4-2018

Power-Efficient and Highly Scalable Parallel Graph Sampling using FPGAs

Tariq

Power-Efficient and Highly Scalable Parallel Graph Sampling using FPGAs

by

Muhammad Usman Tariq

A thesis submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
Electrical and Computer Engineering
Western Michigan University
April 2018

Thesis Committee:

Fahad Saeed, Ph.D., Chair
Bradley J. Bazuin, Ph.D.
Lina Sawalha, Ph.D.

Power-Efficient and Highly Scalable Parallel Graph Sampling using FPGAs

Muhammad Usman Tariq, M.S.E.

Western Michigan University, 2018

Energy efficiency is a crucial problem in data centers where big data is generally represented by directed or undirected graphs. Analysis of this big data graph is challenging due to volume and velocity of the data as well as irregular memory access patterns. Graph sampling is one of the most effective ways to reduce the size of graph while maintaining crucial characteristics. This thesis presents design and implementation of a field programmable gate array (FPGA) based graph sampling method which is both time- and energy-efficient. This is in contrast to existing parallel approaches which include memory-distributed clusters, multicore and GPUs. Our strategy utilizes a novel graph data structure, that we call COPRA which allows time- and memory-efficient representation of graphs suitable for reconfigurable hardware such as FPGAs. Our experiments show that our proposed techniques are 2x faster and 3x more energy efficient as compared to serial CPU version of the algorithm. We further show that our proposed techniques give comparable speedups to graphical processing unit (GPU) and multi-threaded CPU architecture while energy consumption is 10x less than GPU and 2x less than CPU.

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor Dr. Fahad Saeed for his continuous help, support, and guidance in completing my research and keeping me motivated during the period of my Master's program. It wouldn't have been possible for me to conduct this research without his unparalleled skills and knowledge on the topic. It has been an experience of lifetime working with him. His relics of wisdom will always enlighten my path and guide me in my future endeavors. I would also like to acknowledge my committee members Dr. Bradley J. Bazuin and Dr. Lina Sawalha for serving on my thesis committee. I'm also thankful to Umer I. Cheema for providing help on understanding FPGAs. This material is based in part upon work supported by the National Science Foundation under Grant Numbers NSF CRII CCF-1464268 and NSF CAREER ACI-1651724.

I want to thank my parents and siblings for always believing in me and providing me with moral support.

I want to acknowledge my friends and colleagues in Parallel Computing and Data Sciences Lab, Sandino, Muaaz, Taban, Haseeb who were always there and never hesitated to provide any technical help whenever I needed it.

Muhammad Usman Tariq

TABLE OF CONTENTS

Table of Contents—Continued

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Graph sampling is a process to select a subset of vertices and edges to obtain a smaller graph that represents the original graph in certain properties. The representativeness depends on numerous properties that vary depending upon the applications including graph visualization [1], surveying hidden population [2], network analysis [3] etc. Extracting information through graph analysis is an essential process used in many real-world applications. Running simulations and analytics on massively huge graphs is impractical and cost-prohibitive [4]. Graph sampling allows reduction in the amount of data that needs to be processed while maintaining application specific properties e.g. degree distribution, betweenness centrality, degree exponent, rank exponent etc. Graph sampling literature is extensive and includes numerous techniques which are divided into three broad categories: *Selection Based Sampling* [5], *Traversal Based Sampling* [6][7][2] and *Reduction Based Sampling* [3].

1.1. Motivation

In this section, we discuss some of the motivations for our research that include the choice of implementing parallel sampling algorithms, using accelerators instead of generic CPUs and adopting OpenCL approach over conventional HDLs for programming FPGAs.

1.1.1. Big Data Graphs

The amount of data being generated is exploding in a manner unprecedented by anything ever before. According to a report published in 2011 [8] by International Data Corporation (IDC), the total size of stored data in the world was 1.8 zeta-bytes (1.8 trillion gigabytes)

and predicted to grow by 9 folds in the next 5 years. In the world of social networks, as of the fourth quarter of 2017, the total number of monthly active Facebook users worldwide exceeded 3-billion [9], while twitter had over 330-million active monthly users [10]. A major portion of this data expansion is contributed by social networks, Internet of Things (IoT) and World Wide Web (WWW) which employ graphs as the base of storing network information. It is without any doubt that the amount of graph data will keep increasing over the coming years and there is significant need of new technologies to manage, analyze and process this data in an efficient manner. Graph sampling is one of the most effective ways to speedup existing graph algorithms and reduce data sizes at the same time.

1.1.2. Parallel Sampling for Generic Graphs

Most of the existing algorithms for graph sampling are sequential and impractical for sampling of big data graphs, e.g; Breadth First Sampling [11], Edge Sampling, Snow Ball Sampling [6], Metropolis-Hastings Random Walk [12], Forest Fire Sampling [13], and Respondent Driven Sampling [7], etc. Some of these algorithms have the potential for parallelism but they are too inaccurate to produce any useful results. Traversal Based Sampling algorithms are inherently sequential as the selection of next vertex relies on already selected parent vertex, e.g., Snow-Ball Sampling and Forest Fire Sampling. In the parallel domain, only a few studies have been done for sampling graphs which are either application specific [14][15] or too inaccurate [16] to be used in real applications. While the amount of graph data being generated has been increasing rapidly over the recent past, and the growth rate will only keep increasing, the existing techniques are failing to keep up. This creates an urgent need to develop efficient and fast, high performance parallel sampling algorithms, which can tackle the ever increasing demand for processing power and also scale linearly with increasing data sizes, in energy and speedup.

1.1.3. Energy Efficient Accelerators

Recent trend in the development of high performance graph algorithms has mostly been towards sheer speedup using accelerators like GPUs or multi-core processor with only a little if no concern for the amount of energy demanded by them. 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 under utilization of resources which in turn can also waste a lot of energy [17]. The majority of large-scale efforts focus on conventional devices since dedicated accelerators are difficult to program. In the past, FPGAs have had a draw back over other accelerators because of time consuming implementations of proposed architecture using HDLs like Verilog or VHDL [18]. With the introduction of OpenCL for FPGAs, development time is no longer a bottleneck which makes it an ideal choice over conventional accelerators and generic CPUs [18].

1.2. Our Approach

This thesis presents an FPGA based parallel version of the best performing reduction based algorithms [3], published in our paper [19], geared towards power-law 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 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. There are two major parts to our code: 1) Host Side code that executes on a CPU and 2) Device/accelerator code that executes on a FPGA. The host is responsible for writing data to a device, scheduling kernel execution, and reading back results when

kernel execution is complete. Kernels are executed on a device. Using our novel graph data structure and careful parallel design we show that our proposed strategy is able to compete in terms of speed with both CPU and GPU based versions of the sampling algorithms while reducing the energy consumption by a factor of 3. We test our implementation on synthetic power-law graphs that closely resemble real world graphs.

We use a novel data structure called Compressed Parallel-Removal Array (COPRA) to represent a graph in memory. Data is represented in terms of arrays of vertices and edges and no pointers are required as opposed to adjacency list. Since, all the data is in the form of arrays, the parallel processing of a graph becomes easier as arrays can be directly indexed in memory. Nevertheless, graph processing still posses a challenge as memory accesses are irregular and this can potentially slow down the whole process.

