinite Stream of Heterogeneous Graphs using OpenCL for FPGAs

Muhammad Usman Tariq
School of Computing and Information Sciences, Florida International University, mtari008@fiu.edu

Fahad Saeed
School of Computing and Information Sciences, Florida International University, fsaeed@fiu.edu

Follow this and additional works at: https://digitalcommons.fiu.edu/cs_fac

Part of the Computer Sciences Commons

Recommended Citation
Tariq, Muhammad Usman, and Fahad Saeed. "Parallel Sampling-Pipeline for Indefinite Stream of Heterogeneous Graphs using OpenCL for FPGAs." In 2018 IEEE International Conference on Big Data (Big Data), pp. 4752-4761. IEEE, 2018.

This work is brought to you for free and open access by the College of Engineering and Computing at FIU Digital Commons. It has been accepted for inclusion in School of Computing and Information Sciences by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

Parallel Sampling-Pipeline for Indefinite Stream of Heterogeneous Graphs using OpenCL for FPGAs

Muhammad Usman Tariq
School of Computing and Information Sciences
Florida International University
Miami FL 33199 USA
Email: mtari008@fiu.edu

Fahad Saeed*
School of Computing and Information Sciences
Florida International University
Miami FL 33199 USA
Email: fsaeed@fiu.edu

Abstract—**In the field of data science, a huge amount of data, generally represented as graphs, needs to be processed and analyzed. It is of utmost importance that this data be processed swiftly and efficiently to save time and energy. The volume and velocity of data, along with irregular access patterns in graph data structures, pose challenges in terms of analysis and processing. Further, a big chunk of time and energy is spent on analyzing these graphs on large compute clusters and/or data-centers. Filtering and refining of data using graph sampling techniques are one of the most effective ways to speed up the analysis. Efficient accelerators, such as FPGAs, have proven to significantly lower the energy cost of running an algorithm. To this end, we present the design and implementation of a parallel graph sampling technique, for a large number of input graphs streaming into a FPGA. A parallel approach using OpenCL for FPGAs was adopted to come up with a solution that is *both* time- and energy-efficient. We introduce a novel graph data structure, suitable for streaming graphs on FPGAs, that allows time- and memory-efficient representation of graphs. Our experiments show that our proposed technique is 3x faster and 2x more energy efficient as compared to serial CPU version of the algorithm.**

I. Introduction

Graphs, due to their inter-connecting nature, are one of the most commonly used data structures to represent large complex data. Graphical representation makes it possible to interpret and process the data using graph analysis techniques [4]. High throughput data generation technologies in the field of bioinformatics e.g. Next Generation Sequencing in Genomics [23], De novo Peptide Sequencing in Proteomics [21], [30] and fMRI scans in Connectomics [32] are the norm now a days. A big majority of algorithms that run on this data for sequence alignment, protein identification or investigating brain networks, require this data to be represented in the form of graphs. Graph data structures are also the backbone of storing and processing huge amounts of data in Social Networks, IoT Networks, and the World Wide Web. These huge data-sets tend to be in the form of a large number of graphs with each graph representing a subset.

Unfortunately, due to the extremely large size of data it's nearly impossible to run an algorithm on the entire data set at least in a reasonable amount of time. Moreover, it might not be possible to access data in its entirety e.g. the Facebook graph or it might be only available through crawling e.g. world wide web [6]. Graph sampling can be an excellent solution to speed up the process by reducing the amount of data or make up for its unavailability. Sometimes, it is inadvisable to run an algorithm on a complete dataset as it might be noisy and yield incorrect results [2], [22]. In cases like this, sampling becomes a necessary task to filter out the noise and only keep valuable data points. For example, in the field of proteogenomics, the intermediate protein database created via six-frame translation contains identifiable proteins in less than 1% of its size [26]. In proteomics, MS/MS spectra generated through mass-spectrometry are highly noisy or contain peaks that don't contribute towards peptide identification and only less than 10% of the total peaks are useful [2]. It is absolutely necessary to remove the noisy or useless peaks from the original data to get good quality results as well as speed up the process.

Graph sampling is a process where we randomly (or systematically) select a subset of nodes and/or edges to build a subgraph that represents the original graph in a certain set of properties. By preserving these properties we can accelerate a big class of algorithms that rely on or determine these properties [15]. This also allows a reduction in the amount of data that needs to be pro-

*Corresponding author

cessed while maintaining application specific properties e.g. degree distribution, betweenness centrality, degree exponent, rank exponent etc. Graph sampling literature is extensive and includes sampling techniques like *Selection Based Sampling* [1], *Traversal Based Sampling* [12] [14] [27] and *Reduction Based Sampling* [17].

Sequential techniques are inadequate even for the sampling of big data. Existing parallel sampling techniques are either application specific [7] [8] or they are too inaccurate [25] to be used in real applications. Moreover, there exists no sampling technique, sequential or parallel, that addresses the issue of sampling multiple graphs at the same time. One can argue that these techniques can get the job done by sampling one graph at a time but as we'll see later on, this becomes impractical for very large data sets where most of the time will be wasted on reading the input graph from the data storage device. On top of that, allocating power hungry resources for sampling, such as GPU or Intel Phi, can result in significant cost for large data centers. In addition, conventional "One Size Fits All" devices like CPUs are no longer desired as they can lead to underutilization of resources which in turn can also waste a lot of energy [11]. Majority of large-scale efforts focus on conventional devices since dedicated accelerators are difficult to program. In the past, FPGAs have had a drawback over other accelerators because of time-consuming implementations of proposed architecture using HDLs like Verilog or VHDL [28].

