Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200097

Parallel Totally Induced Edge Sampling on FPGAs[1]

Akshit GOEL [a], Sanmukh R. KUPPANNAGARI [b,2], Yang YANG [b], Ajitesh SRIVASTAVA [b] and Viktor K. PRASANNA [b]
[a] *Department of Electrical and Electronics Engineering, Birla Institute of Technology and Science Pilani, India 333031*
[b] *Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California, USA 90089*

Abstract. Graphs are a powerful tool for data representation in a wide range of domains like social, biological, informational, etc. But their extremely large sizes often makes it computationally infeasible to study the entire graphs. Graph sampling provides a solution by generating smaller subgraphs which are computationally feasible to analyze and can be used to infer the properties of the entire graph. In this work, we develop a high throughput parallel implementation of Totally Induced Edge Sampling (TIES) algorithm on FPGA. Prior research has shown that TIES performs better than other sampling techniques in terms of preserving the topological properties of the original graph, and thus generates better quality subgraphs. The algorithm randomly samples the edges and inserts the corresponding vertices into the sampled vertex set until the desired number of vertices are sampled. Then, the edges connecting the sampled vertices are included in the sampled subgraph. We use multiple parallel pipelines to achieve high throughput and faster graph sampling. The parallel pipelines need to access a global dynamic data structure which contains the vertices sampled thus far. To support this, we develop a novel dynamic hash table data structure which supports parallel accesses in each clock cycle. We vary the number of pipelines, the size of the sampled subgraph and analyze the performance of the design in terms of on-chip FPGA resource utilization, throughput and total execution time. Our design achieves a throughput as high as 2471 Million Edges Per Second (MEPS) and performs 3.6x better than the state-of-the-art multi-core design.

Keywords. Parallel Graph Sampling, Totally Induced Edge Sampling, FPGA, Dynamic Data Structure

1. Introduction

Graphs are being used to represent data in a wide range of applications including World Wide Web, social media, genomics, and machine learning [17,9]. Graph analysis is a key computational technology in such applications to understand, codify and derive hidden information. However, the large size of real world graphs prevents efficient processing

[1]This work has been supported by Xilinx and by the U.S. National Science Foundation (NSF) under grant OAC-1911229.
[2]Corresponding Author: Sanmukh R. Kuppannagari; E-mail:kuppanna@usc.edu

even with the help of distributed graph databases and highly optimized processing platforms [3,4,5]. Analyzing multiple smaller representative subgraphs is a possible solution as the results of their analyses can be used to infer the properties of the original graph [1,12,16]. In many data repositories, graphs are stored in form of subgraphs to make their analysis computationally feasible [7,6,8]. Graph sampling algorithms are widely used to generate such representative subgraphs from large graphs.

A popular graph sampling algorithm is Totally Induced Edge Sampling (TIES). It randomly samples edges and thus mitigates the downward degree bias of vertex sampling. Total induction over the sampled vertices selects all the edges between them, which further improves the connectivity in the sampled subgraph. According to [2], Induced Edge Sampling performs better than many sophisticated state-of-the-art node sampling and topology based algorithms, such as forest fire or snowball sampling in terms of preserving graph structure and thus has been our choice for hardware implementation in this work. Sequential sampling of very large graphs can be unacceptably slow, and thus we target high throughput parallel implementation of TIES on FPGA. Most significant challenge in a parallel implementation of TIES is to perform fast parallel access and updates to a data structure storing the sampled vertices.