CHAPTER 2

BACKGROUND AND OUR CONTRIBUTIONS

Here we give a detailed description of state-of-the-art algorithms for graph sampling in sequential and parallel domains and OpenCL implementation for FPGAs. At the end of this chapter we'll list the contributions made towards graph sampling by this thesis.

2.1. Terminology

Before we dig into those algorithms let's first define some terminology that we'll use here and in later sections of the thesis.

Let's say we have a graph $G(V, E)$ with vertex set V and edge set $E = (u, v)|u \in V, v \in V$. We define sampled graph $G_s(V_s, E_s)$ with vertex set $V_s \subseteq V$ and edge set $E_s \subseteq E$ and $E_s = (u, v)|u \in V_s, v \in V_s$.

2.2. Literature Review

We will discuss graph sampling algorithm that are already being used. These algorithms can divided into two broad categories as: **1)** Sequential Algorithms and **2)** Parallel Algorithms.

2.2.1. Sequential Algorithms

As we already mentioned, most of the generic graph sampling algorithms are sequential. The two most basic approaches used are Vertex Sampling and Edge Sampling. These approaches also provide the basis for other sampling approaches like Vertex Sampling with

Escape or Graph Sparsification [20]. 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. We will discuss a few examples from this category. Another group of sampling algorithms is Sampling by Reduction where instead selecting vertices/edges we delete them to get a sub graph that is representative of the original graph.

Below we discuss some commonly used sequential graph sampling techniques:

1. **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 an induced subgraph G_s of the original graph G where we only keep edges $E_s = (u, v | u \in V_s, v \in V_s)$. In this type of sampling we only require minimum information about the graph as the selection of vertices is done randomly.

2. **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 $V_s = u, v | (u, v) \in E_s$. As mentioned in [21], this definition only arises in theoretical discussions and a more realistic version of this algorithm would be to let $V_s = V$. Another version of edge sampling is graph (edge) sparsification.

3. **Traversal Based Sampling:** In this type of sampling we start with one or more vertices and crawl the graph selecting one or more vertices at each step. The criteria on which we select these vertices depends on the type of application and what kind of graph properties we want to preserve. Some of the examples are Snow Ball Sampling [6], Forest Fire Sampling [13] and Respondent Driven Sampling [7]. We'll discuss them in some details below and list some other methods for reference.

    - **Snow Ball Sampling:** SBS was first proposed in [6]. In this sampling technique, we start with some initial set of vertices probably chosen at random and at each

step we ask vertices to name k other vertices. This process is continued for a desired number of stages and then terminated. This type of sampling is used in finding hidden population, e.g., drug abusers, etc. Snow Ball Sampling is also called Network Sampling or Chain Referral Sampling. This technique is very similar to BFS except in BFS we expand the whole neighborhood while in SBS we only pick a limited number of vertices.

- **Forest Fire Sampling [6]:** This technique is similar to SBS as we select a certain number of neighbor at each step. But in FFS the neighbors are selected probabilistically. FFS can be converted to SBS by setting the probability of neighbor selection to $p = \frac{1}{k}$.
- **Respondent Driven Sampling:** RDS was first presented in sociology studies to perform estimation on hidden population, e.g., [7][2]. Recently there has been a trend shift and this technique is gaining popularity. The sampling process is similar to SBS except we adjust the bias of each vertex according to their sampling probability. Authors in [22] also termed this technique as Re-Weighted Random Walk (RWRW).

4. **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 [3]. As discussed in one of the sections above, these include Deletion of Random Vertex (DRV), Deletion of Random Edge (DRE), Deletion of Random Vertex Edge (DRVE). Furthermore, there is hybrid technique that uses a combination of DRE and DRVE. At each step DRVE is executed with probability β and DRVE is executed with probability $(1 - \beta)$. 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 [3] are Exploration by Breadth First Search and Exploration by Depth First Search.

2.2.2. Parallel Algorithms

A 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.

1. **Parallel Random Samplers:** This graph sampling technique, originally presented in [16] is the parallel version of Respondent Driven Sampling [7][2]. 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 size graph much faster in 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 usually leads to some inaccurate results.

2. **Extracting Quasi-Chordal Subgraphs:** This is an application specific graph sampling technique presented in [14][15] which extracts Quasi-Chordal Subgraphs from the original graph. The original graph is distributed into multiple processing units and each unit extracts a chordal subgraph for it's part of the graph. In the next step all the subgraphs are connected together. The border edges that form a triangle with one chordal edge are selected to connect different graphs 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 gene product.

To the best of our knowledge there doesn't exist any graph sampling algorithm implemented on FPGAs using OpenCL.

2.2.3. OpenCL for FPGAs

There are two parts of OpenCL applications. One part runs on a host machine containing any type of CPU, e.g., x86, ARM or an embedded CPU like SoC. This part is responsible for writing data to a device, reading data from a device and executing kernels on a device. The host is responsible for synchronizing all the action that takes place on a device which brings us to second part of OpenCL applications, the OpenCL Kernels. Kernel code follows C syntax. Different annotations are used to specify memory locations for input parameters and other variables used in the kernel. Other attributes are used to specify the parallelism in the kernel, i.e., number of compute units and number of SIMD work items. This part runs on a device (in our case a FPGA). The OpenCL compiler compiles the kernels into a binary file that can be programmed on device and then the host can schedule execution of these kernels. The kernels compiled for FPGA differ from that of GPUs and CPUs, where different threads are running on different processing units. Kernels for FPGA are built by constructing custom pipelined hardware that can execute multiple kernels at the same time by exploiting pipeline parallelism. At a specific time during execution of the kernels different threads will be at different stages of this pipeline [18].

For example, the pipeline for the kernel code shown below would be similar to the one shown in Fig. 2.1.

```
int id = get_global_id(0);
C[id] = (A[id] << 2) + 5;
```

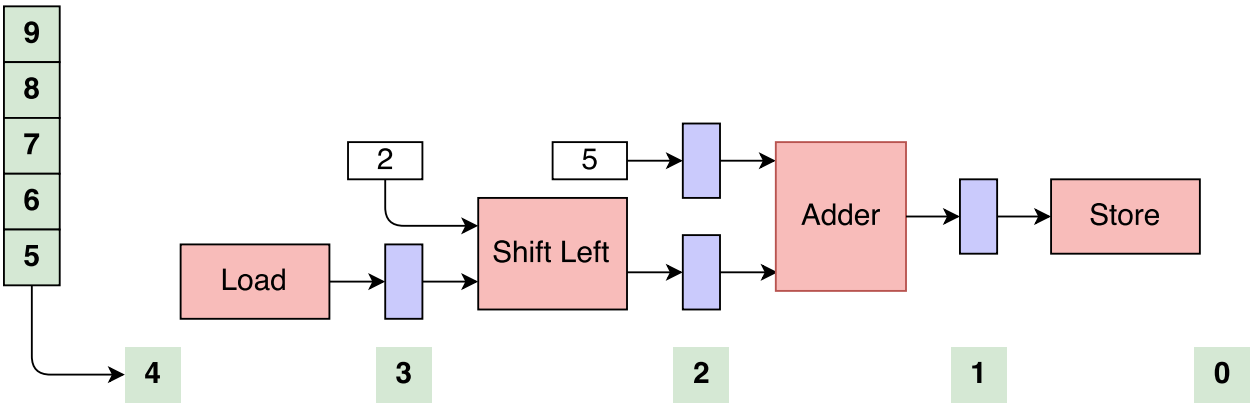

Figure 2.1: Example of custom pipeline.