In this paper, we present an FPGA based parallel version of the best performing reduction based algorithm presented in [17]. These algorithms are geared toward power-law graphs as this category of graphs can correctly simulate a vast and diverse class of real-world graphs. FPGA is our choice of accelerator since FPGAs are highly energy efficient, can provide speedups for correctly designed strategy when compared with CPUs, provide a custom-made solution, and their reconfigurability allows them to be reprogrammed as per changing requirements. We chose a higher level approach by implementing our proposed strategy using Intel FPGA SDK for OpenCL. We deploy a two-level pipeline to parallelize different tasks within the sampling algorithm. On the highest level, the pipeline stages include reading graphs from the storage device and transferring them to FPGA, perform parallel sampling on multiple graphs using a lower level pipeline, and write back the sampled graph to the storage device. The lower level pipeline, for sampling multiple graphs at the same time, further divides the sampling process into three different stage which is explained in section IV. Using our novel graph data structure and careful parallel design we show that our proposed strategy is 3x faster than CPU based versions of the sampling algorithms while reducing the energy consumption by a factor of 2. We test our implementation using synthetic power-law graphs [31] that closely resemble the real world graphs.

Rest of this paper is organized as follows: We provide a detailed literature review in section II. Our contributions are outlined in section III. In section IV, we describe our novel graph data structure and multi-layered software pipeline for sampling in detail. Experimental setup, data generation, and time and energy-consumption analysis, for both FPGA and CPU, is given in section V. In section VI, we conclude this paper.

II. Background

Here we will discuss state-of-the-art graph sampling algorithms that are most commonly used for sampling of real-world graphs. Later in this section, we will give a brief overview of OpenCL for FPGAs and why it is a good choice for our problem.

A. Literature Review

Graph sampling algorithms can be divided into multiple categories based on the type of sampling algorithms or the architecture being used. Here we present the categorization based on sequential vs. parallel sampling techniques.

1) Sequential Algorithms: As we already mentioned, most of the generic graph sampling algorithms are sequential. Two most basic approaches used are Vertex Sampling and Edge Sampling. These approaches also provide a basis for other sampling approaches like Vertex Sampling with Escape or Graph Sparsification [3]. Traversal Based Sampling is a much larger class of algorithms. In this approach vertices or edges are selected by traversing one vertex/edge at a time. Another group of sampling algorithms is Sampling by Reduction where instead of selecting vertices/edges we delete them to get a subgraph that is representative of the original graph.

Below we discuss some commonly used sequential graph sampling techniques:

Vertex Sampling: In vertex sampling we pick a random set of vertices V_s such that $V_s \subseteq V$. The sampled graph is actually induced subgraph G_s of the original graph G where we only keep edges that connect the vertices in the induced subgraph i.e. $E_s = (u, v | u \in V_s, v \in V_s)$. Minimum information is required about the graph as the vertex selection is made randomly.

Edge Sampling: In edge sampling we pick random set of edge E_s such that $E_s \subseteq E$. The only vertices that we keep are the ones that lie on either end of the edges in E_s i.e. $V_s = u, v | (u, v) \in E_s$.

Traversal Based Sampling: In this type of sampling, we start at a random vertex and start exploring the graph by selecting its neighbors and then recursively explore the neighbors. The criteria for selecting neighbors is decided by the specific sampling technique being used. Some of the examples are Snow Ball Sampling [12], Forest Fire Sampling [20] and Respondent Driven Sampling [14].

Reduction Based Sampling: These type of graph sampling techniques go in the opposite direction of traditional ones as they remove vertices or edges from the original graph. Some of the reduction based techniques are presented in [17]. These include Deletion of Random Vertex, Deletion of Random Edge, and Deletion of Random Vertex-Edge. In contraction based techniques there are Random Edge Contraction where an edge is picked at random and contracted or Random Vertex Edge Contraction where a random vertex edge is picked and contracted.

Exploration techniques can also be categorized as reduction based techniques. Two of these, presented in [17], are Exploration by Breadth First Search and Exploration by Depth First Search.

2) Parallel Algorithms: Very few studies have been done on parallel graph sampling and the ones that exist are application specific. Here we discuss few of those sampling techniques.

Parallel Random Samplers: This graph sampling technique, originally presented in [25], is the parallel version of Respondent Driven Sampling [14] [27]. Here, we start multiple random walkers from a starting vertex that sample the graph in parallel. In addition to reduced sampling time, this technique introduces some conflicting effects. First, since multiple samplers are working at the same time we get the desired sampled graph much faster in a lesser number of hops. This reduces the effects of changes in the graph if the graph is evolving in time. On the other hand, there is some redundant sampling of nodes near the starting point which leads to some inaccurate results.

Extracting Quasi-Chordal Subgraphs: This is an application specific graph sampling technique presented in [7] [8] which extracts Quasi-Chordal Subgraphs from the original graph. The original graph is distributed among multiple processing units and each unit extracts a chordal subgraph for its part of the graph. In the next step, all the subgraphs are connected together. The targeted graphs for this sampling method are gene correlation networks where we have to maintain highly connected parts of the graph since they indicate important functional units of the gene product.

FPGA Techniques: In our previous publication [29], we implemented a strategy to do parallel sampling for a static graph using OpenCL for FPGAs. Though energy efficient, this strategy only works for a single graph in memory and does not consider the time to read the graph from the storage device, which can be considerable depending on the graph size. Therefore, sampling multiple graphs using this strategy would yield impractical time requirements.