In this work, we focus on sampling a small subgraph (thousands or tens of thousands of vertices) from a large graph. The sampled subgraph can then be used for downstream applications such as GCN [16]. Specifically, given a large graph $G = (V, E)$ in external memory, we need to generate a subgraph $Gs = (Vs, Es)$ and store it on the FPGA on-chip memory. Our design consists of multiple parallel pipelines which randomly select edges from E until a desired number $|Vs| \approx Ns$ distinct vertices in V have been sampled. This is followed by total induction phase that iterates over all the edges corresponding to the sampled vertices in E for induction in Vs. The major contributions of this work are as follows:

1. We develop a high throughput parallel implementation of the Totally Induced Edge Sampling (TIES) algorithm on FPGA. To the best of our knowledge, this is the first parallel FPGA implementation of the algorithm.
2. We design a novel parallel on-chip dynamic hash table data structure that enables multiple pipelines to read and insert data concurrently without stalling.
3. Our design achieves a throughput as high as 2471 Million Edges Per Second (MEPS) using 8 pipelines.

The rest of the paper is organized as follows: Section 2 presents the related work; Section 3 gives a brief overview of the algorithm; Section 4 discusses the architecture design for algorithm implementation and Section 5 reports the experimental results.

2. Related Work

In [2], the authors propose Induced Edge Sampling (or Totally Induced Edge Sampling), and discuss its merits as compared to other sampling techniques. According to [2], TIES offsets the downward degree bias of node sampling. It improves connectivity of the subgraph, and thus better preserves the topological properties of the input graph than many other sampling algorithms. In [7], the authors present a parallel implementation of the TIES algorithm on multi-core platform which samples a graph of size 1442M edges in $<$

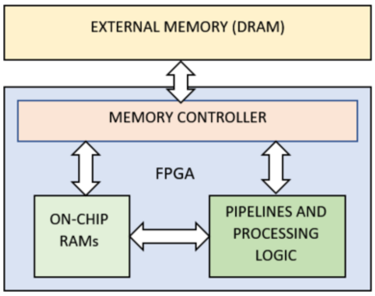

Figure 1. Overall Architecture Model

2.5 seconds. The implementation involves atomic operations to prevent concurrent updates by distinct threads. This synchronization steps introduces inefficiencies. Authors in [11] develop an FPGA implementation of a reduction based sampling algorithm which generates a subgraph by removing nodes and edges. This is highly inefficient for sampling small subgraphs from very large graphs. To the best of our knowledge, the work done in [11] is the only prior FPGA graph sampling implementation.

3. Overview of the TIES Algorithm

A detailed description of the TIES algorithm can be found in [2]. A brief description is as follows: The algorithm takes two inputs: 1) The graph to be sampled i.e. $G = (V, E)$ and 2) The number of vertices to be sampled Ns. The algorithm outputs a subgraph $Gs = (Vs, Es)$ from a graph $G = (V, E)$ such that $|Vs| \approx Ns$.

There are two major steps or phases of the algorithm which are described as following [2]:

1. Phase-1
   (a) Randomly select an edge $e_{i,j} = (i, j)$ from the set E.
   (b) Sample the vertices i and j into set Vs, if not already sampled.
   (c) Repeat (a) and (b) until $|Vs| = Ns$. This completes the set Vs.
2. Phase-2
   For each edge $e_{i,j}$ in set E, include it in set Es if i and $j \in Vs$. This completes the set Es

The sets Vs and Es represent the sampled subgraph $Gs = (Vs, Es)$.

4. Parallel Graph Sampling Architecture on FPGA

4.1. Architecture Model

Fig. 1 shows the high level view of the proposed architecture. The external memory stores the graph to be sampled. The FPGA design consists of multiple parallel pipelines which access the external memory containing the inputs graph through the memory controller.

The graph is sampled by the pipelines and is stored on the on-chip memory. Each pipeline is identical and operates independent of other pipelines.

We assume that each access to the external memory incurs a latency of l clock cycles. The value of l is typically 10-50 clock cycles. Memory latency does not impact our design since all the pipelines operate without stalling i.e. after an initial latency, the data is continuously streamed from the external memory without any delay. The external memory has a peak bandwidth bw. To achieve high throughput we increase the number of pipelines and saturate bw.

Phase-1 implementation consists of p pipelines which sample $\frac{p}{2}$ edges i.e. p vertices in parallel. Phase-2 implementation consists of p pipelines and operate on p vertices simultaneously. The design also consists of a dynamic hash table data structure that supports p inserts/searches in parallel.