2.3. Our Contributions

The contributions of the thesis are as follows:

1. We present a fast and energy-efficient graph sampling strategy using CPU-FPGA model for different sampling strategies. Three parallel sampling algorithms [3] that we implement are: **a) Delete Random Vertex (DRV)**, **b) Delete Random Edge (DRE)** and **c) Delete Random Vertex-Edge (DRVE)**. To the best of our knowledge this is the first time a graph sampling strategy has been implemented on a CPU/FPGA computing model.

2. In order to accomplish the task above, we designed and implemented a novel data structure, called Compressed Parallel-Removal Array (COPRA) that allows efficient update on graphs while conserving scarce memory bandwidth and locality resources on FPGA. Our data structure allows efficient vertex, edge and vertex-edge removal on FPGA without the need to update data/pointers on external memory or disk. This

also minimizes re-reads of graph source and target points from external memory which is the key to improving the performance.

The rest of the thesis is organized as follows: In chapter 3, we describe our work in details. We present our results and compare them with CPU and GPU versions in chapter 4. In chapter 5, we conclude the thesis and describe our future goals.

CHAPTER 3

PROPOSED METHODS

In this chapter we present our novel data structure COPRA used to represent graph in memory and later we'll discuss our sampling algorithms.

3.1. COPRA: 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 also allow multiple threads to access data points and concurrently make changes in parallel. Further, a data structure that can allow modification of the graph (as a result of sampling) without accessing external memory. Such a structure will be essential for efficient bandwidth utilization on an FPGA. Keeping these constraints in mind we introduce COPRA, a novel data structure, which is a modified version of Compressed Sparse Row (CSR) to represent graph in memory. Let us briefly discuss Compressed Sparse Row (CSR) before we get into the details of our modifications necessary for FPGA based strategy.

Assume a graph $G(V, E)$ as shown in Fig. 3.1 (left) with $V(G)$ as its vertex set and $E(G)$ as its edge set. The graph is represented using Compressed Sparse Row (CSR) structure (right). In CSR, vertices and edges are stored in two separate arrays V and E. Original vertices are represented by the indices of V. While the value stored at each index of V points to the starting location of edges of the corresponding vertex. In edge array E, for each edge, we store the vertex number at its other end. In its simplest form space complexity of CSR is $O(V + 2E)$. CSR representation of graph G is shown in Fig. 3.1 (right).

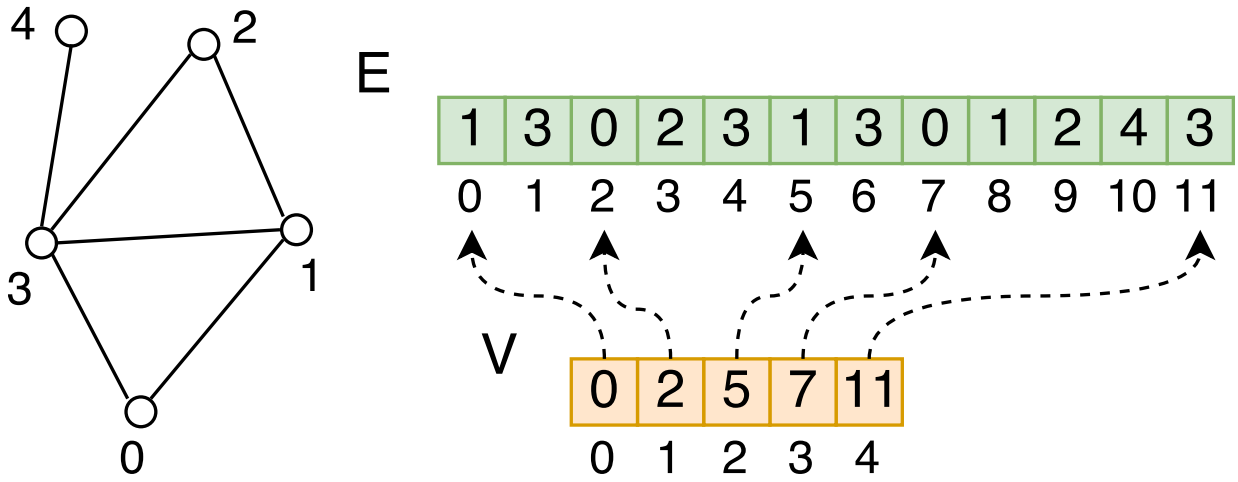

Figure 3.1: Graph G and its CSR representation

Even though CSR representation is simple and memory efficient, vertex and edge removal is not possible in its original format. Using CSR in its original form will require multiple updates to the data at external memory which will result in inefficient on-chip memory and bandwidth usage. Such a scheme will be inefficient both in terms of time and energy. Other graph data structure such as adjacency matrix of adjacency list also pose similar challenges i.e. higher memory complexity $O(V^2)$ and sequential memory accesses respectively. In order to tackle with these challenges, we introduce two major changes in CSR to allow graph modification in an efficient manner (vertex removal and edge removal). The design of COPRA is discussed below.

3.1.1. Vertex Removal

When trying to delete a vertex say vertex "1" Fig. 3.1, we'll lose some other information as to where edges of vertex "0" end in E i.e. the number of neighbors of vertex "0" will be lost. This value comes in handy when we are trying to traverse neighbors of vertex especially if most of the vertices from the original graph have been removed. To solve this problem

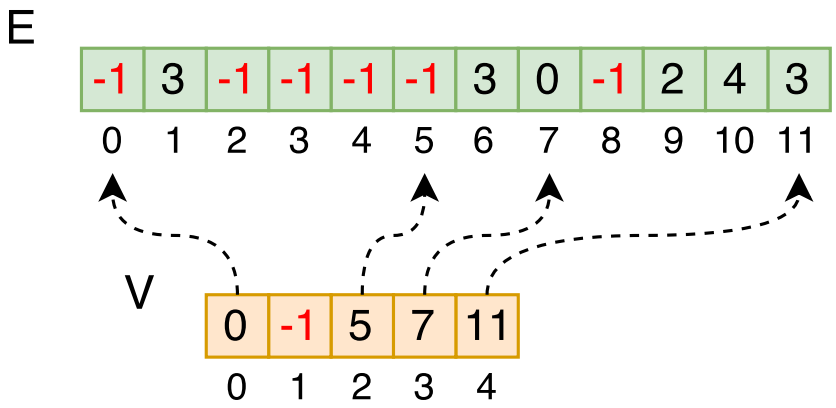

Figure 3.2: Removing a vertex from CSR graph representation causes loss of information related to other vertices. In this figure vertex "1" has been deleted and we lose information about the degree of vertex "0".

we introduce another vertex array and rename the two vertex arrays as *V-Start* and *V-End* as they keep track of starting and ending positions of their edges in E. Fig. 3.2 shows CSR graph when vertex "1" has been removed. Here, we are unable to tell if the edges connected to vertex "0" extend to index 1 or 4. On the contrary, as shown in Fig. 3.3 that even if we remove some vertices it wouldn't affect the information of any other vertex.