B. OpenCL for FPGAs

OpenCL for FPGAs consists of **1)** OpenCL compiler for FPGA that converts the code into a binary file that can be downloaded onto FPGA and **2)** Runtime environment that lets host(CPU) program the FPGA using the binary file through PCIe, transfer data between CPU and FPGA memory, and execute OpenCL kernels on FPGA. In this host-device environment, the parallelizable part of the program is written in OpenCL which will execute on FPGA while the rest of the tasks are taken care of by the CPU e.g. reading graphs into CPU memory and transferring of data to and from FPGA. Since these tasks are executing on different systems, they can be parallelized using a software pipeline. In addition, since each OpenCL kernel gets programmed onto the FPGA fabric at the same time, it occupies its own resources and can execute along with other kernels. This provides the opportunity of creating a separate software pipeline where multiple kernels are performing different tasks for separate graphs. More information about OpenCL for FPGA can be found at [28].

III. Our Contribution

The contribution of this paper is two folds i.e. design of a novel data structure to represent graphs in memory and implementation of a novel multi-layered pipelined technique to sample multiple graphs using the above-mentioned data structure.

A. Graph Data Structure

We develop a compact data structure to represent multiple graphs in memory with the space complexity of $O(|V| + |E|)$, where V is the vertex set and E is the edge set of graphs being represented. This data structure allows for the removal of vertex in $O(d)$ time where d is the degree of the vertex being removed. Storing multiple

graphs in one data structures allows them to be processed by the kernel at the same time.

B. Sampling Algorithm

Our sampling algorithm consists of two levels of software pipelines that allow for sampling of multiple graphs at the same time by overlapping the sampling process of one batch of graphs with the reading of next batch into the memory. Same is the case for writing one batch of graphs to the storage device. The sampling process consists of three parts: 1) Removing a small portion of random vertices, 2) finding the largest connected component using parallel BFS, and 3) deleting the smaller components. Each of these tasks is performed by a separate kernel. Since each kernel is programmed onto FPGA fabric, it can execute concurrently with other kernels. We take advantage of this by pipelining these kernels for different sets of graphs to further speed up the sampling process.

IV. Proposed Methods

A. Graph Data Structure for Parallel Processing

The sampling methodologies that we design and implement on FPGA require removing vertices and edges from the original graph in parallel. Therefore, we need a data structure which can deal with irregular memory access patterns of graphs and allow multiple threads to operate concurrently. Such a structure will be essential for efficient bandwidth utilization on an FPGA. Keeping these constraints in mind we introduce a novel data structure, which is a modified version of Compressed Sparse Row (CSR) to represent graphs in memory. We will discuss our data structure and the process involved, to remove multiple vertices and edges in detail.

Consider the graph shown in figure 1 (left) and its Compressed Sparse Row (CSR) representation (right). In CSR, vertices and edges are stored in two separate arrays V and E. Vertex numbers are represented by the indexes of V while the value stored at each index of V points to the starting location of edges of that vertex. In edge array E, we store the vertex number, for each edge, at its other end. In its simplest form, the space complexity of CSR is $O(V + E)$.

CSR represents graph data in a simple and compact format but it has a huge drawback in terms of making modifications in the graph especially deleting vertices and edges. It will require multiple reads and writes to memory and keeping track of edges for certain vertices will become difficult. This, in turn, will consume memory bandwidth essentially slowing down the entire

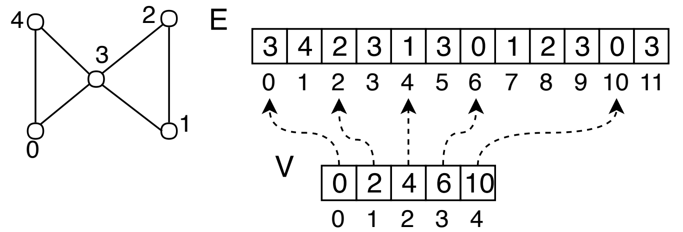

Fig. 1: Graph representation using CSR data structure.

sampling process. Alternatives to CSR, e.g. adjacency list or adjacency matrix are also not feasible. Adjacency list uses linked list to store the graph which is sequential in nature and cannot be used in a highly parallel environment. On the other hand, the adjacency matrix is too expensive in terms of space complexity i.e. $O(|V|^2)$, that it becomes impractical for large graphs. To tackle these issues, we present a novel data structure, shown in figure 2, which is space efficient i.e. $O(|V| + |E|)$ and allows for efficient removal of vertices in $O(d)$ time where d is the degree of the vertex, being removed.

1) Vertex Removal: In the original CSR data structure, adjacent vertices the array are dependent on each other in terms of knowing where the edges of current vertex end and the next ones start. Hence, removing vertices will become more and more expensive with sparsification of the graph as we'll have to traverse more entries of the array. The introduction of the second array takes care of this problem as each vertex stores its information separately and no two vertices are dependent on each other as shown in figure 2. Using two arrays, the subsequent deletion of vertices will not incur any additional removal cost.

B. Sampling Algorithm

Here we present different parts of our sampling algorithm. First, we explain the outer and inner pipelines and different stages within those pipelines. Later on, we give the kernel pseudo code for deleting random vertices in parallel (DRV) and parallel breadth-first search (BFS). The code for removing smaller components is similar to DRV as we remove vertices that are not part of the largest connected component. Simplified pseudo code for our sampling algorithm using below-mentioned pipelines and kernels is given in algorithm 2. The work-flow is explained in figure 5.