Algorithm 1 Phase-1

```
  rand() $\rightarrow$ random number generator; hash() $\rightarrow$ hash function
  for $pipe = 0, 1, ..., p-1$ in parallel do
    while $|localVs[pipe]| \leq Ns/p$ do
      if pipe % 2 = 0 then
        $addr \leftarrow rand()$
        $(i, j) \leftarrow edge\_list(addr)$
        $v[pipe] \leftarrow i$
        $v[pipe+1] \leftarrow j$
      end if
      $v[pipe].hash\_value = hash(v[pipe])$
      if $v[pipe] \neq hash\_table[pipe](v[pipe].hash\_value)$ then
        $hash\_table[pipe](v[pipe].hash\_value) = v[pipe]$
        $localVs \leftarrow localVs \cup v[pipe]$
      end if
    end while
  end for
```

Algorithm 2 Phase-2

```
  $mv \rightarrow 0$; $n \rightarrow 0$
  for $pipe = 0, 1, ..., 2p-1$ in parallel do
    while $mv \leq Ns$ do
      $i \leftarrow localVs[pipe].(mv)$
      while $n \leq |i.edges|$ do
        $j \leftarrow i.edges(n)$
        $j.hash\_value = hash(j)$
        if $j = hash\_table[pipe](j.hash\_value)$ then
          $localEs[pipe] \leftarrow localEs[pipe] \cup (i, j)$
        end if
      end while
    end while
  end for
```

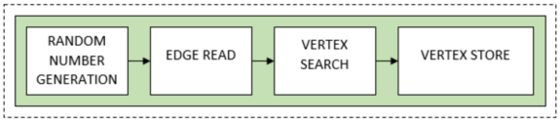

Figure 2. Pipeline Stages: Phase-1

4.2. *Pipeline Design: Phase-1*

The objective of Phase-1 is to sample N_s unique vertices. Phase-1 consists of p parallel pipelines. The various pipeline stages are shown in Fig. 2. These stages are – Random Number Generation, Edge Read, Vertex Search and Vertex Store. Each pipeline continues to operate in Phase-1 until it samples Ns/p unique vertices. Some pipelines may take more iterations than others because edges sampled by them might have common vertices. The various stages of each pipeline are described below:

1. *Random Number Generation*: Since each edge is represented by two vertices, the number of edges required to be sampled to generate p vertices simultaneously are $p/2$. Therefore, in every clock cycle, only $p/2$ pipelines generate random numbers which serve as indices to read edges from the external memory. Thus, a total of $p/2$ edge indices are given as input to the external memory in every clock cycle.
2. *Edge Read*: The architecture assumes that the graph is stored in the external memory in the form of a simple data structure where each edge is represented by its two end vertices. We assume the latency associated with fetching information from the external memory to be l clock cycles. The memory controller is given an input of $p/2$ memory addresses in each clock cycle. Thus after an initial delay of l, the memory outputs $p/2$ edges every clock cycle with each edge represented by its two vertices.
3. *Vertex Search*: Each pipeline is associated with an individual hash table and Local Vertex register. The details of the hash table design are discussed in Section 4.4. The vertices fetched by the pipelines in 'Edge Read' stage are searched in the corresponding hash table to check if they have already been sampled or not. Since each pipeline is associated with a separate hash table, therefore all p vertices fetched in previous stage are searched in O(1) time. The search returns whether the vertex is stored in the table or not.
4. *Vertex Store*: In this stage, each pipeline stores the fetched vertex if the vertex search stage returns negative value. The vertex is inserted in both the hash table and Local Vertex register of the corresponding pipeline. Thus, in a given clock cycle, a maximum of p vertices are inserted collectively by all the pipelines. Concatenation of all Local Vertex registers at the end of Phase-1 gives the sampled vertex set Vs.