3.1.2. Edge Removal

The second problem is with edge removal or more specifically random edge removal. As each edge has two entries in E, one for each vertex it's incident on, in original CSR format we don't have any information available to find the other end of randomly selected edge. Suppose, we were to randomly delete the edge that connects vertex "1" and "3" (In reality we'll pick either index 6 or 15 and search for the other value). Say, the randomly selected index was 6. Looking at the value we instantly know that one side is connected to vertex "3" but to find out the vertex on the other side of this edge we'll have to traverse the vertex array and look for the reference to array E in V that's nearest to index 6. This will be

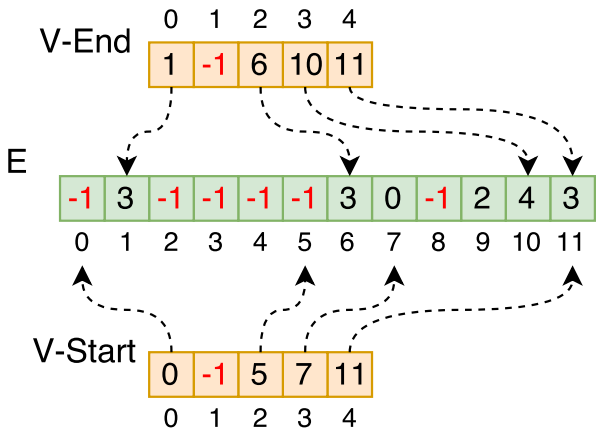

Figure 3.3: Introducing first modification in CSR i.e. adding another vertex array *V-End* to keep track of ending position of edges of each vertex in array *E*. In this way, we can remove any vertex from the original graph without worrying about lose of information i.e. we still know the degree of vertex "0" even after we have removed vertex "1" as opposed to original CSR structure.

very time consuming and become impractical in large graphs. To overcome this problem, we add additional values in edge array, i.e. at the beginning of each edge set (edges incident to one vertex) we add the vertex number (blue values). To indicate that this value is not an edge we add another flag (red values) before the vertex entry which will help us get correct vertex values from edge array. Now, let's take the same example of randomly picking index "6". After picking this value, we traverse the E backwards till we hit the value -2 (index 3). Once we are there, all we have to do is pick the next value (index 4, value "1") and that would be the vertex on the other side of this edge. Next step is fairly simple i.e. we traverse the edges of vertex "3" and look for value "1" and remove it. The final version of COPRA is shown in Fig. 3.4.

3.2. Parallel Removal of Vertices, Edges and Vertex-Edges

First step of sampling process is to remove small portion of Vertices, Edges and Vertex-Edges, depending on the sampling strategy, from the original graph. Our method is highly

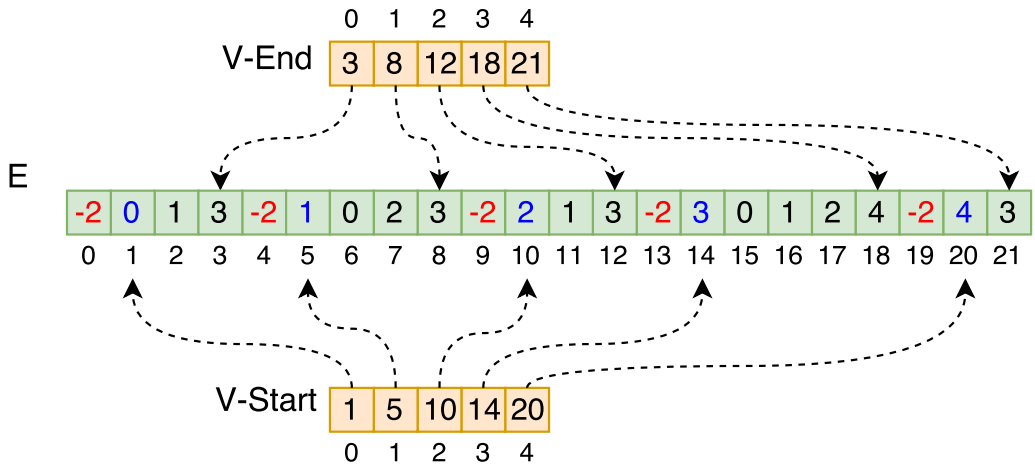

Figure 3.4: Modified CSR representation: Final version of our proposed data structure COPRA. Edge array "E" is indexed by two vertex arrays "V-Start" and "V-End" and we also have corresponding vertex number before each edge set. Space complexity of this structure is $O(4V + 2E)$.

scalable as each thread removes one vertex (edge or vertex-edge) in parallel. Even if multiple threads end up accessing the same memory location the end result of sampling still remains the same. Here we discuss three algorithms that remove vertices, edges and vertex-edges from graphs respectively.

3.2.1. Parallel Vertex Removal

Kernel pseudo-code for removing vertices is give in Algorithm 1. In the outer-loop of our algorithm we traverse the edges of the vertex v being removed. After removing one side of an edge $e = (v, u)$ we traverse the edges of the neighboring vertex u in the inner loop to remove the other side.

Algorithm 1 Remove random vertices.

Input: $vStart, vEnd, edges, rands$

```
1: tid ← thread-id
2: v ← rands[tid]
3: sPos ← vStart[v], vStart[v] ← −1
4: ePos ← vEnd[v], vEnd[v] ← −1
5: for i = sPos to ePos do
6:    nv ← edges[i]edges[i] ← −1
7:    sPos1 ← vStart[nv]
8:    ePos1 ← vEnd[nv]
9:    for j = sPos1 to ePos1 do
10:      if edges[j] == v then
11:         edges[j] = −1
12:      end if
13:   end for
14: end for
```

In OpenCL for FPGAs, nested loops can degrade performance and should be avoided if possible. In our kernel, we use loop unrolling to reduce number of iterations of inner loop. We get the maximum performance gain 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.

3.2.2. Parallel Edge Removal

Parallel Edge Removal consists of two parts: 1) Remove one side of the edge and find the vertex on the other side, 2) Remove the other side of the vertex. First part is done by traversing the edge array backwards till we find -2. After getting the vertex connected to the other side we traverse its edges and delete the reference to first vertex. This is explained in Algorithm 2

Algorithm 2 Remove random edges.

Input: $vStart, vEnd, edges, rands$

```
1: tid ← thread-id
2: e ← rands[tid]
3: v ← edges[e], edges[e] ← −1
4: i ← e − 1
5: while edges[i] ≠ −2 do
6:    i ← i − 1
7: end while
8: sPos ← vStart[v]
9: ePos ← vEnd[v]
10: for j = sPos to ePos do
11:    if edges[j] == edges[i + 1] then
12:       edges[j] = −1
13:    end if
14: end for
```

Both loops are unrolled 16 times to get the maximum performance.

3.2.3. Parallel Vertex-Edge Removal

This technique includes picking random vertex and then deleting one of the random edges from that vertex. To achieve this we need two arrays containing random values, we call these "randV" and "randE". The size of "randE" is determined based on the maximum degree D of the graph. Once, we pick a vertex v at random, we pick a value from "randE" depending on which thread we're in. As this value (say e) might be out of bound for a certain vertex with degree $d < D$ (since not each vertex is of maximum degree) we truncate this value by taking the mod of e (e mod d). After this, deleting the other end of this edge is straight forward since we already know which vertex v this edge e belongs to. Pseudo code is given in Algorithm 3.

Algorithm 3 Remove random vertex-edges.