1) Outer Pipeline: The outer pipeline, shown in figure 3 (left), executes at host level where we initialize three different threads and assign each thread a different responsibility. i.e. 1) Read graph from a storage device,

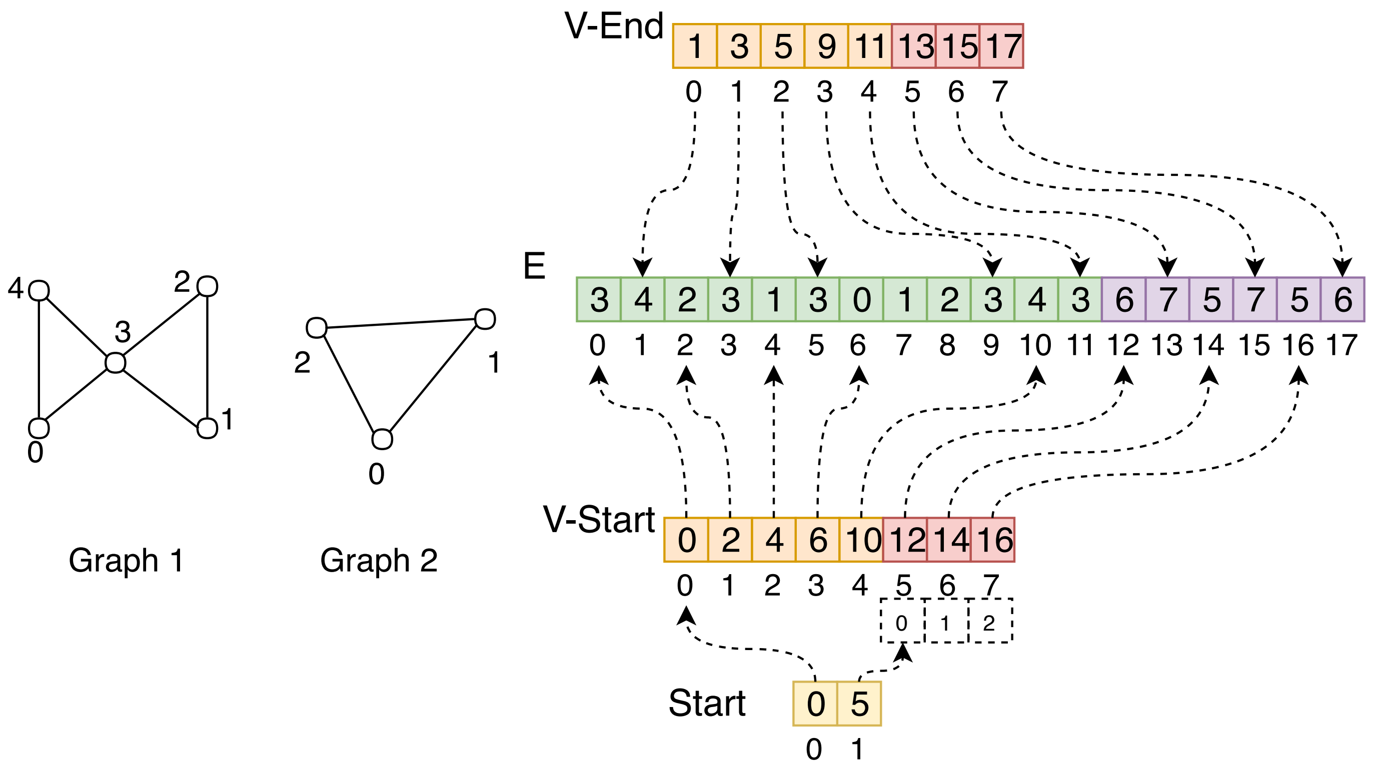

Fig. 2: Our novel graph data structure to compactly store multiple graphs in a hierarchy of arrays. The smallest array $Start$ stores the indexes for each graph. Array $V-Start$ represents vertices for each graph. Note that for the first graph, the index numbers are the same as the vertex number. But for every graph afterward, we have to subtract the starting point from the index to get the actual vertex number. This starting point value can be obtained from the $Start$ array. $V-Start$ points to the starting point of edges, stored in E, for every vertex. The addition of $V-End$ allows for efficient removal of vertices by keeping track of the last edge, in E, for every vertex.

2) Execute the lower level/inner pipeline, and 3) Write sampled graph to the storage device. To begin with, graphs are divided into batches and one batch is read by a separate thread at stage one and transferred to FPGA memory. Next, the batch is further divided into three sub-batches and each stage in the inner pipeline works on a separate sub-batch. While stage two of the pipeline is executing, stage one is reading the next batch from the storage device. Finally, the batch sampled by the second stage is written back to the storage device by stage three while stage two is sampling the second batch and stage one is reading the third batch. This process continues till no graphs are left.

2) Inner Pipeline: The inner pipeline, shown in figure 3 (right), executes at device/FPGA level, where we execute different kernels simultaneously on different sub-batches of graphs. This is a cyclic pipeline where each graph (or sub-batch) needs to go through every stage more than once, depending on the required size of the sampled graph. At the DRV stage, a small portion of vertices is randomly removed from all the graphs in the sub-batch. Next, BFS finds the largest connected components for each graph in the current sub-batch and the next sub-batch is brought in to be processed by DRV. Once, BFS is done, it provides this information to RC where it removes all the smaller components and only keeps the largest one. If the size of the largest connected component is within the error margin of the desired sample size, we move the pipeline forward, i.e. prepare the sub-batch to be written back to host memory. Otherwise, we send it back to DRV stage for further reduction of sample size. The pseudo code for different kernels, DRV and BFS, is given in algorithm 1 and algorithm 3.