4.3. *Pipeline Design: Phase-2*

Phase-2 consists of p parallel, identical five stage pipelines as shown in Fig. 3. These stages are – Vertex Read, Edge-Index Generation, Edge Read, Vertex Search, Edge Store. During Phase-2, all the edges either incident to or from the sampled vertices are checked for induction in Es. If the number of edges corresponding to their sampled vertices are less, some pipelines might finish Phase-2 before others. Each pipeline has a Lo-

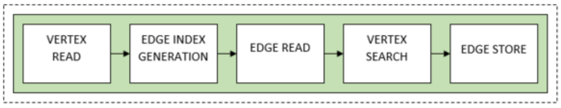

Figure 3. Pipeline Stages: Phase-2

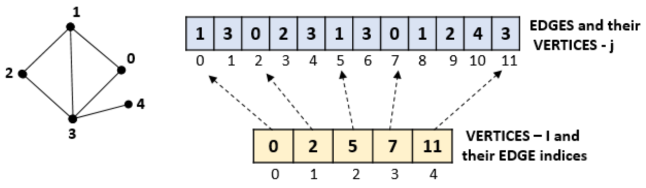

Figure 4. Input Graph Representation: Phase-2

cal Edge register made up of the on-chip memory to store the edges induced by it. The various stages of a pipeline are described below:

1. *Vertex Read*: The architecture assumes that the graph is stored in external memory in form of a CSR matrix as shown in the Fig. 4. To read an edge, we require two indices-one for reading the vertex and the other for reading the edge. Each pipeline reads a sampled vertex from its Local Vertex register and iterates over its edges. For each edge, the corresponding edge index is generated in the *Edge Index Generation* stage. Once all the edges of a vertex are processed, the next sampled vertex in the Local Vertex register is read.
2. *Edge Index Generation*: All the edges associated with a vertex are stored consecutively and a pointer points to the first edge of this set as shown in Fig. 4. Each pipeline inputs the vertex index (generated in previous stage) to the external memory, and fetches the edge index for the first edge of the vertex's edge set. The edge index fetched is incremented by $\log_2 m$ bits (assuming $|E| = m$) in every subsequent clock cycle to read the next edge until the edge index given by the next vertex of CSR matrix is reached. Each pipeline generates an edge index for the external memory and therefore a total of p edge indices are generated per clock cycle.
3. *Edge Read*: Each pipeline inputs the edge index generated in the previous stage to the external memory, and fetches the edge corresponding to it. Each fetched edge is represented by one of its vertex (other one being the vertex selected in Phase-1 itself). We assume the latency associated with fetching information from the external memory to be l clock cycles. Since for each edge we need two external memory accesses, thus after an initial delay of $2*l$ clock cycles, at the end of each clock cycle the memory gives a total of p edges for all pipelines.
4. *Vertex Search*: Each pipeline searches the vertex fetched in the previous stage in its corresponding hash table to check if it belongs to Vs or not. Since each pipeline has its separate hash table (Section 4.4), therefore all pipelines operate in parallel. The edges whose vertices belong to Vs are included in the subgraph in the next stage.
5. *Edge Store*: Each pipeline consists of a Local Edge register made up of on-chip memory. If in the previous stage, it is indicated that the edge vertex belongs to Vs,

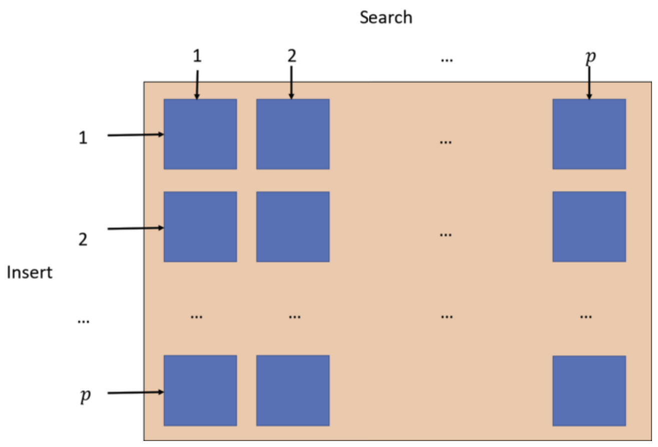

Figure 5. Overall hash table Architecture

then that edge is induced and stored in the corresponding pipeline's Local Edge register in form of its two end vertices. Concatenation of all Local Edge registers at the end of Phase-2 gives the sampled edge set Es.