Input: $vStart, vEnd, edges, randV, randE$

1: $tid \leftarrow$ thread-id
2: $v \leftarrow randV[tid]$
3: $sPos \leftarrow vStart[v]$
4: $ePos \leftarrow vEnd[v]$
5: $d \leftarrow ePos - sPos$ {vertex degree}
6: $e \leftarrow randE[tid] \mod d$
7: $vn \leftarrow edges[sPos + e]$ {neighbor of v}
8: $sPos1 \leftarrow vStart[vn]$
9: $ePos1 \leftarrow vEnd[vn]$
10: **for** $j = sPos1$ to $ePos1$ **do**
11: **if** $edges[j] == v$ **then**
12: $edges[j] = -1$
13: **end if**
14: **end for**

Loop is unrolled 16 times for maximum performance gain.

3.2.3.1. Parallel Breadth First Search One of the main bottlenecks in majority of graph algorithms is to traverse the graph to determine connectivity, search a vertex or count the number of vertices or edges. For our algorithm, we determine the connectivity and count the number of vertices in the largest component in each iteration to keep track of the sample size. The implementation is similar to sequential BFS where the graph is explored using an expanding frontier. This frontier is stored in a queue which is inherently sequential. We follow the strategy presented in [23] to replace the queue with an array of size V where each element of the array is associated with each vertex. Each thread of kernels checks every vertex in parallel in each iteration to determine the next frontier. Kernel implementation is given in algorithm 5 while the complete BFS including the communication pattern between host and device is given in algorithm 4 line $4 - 8$.

3.2.4. Data Integrity

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). Therefore, it is critical that we get consistent and accurate results even when data is accessed concurrently. Some scenarios, where this might happen is:

- In parallel DRV two threads might pick neighboring vertices to remove. These vertices will share an edge and these two threads will try to remove the same edge.
- In parallel DRE or DRVE two threads might pick same edge or vertex-edge to remove as each edge has two entries in edge array E (one for each vertex).

In above two scenarios or any other, more than one thread will try to remove a value (vertex, edge or vertex-edge) from the same memory location. 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, edge or vertex-edge has already been removed and we won't perform any actions on that data.

3.3. Sampling Algorithm

After having a memory efficient data structure to represent and update graph structures, we discuss design and implementation details of our sampling strategy on FPGA. The sampling strategies are implemented using our novel graph data structure that we presented in the section above. Our data structure is ideal for the FPGA based sampling strategy since

it allows removal of vertices, edges, and vertex-edges for a graph while conserving significant memory bandwidth. There are three major steps involved in our implementation:

1. Launch one of the three kernels described in section 3.2 based on the sampling strategy to remove a small set of vertices/edges.

2. Run parallel BFS [23] kernel (Algorithm 5) to find the largest connected component from the graph resulting from previous step.

3. Remove all the vertices that don't belong to the largest connected component. We reuse 1 here to remove vertices in parallel.

4. Repeat above three steps till we are left with a graph of required size.

This process is described in Algorithm 4. The number of iterations in the procedure described above can be significantly large depending upon the number of vertices (edge or vertex-edges) removed in one step. For first two strategies i.e. random vertex removal and random edge removal, we can remove relatively larger number of vertices/edges in each step respectively in the beginning of the process and reduce graph size exponentially i.e. halve the number of vertices/edges being removed at each step without disconnecting the graph too much. Experiments show that the optimum value for number of vertices/edges to be removed at the first step is 50% of vertices/edges to be removed. We keep reducing this number by the factor of 2 at each step till we reach 3% after which we remove 3% during each iteration as this number is small enough to get accurate resultant graph size but not too small to cause redundant iterations. Notice that this technique is only valid when removing large portion of vertices/edges doesn't disconnect the graph significantly otherwise we'll end up with multiple components of size smaller than the desired sample size.

Algorithm 4 Parallel graph sampling.

```
1: while Graph Size > Required Sample Size do
2:    Launch Kernel to Remove Vertices, Edges or Vertex-Edges.
3:    compSize ← 0
4:    while compSize < Required Sample Size do
5:       while continue == true {Run BFS} do
6:          continue ← false
7:          Launch BFS kernel described in Algorithm 5.
8:       end while
9:    end while
10:   Launch Kernel to remove all vertices that are not in Largest Connected Component.
11: end while
```

Kernel for Parallel BFS is shown in algorithm 5:

Algorithm 5 Parallel breadth first search.

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

The flow chart of the entire algorithm is given in Fig 3.5.

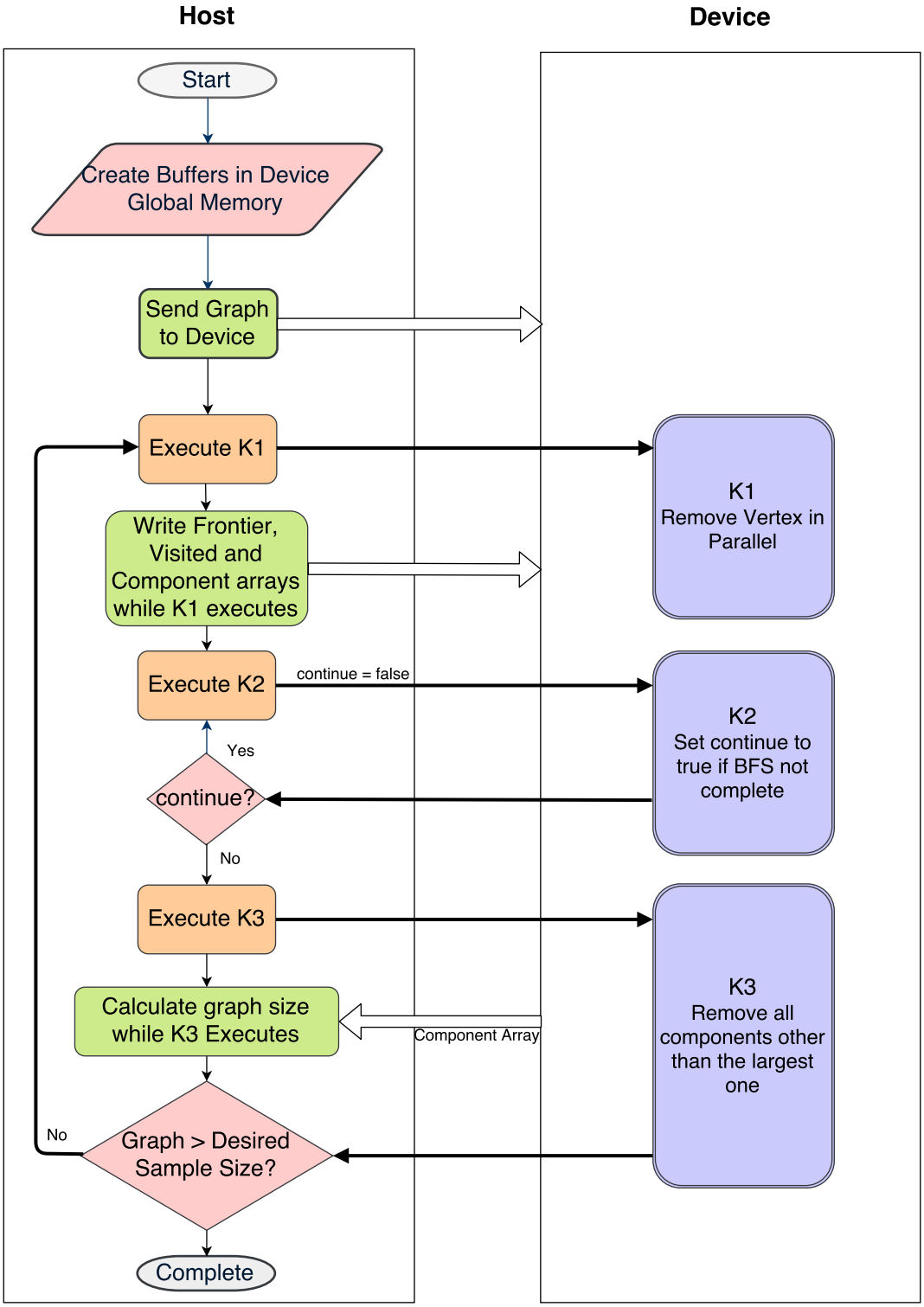

Figure 3.5: Parallel graph sampling.