3) Delete Random Vertices: Each thread removes one vertex by first deleting the entry for the given vertex and traversing its neighbors to remove all the edges connected to that vertex. The pseudo-code for the kernel is given in Algorithm 1. Each work-group works of one graph and the offset for each graph is stored in the input

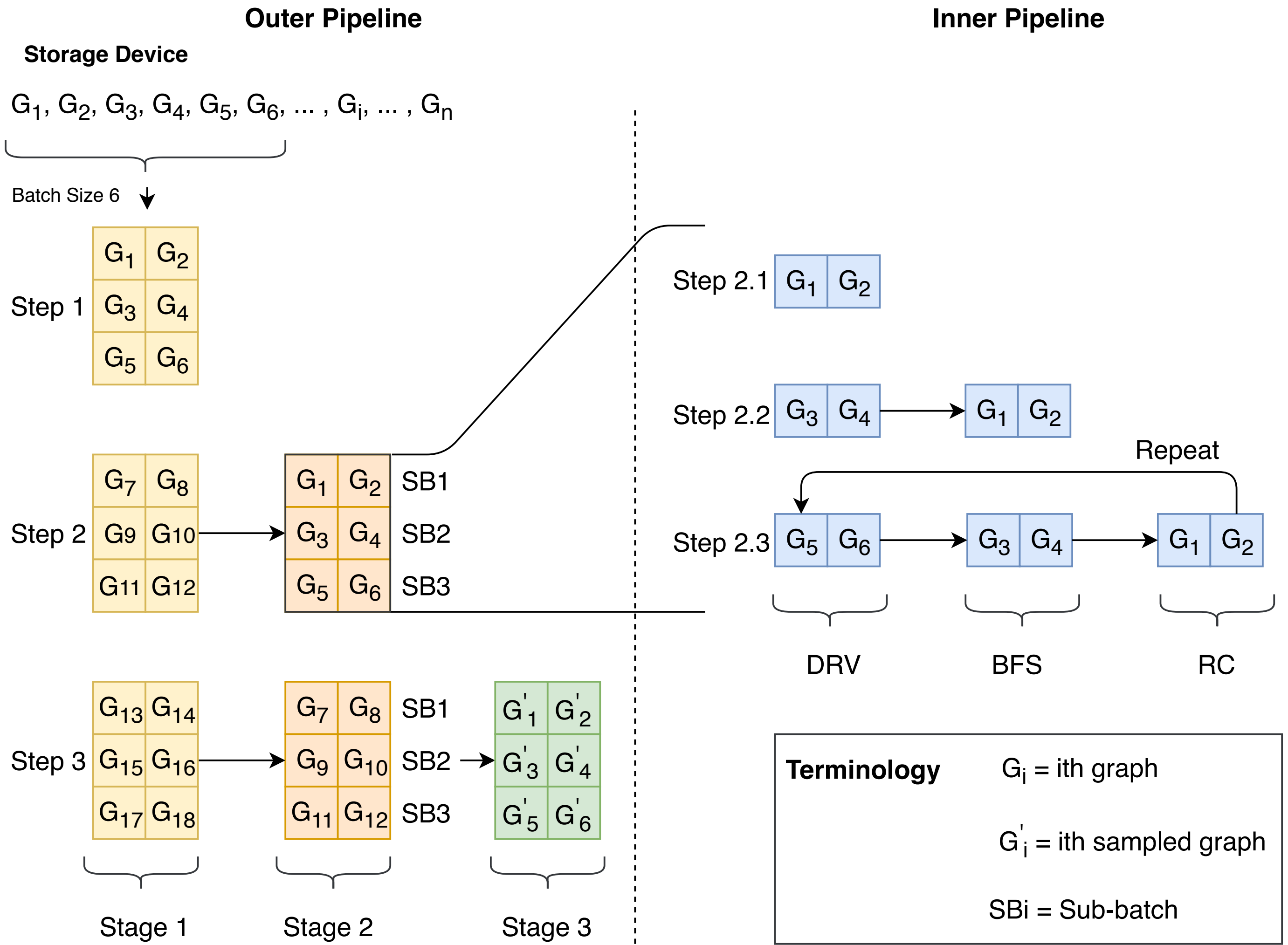

Fig. 3: Different steps during the execution of pipelines. **Outer Pipeline:** In the outer pipeline, a batch of graphs is read into the host memory. A batch of size six is considered in this example but in reality, the batch size is decided based on the device's (FPGA) available memory. Once, the batch is ready, it is transferred to the device's memory where it is sampled in the inner pipeline. **Inner Pipeline:** In the inner pipeline, each stage processes two graphs at the same time. The limit is imposed by the implementation of OpenCL for FPGA where only two work-groups can execute at the same whenever threads might finish out-of-order. Each graph goes through every stage of inner pipeline multiple time. Upon finishing, the sampled graphs are transferred back to the host where stage 3 of the outer pipeline will write the sampled graphs back to the storage device.

array *start*.

As there are nested loops in our kernel, this can degrade the performance of the kernel significantly. To overcome this hurdle, the inner loop is unrolled to reduce the number of iterations. Maximum performance is gained by unrolling the inner loop 16 times. This loop unrolling doesn't add any overhead in terms of memory accesses since the burst size of global memory used (DDR3) is 64 bytes.

There is no guarantee that no two threads will access the same memory location (rather it's very likely that two or more vertices will end up accessing the same address). We ensure that data integrity, using our strategy, is maintained by a simple observation that the value being written to any location is -1 (to indicate removal). Whenever a thread reads a value from memory, it can check that the value is not -1. If it is, we know that it is invalid i.e. the corresponding vertex or edge has already been removed and we won't perform any actions on that data.