CHAPTER 4

RESULTS

4.1. Data Generation and Experimental Setup

The techniques discussed in our work are targeted towards sampling of power-law graphs i.e. the graphs where number of vertices of degree x is proportional to $x^{-\alpha}$, where $\alpha > 0$ is a constant. Since vast majority of real world graphs follows power-law degree distribution; it makes our proposed technique more widely applicable. Different studies have shown that degree sequence of World Wide Web [24][25][26] and Internet router graph [27] follow power-law distributions among other applications [28] [29] [30]. For evaluation of our proposed strategy we generated synthetic power-law graphs using parameters to match the real world graphs that follow power-law i.e. graphs of up to 30M vertices with power-law degree distribution of exponent of 2.71 and average degree of 5.

We implemented the sampling algorithms on Pentium(R) and parallel version is implemented using OpenCL 1.2 on Intel Xeon W3565, Core-i5 2600, DE5-Net Stratix V GX FPGA Development Kit and GeForce GTX 480 GPU. Specifications of these devices are shown in Table 4.1.

Table 4.1: Specifications of FPGA, GPU and CPU used in our experiments.

	DE5-Net Stratix V	GeForce GTX 480	Xeon W3565	Core-i5
Base Frequency	-	700 MHz	3.20 GHz	3.3 GHz
# Logical Cores	1	480	8	4
Memory	4 GB	1536 MB	6 GB	6 GB

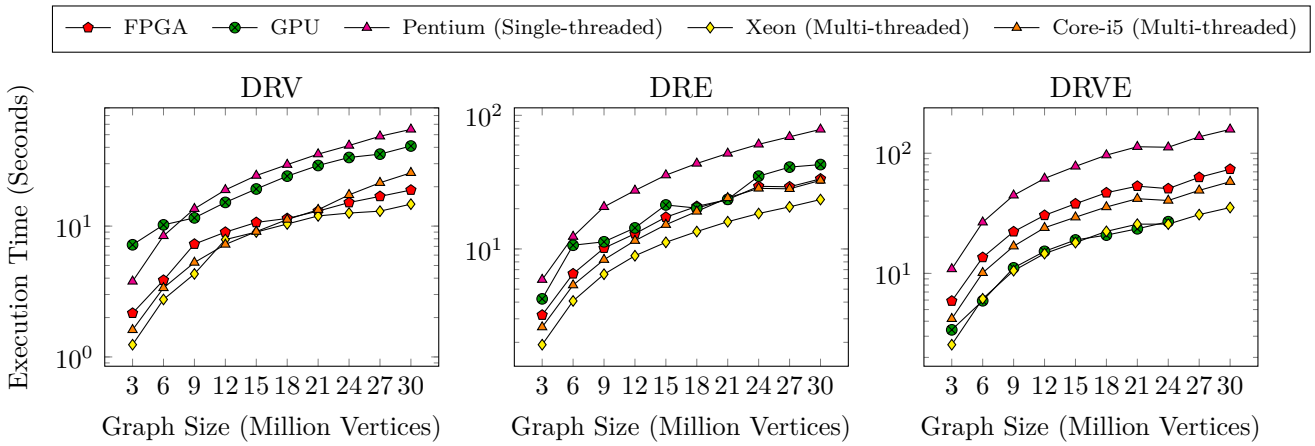

Figure 4.1: Execution time for different sampling techniques: DRV, DRE, DRVE with increasing graph size, executed on FPGA, GPU and CPU.

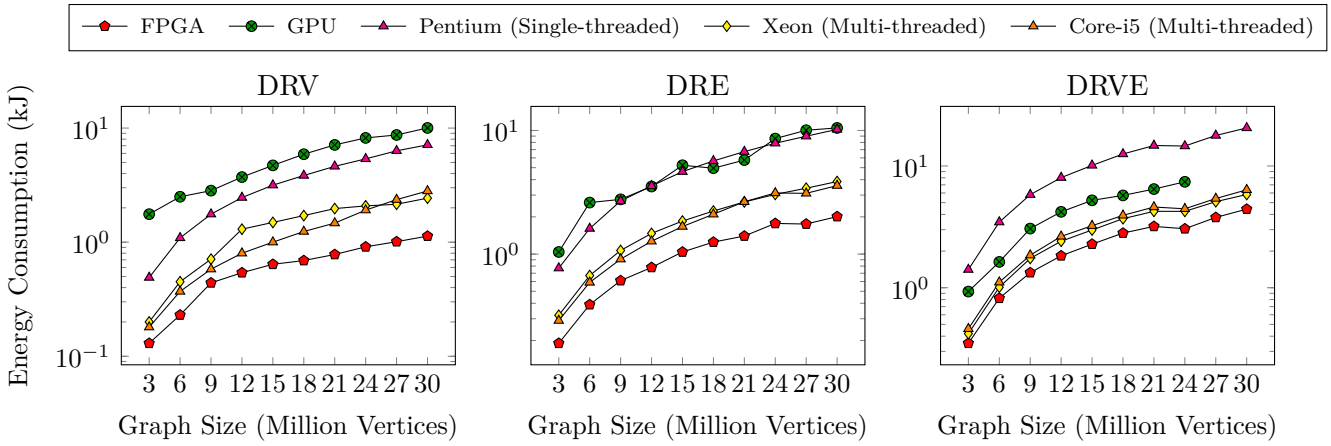

Figure 4.2: Energy consumption for different sampling techniques: DRV, DRE, DRVE with increasing graph size, executed on FPGA, GPU and CPU.

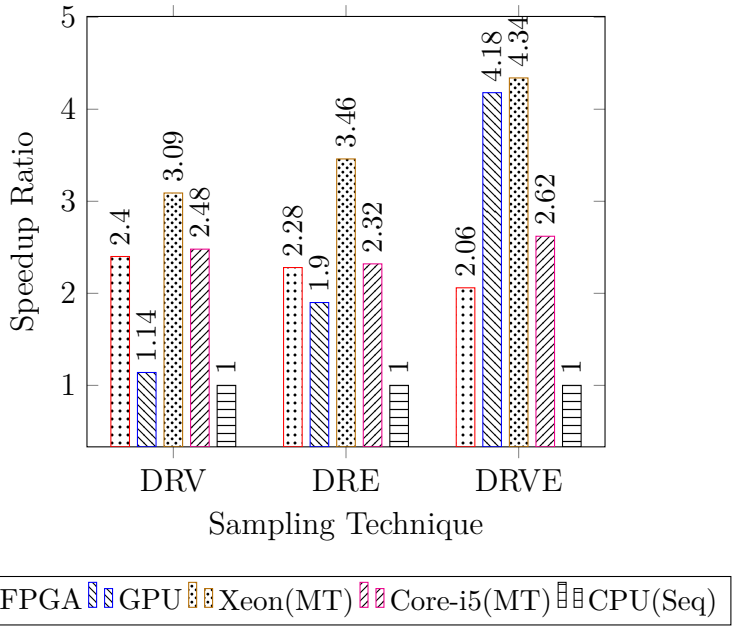

Figure 4.3: Speedup ratio for different sampling techniques when executed on FPGA, GPU and CPU (Multi-Threaded) w.r.t CPU sequential version.