4) Parallel Breadth First Search: We use parallel breadth-first search algorithm [13] which uses level

Algorithm 1 Delete Random Vertices (DRV)

Input: $vStart, vEnd, edges, rands, start$
1: $tid \leftarrow$ thread-id
2: $wid \leftarrow$ work-group-id
3: $v \leftarrow rands[tid]$
4: $o \leftarrow tart[wid]$
5: $sPos \leftarrow vStart[v + o], vStart[v + o] \leftarrow -1$
6: $ePos \leftarrow vEnd[v + o], vEnd[v + o] \leftarrow -1$
7: **for** $i = sPos$ to $ePos$ **do**
8: $\quad nv \leftarrow edges[i]$
9: $\quad edges[i] \leftarrow -1$
10: $\quad sPos1 \leftarrow vStart[nv]$
11: $\quad ePos1 \leftarrow vEnd[nv]$
12: $\quad$ **for** $j = sPos1$ to $ePos1$ **do**
13: $\quad\quad$ **if** $edges[j] == v$ **then**
14: $\quad\quad\quad edges[j] = -1$
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: **end for**

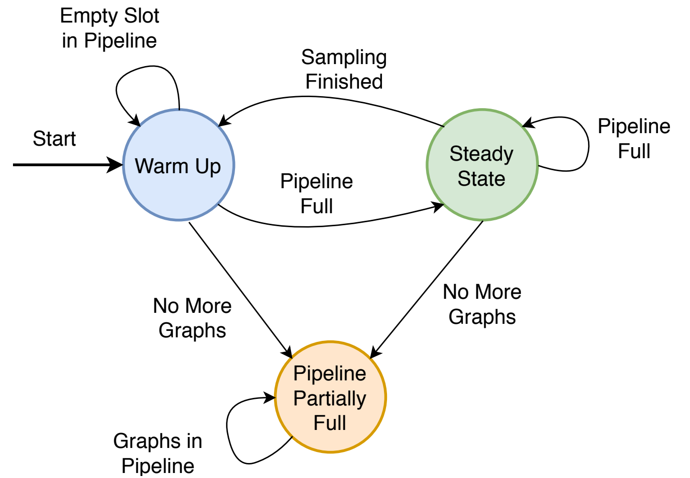

Fig. 4: Controller design for the pipeline.

TABLE I: Specifications of FPGA and CPU used in our experiments.

	DE5-Net Stratix V	Xeon W3565
Base Frequency	-	3.20 GHz
# Logical Cores	1	8
Memory	4 GB	6 GB

synchronization. Starting with a single node, each level is traversed in parallel. Instead of using a queue, the frontier is maintained in an array of size $|V|$. We initialize $|V|$ threads and each thread checks the corresponding entry in the frontier array. If the vertex is marked as the frontier, the thread loops through its neighbors (that haven't already been visited) and places them in the frontier array. The level parallelism depends on the number of vertices in the frontier at a given time. The pseudo code for BFS kernel is given in algorithm 3.

Algorithm 2 Parallel Breadth First Search

Input: $vStart, vEnd, edges, frontier, visited,$
$component, continue$
1: $tid \leftarrow$ thread-id
2: **if** $frontier[tid] == 1$ **then**
3: $\quad frontier[tid] \leftarrow 0$
4: $\quad visited[tid] = 1$
5: $\quad component[tid] = 1$
6: $\quad$ **for all** $v \in N(tid)$ {Neighbors of vertex tid} **do**
7: $\quad\quad$ **if** $visited[v] == 0$ **then**
8: $\quad\quad\quad frontier[v] = 1$
9: $\quad\quad\quad continue = true$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: **end if**

Different states of the pipeline are controlled by a controller program that runs on CPU and controls which kernel executes which batch of graphs and keeps kernels in sync. The controller state machine is shown in figure 4.

V. Results

A. Experimental Setup

Reduction based sampling by removing random vertices is best suited for power-law graphs. Power-law graph can be described as a graph where degree distribution x follows the exponential curve with a certain constant α as the exponent. Power-law has been proven to fit real world graphs accurately on numerous occasions. Different studies have shown that degree sequence of World Wide Web [19] [18] [16] and Internet router graph [10] follow power-law distributions among other applications [9] [5] [24]. For evaluation of our proposed strategy we generate different number of random graphs each on 1 million vertices in size. These graphs follow the power-law degree distribution with the exponent of 2.7 and average degree of 5. The random graph generation tool that we used is described at [31].

We implemented the sampling algorithms on Xeon W3565 and parallel version is implemented using OpenCL 1.2 on DE5-Net Stratix V GX FPGA Development Kit. Specifications of these devices are shown in Table I.

Algorithm 3 Parallel Graph Sampling

```
1:  while There are graphs to be sampled do
2:    Wait for any incomplete threads to join.
3:    Initialize a thread to read the next batch into the pipeline.
4:    Initialize a thread to write sampled graphs (if there are any) to the storage device.
5:    while There are graphs in the pipeline do
6:      Launch Kernel to remove vertices for stage 1.
7:      Launch Kernel to perform BFS for stage 2.
8:      Launch Kernel to perform component removal for stage 3.
9:      if Stage 3 complete then
10:       Shift pipeline and insert new batch at stage 1.
11:     else
12:       Rotate pipeline clockwise.
13:     end if
14:   end while
15: end while
```