4.2. Timing Analysis

We implemented sampling strategies on CPU, FPGA and GPU where sequential version of the algorithm runs on Pentium(R) while FPGA, GPU, Xeon and Core-i5 run the parallel version. The results in Fig. 4.1 show that our FPGA based technique is faster by more than 2x as compared to algorithms running on the CPU for all three sampling strategies. For FPGA vs GPU timing is comparable for DRV and DRE but GPU outperforms FPGA for DRVE. The relative speed up of FPGA and GPU with respect to CPU is shown in Fig. 4.3.

4.3. Energy Efficiency Analysis

To measure power efficiency we need to measure power consumed during execution of algorithms on each device. Power consumption for each device was measured experimentally

Table 4.2: Comparison between sequential and parallel sampling techniques in terms of graph characteristics.

		Average Degree		Degree Exponent		Rank Exponent	
	Sampling Ratio	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel
DRV	8.58	4.826	4.826	-2.015	-2.016	-0.531	-0.531
	17.02	4.654	4.654	-2.013	-2.013	-0.509	-0.509
	29.28	4.394	4.394	-2.008	-2.007	-0.475	-0.475
	37.20	4.213	4.213	-2.024	-2.022	-0.451	-0.451
	48.69	3.943	3.943	-2.012	-2.011	-0.415	-0.415
	59.58	3.679	3.679	-1.99	-1.99	-0.377	-0.377
	69.67	3.396	3.395	-1.971	-1.971	-0.335	-0.335
DRE	5.52	4.665	4.665	-2.023	-2.023	-0.511	-0.511
	17.66	4.044	4.044	-2.038	-2.038	-0.429	-0.429
	27.83	3.622	3.622	-2.054	-2.054	-0.369	-0.369
	39.01	3.24	3.24	-2.067	-2.067	-0.311	-0.311
	51.13	2.898	2.898	-2.093	-2.094	-0.256	-0.256
	59.65	2.695	2.695	-2.116	-2.116	-0.22	-0.22
	72.75	2.433	2.433	-2.142	-2.142	-0.17	-0.17
DRVE	9.27	5.059	5.059	-2.014	-2.014	-0.499	-0.499
	17.57	5.117	5.117	-2.015	-2.018	-0.451	-0.451
	31.73	5.235	5.234	-2.015	-2.017	-0.371	-0.317
	37.76	5.292	5.292	-2.014	-2.016	-0.338	-0.388
	48.12	5.409	5.409	-2.011	-2.015	-0.281	-0.281
	60.21	5.583	5.583	-2.008	-2.005	-0.215	-0.215

using a *watt meter*. For sequential version we use Pentium(R) to minimize power consumed by idle cores. To measure power, we disconnect both GPU and FPGA from the system and run the algorithm with minimum (constant) number of background applications running. Average power consumed by CPU for our three implementations is around 70 Watts. Next we connect DE5-Net to the system through PCIe and provide power from the system power-supply. The average power consumed by the system for the execution of FPGA code is around 60 Watts. For GPU power consumption, we first disconnect DE5-Net from the system and then connect GPU with power being drawn from system power-supply. The average power consumed by the system during execution of GPU code is around 245 Watts.

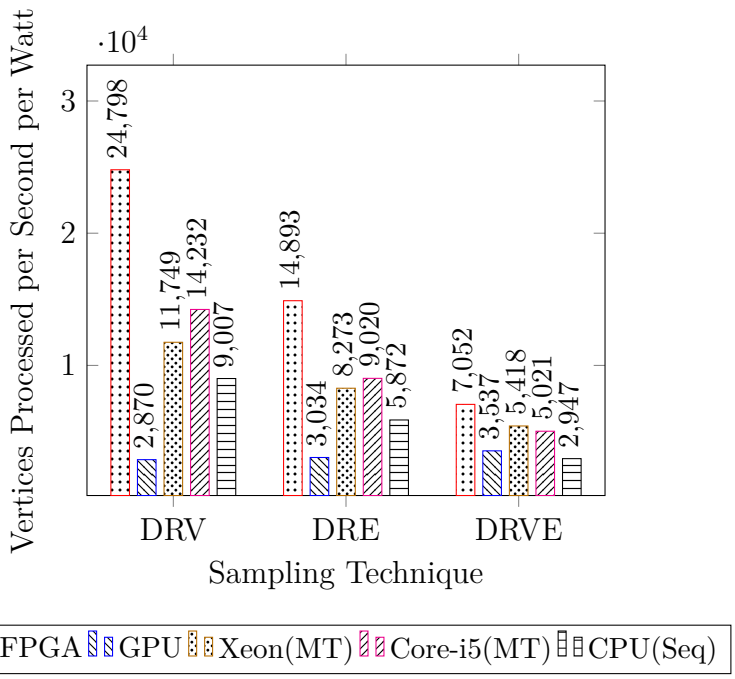

Figure 4.4: Energy efficiency in terms of "Vertices Processed per Second per Watt" for different sampling techniques when executed on FPGA, GPU and CPU.

Note that we only use Pentium CPU as host for FPGA and GPU versions as only a small fraction of host computational resources are required to schedule kernels. For Xeon and Core-i5 we can directly measure power as no extra devices are connected to them. Fig 4.4 shows the power efficiency comparison of different sampling strategies.

We measure energy efficiency as "Number of Vertices Processed per Second per Watt". When compared with GPU implementation of these algorithms, FPGAs have a comparable speedup but outperform GPUs more than 2x in energy efficiency.

4.4. Quality Assessment

Our proposed FPGA based strategy is only accurate if it produces results similar to the sequential algorithms described in [3]. We run sequential and parallel version of three

sampling algorithms on the largest graph (30M vertices) and measure characteristics like *Average Degree*, *Degree Exponent* and *Rank Exponent*. Rank Exponent and Degree Exponent are defined as follows [3]:

Rank Exponent is defined as the slope of log-log plot of node degrees vs their rank, where a node of kth highest degree will have a rank of k.

Degree Exponent is defined the slope of log-log plot of degree frequency vs degree.

Our experiments show that, for sampling percentages ranging from 10% to 70%, our proposed FGPA based strategy exhibits similar results with negligible differences as shown in Table 4.2.

4.5. Discussion

Even though FPGA has the best power efficiency, speedup for DRV and DRE is higher as compared to DRVE. Remember that we use exponential graph reduction technique (see section 3) to speedup DRV and DRE. However, our proposed technique does not give significantly superior results for DRVE. To explain this anomaly in our results, we consider the structure of power-law graphs that we use for our experiments. Power-law graphs have large number of vertices with a small degree while few vertices are of high degree. While removing a vertex-edge we pick a random vertex and then remove one of its edges randomly with probability of selecting a vertex of lower degree being significantly higher. This will disconnect the graph significantly even with relatively small number of vertex-edges being removed. Therefore, our exponential reduction techniques that is primarily included in the design for speeding up the sampling process is not as effective for DRVE.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1. Conclusion

The goal of this thesis is to develop a graph sampling technique using FPGAs which can sample large graphs in parallel, is highly scalable and energy-efficient. We develop parallel FPGA-based sampling strategy that include Deletion of Random Vertex (DRV), Deletion of Random Edge (DRE) and Deletion of Random Vertex-Edge (DRE). Developing algorithms that process graph using thread level parallelism is big challenge because of the irregular and interconnected nature of graphs. Our proposed FPGA based design not only exploits fine grained parallelism for sampling process but is also highly energy efficient when compared with CPU and GPU versions. Our extensive experiments indicate that our proposed technique compared with sequential (CPU) version provides 2x speedup and is 3x more energy efficient. When compared with GPU version, our FPGAs based strategy provides comparable execution run-times while having 10 times better energy efficiency. We also show that our FPGAs based strategy is more energy efficient when compared with parallel versions of CPUs. We also measure graph characteristics of sampled graph and compare them with the original graph and those produced by sequential sampling algorithms. These characteristics are within the bounds of originally proposed algorithms [3] and closely match with the sequential version.

5.2. Future Work

Having a memory efficient data structure and being able to sample from static graphs in parallel, our next goal is develop out-of-core algorithms for graphs that are too big to fit into the memory. We intend to implement these algorithms using OpenCL for FPGAs since, as presented earlier, they are the ideal choice in terms of speedup and energy efficiency. This project will be three folds:

1. Out-of-Core Algorithms: In this part of the project, our goal will be to extend the algorithm presented in this thesis to sample from indefinitely larger graphs that do not fit into FPGA memory. The graph will be stored in an external storage device e.g. HDD, SSD etc. and our algorithm will efficiently read portions of graph from the storage sample them and write the sampled graph back to the storage in a seamless fashion.

2. Large Number of Small Graphs: A variety of applications in proteomics and genomics depend on graph algorithms as the data being generated is represented in graphical form. The stream of data consists of countless graphs of relatively smaller size. Sampling multiple graphs at the same time presents its own challenges as we intend to take on this task in our next project.

3. Streaming Graphs: In various applications, especially in data centers, there is potential infinite size of data being streamed into the storage servers and it is of ultimate importance that statistical information be collected about that data to help analyze its properties. Sampling algorithms for streaming graphs can play a vital role in helping analyze this data at run time.

BIBLIOGRAPHY

[1] M. Kurant, M. Gjoka, Y. Wang, Z. W. Almquist, C. T. Butts, and A. Markopoulou, "Coarse-grained topology estimation via graph sampling", in *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, ACM, 2012, pp. 25–30.

[2] M. J. Salganik and D. D. Heckathorn, "Sampling and estimation in hidden populations using respondent-driven sampling", *Sociological methodology*, vol. 34, no. 1, pp. 193–240, 2004.

[3] V. Krishnamurthy, M. Faloutsos, M. Chrobak, J.-H. Cui, L. Lao, and A. G. Percus, "Sampling large internet topologies for simulation purposes", *Computer Networks*, vol. 51, no. 15, pp. 4284–4302, 2007.

[4] X. A. Dimitropoulos and G. F. Riley, "Creating realistic bgp models", in *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, IEEE, 2003, pp. 64–70.

[5] N. Ahmed, J. Neville, and R. R. Kompella, "Network sampling via edge-based node selection with graph induction", 2011.

[6] L. A. Goodman, "Snowball sampling", *The annals of mathematical statistics*, pp. 148–170, 1961.

[7] D. D. Heckathorn, "Respondent-driven sampling: A new approach to the study of hidden populations", *Social problems*, vol. 44, no. 2, pp. 174–199, 1997.

[8] J. Gantz and D. Reinsel, "Extracting value from chaos", *IDC iview*, vol. 1142, no. 2011, pp. 1–12, 2011.

[9] Facebook, *Facebook q4 2017 results*, `https://s21.q4cdn.com/399680738/files/doc_financials/2017/Q4/Q4-2017-Earnings-Presentation.pdf`, [Online; accessed 9-March-2018], 2017.

[10] Twitter, *Twitter q4 2017*, `http://files.shareholder.com/downloads/AMDA-2F526X/5984838773x0x970886/28A12845-4E0D-4F74-B82D-077241BCD7FE/Q4_2017_Selected_Company_Metrics_and_Financials.pdf`, Online: accessed 9-March-2018, 2018.

[11] C. Doerr and N. Blenn, "Metric convergence in social network sampling", in *Proceedings of the 5th ACM Workshop on HotPlanet*, ACM, 2013, pp. 45–50.

[12] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines", *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[13] J. Leskovec, J. Kleinberg, and C. 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*, ACM, 2005, pp. 177–187.

[14] K. Dempsey, K. Duraisamy, H. Ali, and S. Bhowmick, "A parallel graph sampling algorithm for analyzing gene correlation networks", *Procedia Computer Science*, vol. 4, pp. 136–145, 2011.

[15] K. Dempsey, K. Duraisamy, S. Bhowmick, and H. 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*, IEEE, 2012, pp. 725–734.

[16] A. H. Rasti, M. Torkjazi, R. Rejaie, N. Duffield, W. Willinger, and D. Stutzbach, "Respondent-driven sampling for characterizing unstructured overlays", in *INFOCOM 2009, IEEE*, IEEE, 2009, pp. 2701–2705.

[17] J. Fowers, G. Brown, J. Wernsing, and G. Stitt, "A performance and energy comparison of convolution on gpus, fpgas, and multicore processors", *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 25, 2013.

[18] D. Singh, "Implementing fpga design with the opencl standard", *Altera whitepaper*, 2011.

[19] U. Tariq, U. I. Cheema, and F. Saeed, "Power-efficient and highly scalable parallel graph sampling using fpgas", in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, IEEE, 2017, pp. 1–6.

[20] A. Benczur and D. R. Karger, "Randomized approximation schemes for cuts and flows in capacitated graphs", *arXiv preprint cs/0207078*, 2002.

[21] P. Hu and W. C. Lau, "A survey and taxonomy of graph sampling", *arXiv preprint arXiv:1308.5865*, 2013.

[22] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou, "Walking in facebook: A case study of unbiased sampling of osns", in *Infocom, 2010 Proceedings IEEE*, IEEE, 2010, pp. 1–9.

[23] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda", in *International Conference on High-Performance Computing*, Springer, 2007, pp. 197–208.

[24] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "Trawling the web for emerging cyber-communities", *Computer networks*, vol. 31, no. 11, pp. 1481–1493, 1999.

[25] ——, "Extracting large-scale knowledge bases from the web", in *VLDB*, vol. 99, 1999, pp. 639–650.

[26] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins, "The web as a graph: Measurements, models, and methods", in *International Computing and Combinatorics Conference*, Springer, 1999, pp. 1–17.

[27] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology", in *ACM SIGCOMM computer communication review*, ACM, vol. 29, 1999, pp. 251–262.

[28] X. Deng, Z. Feng, and C. H. Papadimitriou, "Power-law distributions in a two-sided market and net neutrality", in *International Conference on Web and Internet Economics*, Springer, 2016, pp. 59–72.

[29] X. 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, Xi'an China*, Springer, 2017, pp. 499–508.

[30] M Olmedilla, M. R. Martínez-Torres, and S. Toral, "Examining the power-law distribution among ewom communities: A characterisation approach of the long tail", *Technology Analysis & Strategic Management*, vol. 28, no. 5, pp. 601–613, 2016.