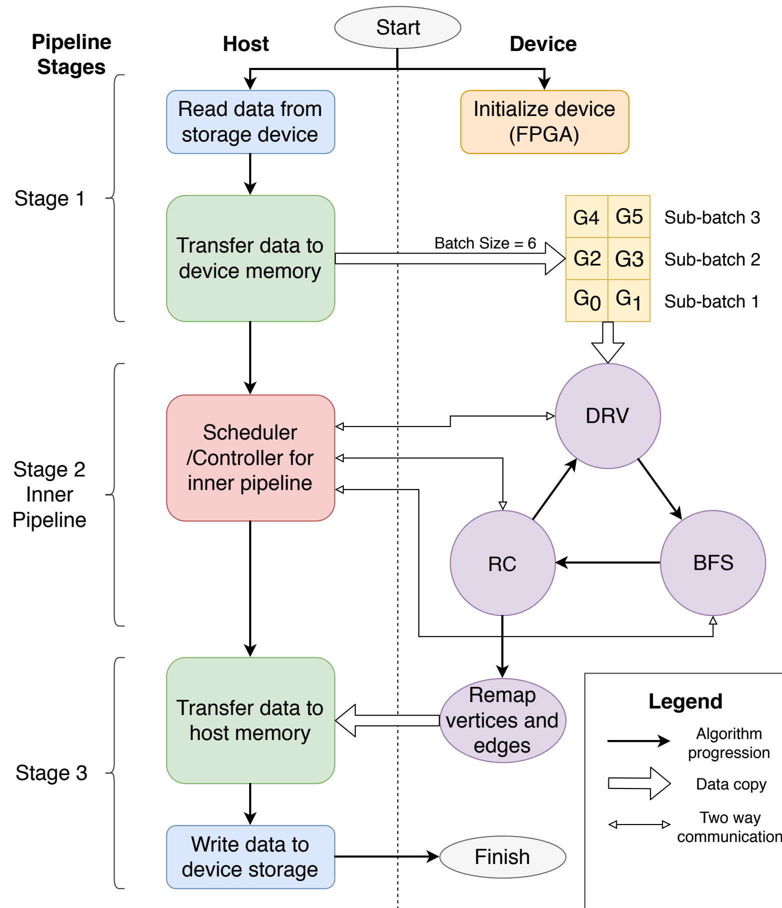

Fig. 5: Workflow for parallel sampling pipeline.

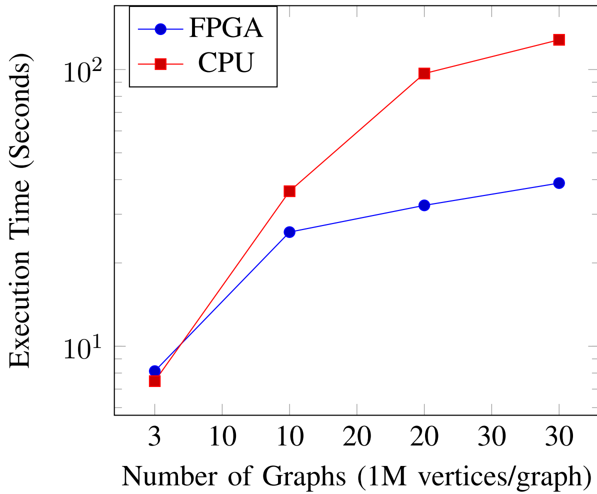

Fig. 6: Execution time for FPGA and CPU for different input size.

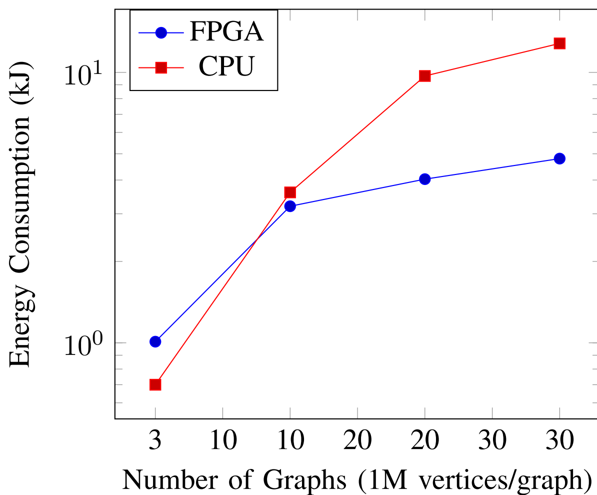

Fig. 7: Energy consumption for FPGA and CPU for different input size.

B. Timing Analysis

We implemented sampling strategies on CPU and FPGA. The results in Fig. 6 show that our FPGA based technique is faster by more than 3x as compared to algorithms running on the CPU.

C. Energy Efficiency Analysis

We measure the power consumption of our algorithm using a Watt-meter connected to the CPU and FPGA power supply. The energy consumption is then calculated by multiplying the time it takes to execute the algorithm by the power consumption. In our experiments, we found that CPU runs on the average power of 100 Watts during the execution of our algorithm while FPGA runs of 25 Watts power. Energy consumption for different techniques is given in figure 7.

D. Quality Assessment

We measure the quality of sampled graph using graph parameters such as Degree Exponent and Rank Exponent. Our analysis shows that the sampled graphs using our technique only show a difference of less than 1% when compared with the original graphs in terms of above-mentioned qualities. The two qualities are defined below:

VI. Conclusion

This paper presents a multi-layered software pipeline for the sampling of streaming multiple graphs using OpenCL for FPGAs. Upper-level pipeline parallelizes the task of reading graphs from the device storage, sampling the graphs and writing the graph back the storage device. On the other hand, the lower level pipeline consists of three stages as well i.e. deleting random vertices (DRV), running breadth-first search (BFS) algorithm to find the largest connected component, and removing the smaller components. Our results show that pipelining these processes can speed up the sampling algorithm by a factor of 3 when compared to the CPU version while being 2x more energy efficient. This is a significant energy-reduction when considered in the context of very large data-centers. Quality assessment of our sampled graphs shows that they closely match with the original graphs with the error value of less than 1%.

Acknowledgment

We would also like to acknowledge the donation of a DE5-NET-450 FPGA board from Intel Altera which was used for all FPGA based experiments performed in this paper. This material is based in part upon work supported by the National Science Foundation under Grant Number NSF CAREER ACI-1651724. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Nesreen Ahmed, Jennifer Neville, and Ramana Rao Kompella. Network sampling via edge-based node selection with graph induction. 2011.

[2] Muaaz Gul Awan and Fahad Saeed. Ms-reduce: An ultrafast technique for reduction of big mass spectrometry data for high-throughput processing. *Bioinformatics*, 32(10):1518–1526, 2016.

[3] Andras Benczur and David R Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *arXiv preprint cs/0207078*, 2002.

[4] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*, volume 290. Citeseer, 1976.

[5] Xuelian Cai. The improved weighted evolution model of the as-level internet topology. In *Information Technology and Intelligent Transportation Systems: Volume 1, Proceedings of the 2015 International Conference on Information Technology and Intelligent Transportation Systems ITITS 2015, held December 12-13, 2015, Xian China*, pages 499–508. Springer, 2017.

[6] Carlos Castillo. Effective web crawling. In *Acm sigir forum*, volume 39, pages 55–56. Acm, 2005.

[7] Kathryn Dempsey, Kanimathi Duraisamy, Hesham Ali, and Sanjukta Bhowmick. A parallel graph sampling algorithm for analyzing gene correlation networks. *Procedia Computer Science*, 4:136–145, 2011.

[8] Kathryn Dempsey, Kanimathi Duraisamy, Sanjukta Bhowmick, and Hesham Ali. The development of parallel adaptive sampling algorithms for analyzing biological networks. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 725–734. IEEE, 2012.

[9] Xiaotie Deng, Zhe Feng, and Christos H Papadimitriou. Power-law distributions in a two-sided market and net neutrality. In *International Conference on Web and Internet Economics*, pages 59–72. Springer, 2016.

[10] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM computer communication review*, volume 29, pages 251–262. ACM, 1999.

[11] Jeremy Fowers, Greg Brown, John Wernsing, and Greg Stitt. A performance and energy comparison of convolution on gpus, fpgas, and multicore processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):25, 2013.

[12] Leo A Goodman. Snowball sampling. *The annals of mathematical statistics*, pages 148–170, 1961.

[13] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International Conference on High-Performance Computing*, pages 197–208. Springer, 2007.

[14] Douglas D Heckathorn. Respondent-driven sampling: a new approach to the study of hidden populations. *Social problems*, 44(2):174–199, 1997.

[15] Pili Hu and Wing Cheong Lau. A survey and taxonomy of graph sampling. *arXiv preprint arXiv:1308.5865*, 2013.

[16] Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S Tomkins. The web as a graph: measurements, models, and methods. In *International Computing and Combinatorics Conference*, pages 1–17. Springer, 1999.

[17] Vaishnavi Krishnamurthy, Michalis Faloutsos, Marek Chrobak, Jun-Hong Cui, Li Lao, and Allon G Percus. Sampling large internet topologies for simulation purposes. *Computer Networks*, 51(15):4284–4302, 2007.

[18] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Extracting large-scale knowledge bases from the web. In *VLDB*, volume 99, pages 639–650, 1999.

[19] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the web for emerging cyber-communities. *Computer networks*, 31(11):1481–1493, 1999.

[20] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM, 2005.

[21] Bin Ma, Kaizhong Zhang, Christopher Hendrie, Chengzhi Liang, Ming Li, Amanda Doherty-Kirby, and Gilles Lajoie. Peaks: powerful software for peptide de novo sequencing by tandem mass spectrometry. *Rapid communications in mass spectrometry*, 17(20):2337–2342, 2003.

[22] Alexey I Nesvizhskii. Proteogenomics: concepts, applications and computational strategies. *Nature methods*, 11(11):1114, 2014.

[23] Petr Novák, Pavel Neumann, and Jiří Macas. Graph-based clustering and characterization of repetitive sequences in next-generation sequencing data. *BMC bioinformatics*, 11(1):378, 2010.

[24] M Olmedilla, M Rocío Martínez-Torres, and SL Toral. Examining the power-law distribution among ewom communities: a characterisation approach of the long tail. *Technology Analysis & Strategic Management*, 28(5):601–613, 2016.

[25] Amir Hassan Rasti, Mojtaba Torkjazi, Reza Rejaie, Nick Duffield, Walter Willinger, and Daniel Stutzbach. Respondent-driven sampling for characterizing unstructured overlays. In *INFOCOM 2009, IEEE*, pages 2701–2705. IEEE, 2009.

[26] Brian A Risk, Wendy J Spitzer, and Morgan C Giddings. Peppy: proteogenomic search software. *Journal of proteome research*, 12(6):3019–3025, 2013.

[27] Matthew J Salganik and Douglas D Heckathorn. Sampling and estimation in hidden populations using respondent-driven sampling. *Sociological methodology*, 34(1):193–240, 2004.

[28] Deshanand Singh. Implementing fpga design with the opencl standard. *Altera whitepaper*, 2011.

[29] Usman Tariq, Umer I Cheema, and Fahad Saeed. Power-efficient and highly scalable parallel graph sampling using fpgas. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. IEEE, 2017.

[30] Ngoc Hieu Tran, Xianglilan Zhang, Lei Xin, Baozhen Shan, and Ming Li. De novo peptide sequencing by deep learning. *Proceedings of the National Academy of Sciences*, 114(31):8247–8252, 2017.

[31] Fabien Viger and Matthieu Latapy. Random generation of large connected simple graphs with prescribed degree distribution. In *11th International Conference on Computing and Combinatorics. Kunming, Yunnan, Chine*, 2005.

[32] Jinhui Wang, Xindi Wang, Mingrui Xia, Xuhong Liao, Alan Evans, and Yong He. Gretna: a graph theoretical network analysis toolbox for imaging connectomics. *Frontiers in human neuroscience*, 9:386, 2015.