Note that we assume different formats for both the phases. However, these formats can be generated just once in an offline manner and used to generate multiple subgraphs. Therefore, we do not consider format conversion time in our design and its evaluation.

4.4. Hash Table Architecture

In Phase-1, a vertex is inserted into the sampled vertex set only if it has not been already inserted before. Similarly, in Phase-2, an edge is inserted into the sampled edge set if both its vertices are sampled. As our design consists of p pipelines, we require a data structure that performs p parallel lookups of vertices without stalling. Thus, we propose a hash table to store the sampled vertices for quick parallel lookup. The hash table is implemented using on-chip FPGA RAMs (BRAM or URAM [14]). As a BRAM/URAM block supports a single read/write operation per clock cycle, implementing a hash table which supports p queries per cycle is challenging.

To address this challenge, we implement an on-chip hash table that can process p parallel queries in each clock cycle (in a pipelined manner) as shown in Figure 5. The hash table consists of p^2 hash blocks each of size $\frac{N_s}{p}$ vertices. Each hash block is implemented as two level hash tables. For insertion queries, pipeline p_l , $0 \leq l \leq p$ inserts the same value in each hash block along the row l. For search queries, each pipeline p_l reads all the hash blocks along the column l. Thus, the value inserted by a pipeline p_l will be available to a pipeline $p_{l'}$ via hash block ll' i.e. the block at the intersection of lth row and l'th column. Assuming it takes k cycles to insert into the pipeline, our design ensures that a vertex inserted by a pipeline is stored in at least one hash block of all the pipelines with a delay of k cycles.

5. Experiments and Results

5.1. Experimental Setup

The experiments were conducted using the Xilinx Alveo U200 Accelerator Card [13]. The target FPGA device has 1,182,240 LUTs, 591,840 LUTRAMs, 2,364,480 Flip-flops, and 35MB of on-chip SRAMs. We assume the external memory in the form of one DDR4 SDRAM chip, which has a peak data transfer rate of 19.2 GB/s. We synthesize, place-and-route, and simulate our designs using Xilinx Vivado Design Suite 2018.2 [15]. We evaluate the performance of our design using the the soclj dataset [10] which is a LiveJournal friendship social network. The dataset has 4.85 Million vertices and 68.46 Million edges. The vertices have an average degree of 14.1.

5.2. Evaluation Methodology and Performance Parameters

We analyze the performance of our design by varying the number of pipelines from 2 to 8, and varying the number of vertices to be sampled from 1,000 to 20,000. We target our design at 300 MHz FPGA frequency. Each design case is evaluated on the following performance parameters:

1. *Execution time*: The sum of execution times of Phase-1 and Phase-2, i.e. the total time taken to generate the subgraph from the input graph.
2. *Throughput*: Throughput of the design in terms of million edges sampled per second (MEPS). MEPS is calculated by dividing the number of edges in the sampled subgraph with total execution time.
3. *Resource Consumption*:Utilization of FPGA resources in terms of percent usage of LUTs, flip-flops, and on-chip RAMs (BRAM and URAM).

5.2.1. Results

Resource utilization of the design is reported in Figure 6. Figure 6 (a) shows the resource utilization by fixing the number of sampled vertices at 10,000, and changing the number of pipelines from 2 to 8. The Vivado tool chooses to use URAM as the default resource for large on-chip storage, however, in the event this leads to failure in meeting the target clock frequency, it falls back to using BRAM. We observe this behaviour for the cases of number of pipelines equal to 2 and 8. For the case of 2 pipelines, the large size of each hash block leads to this behaviour, while for the case of 8 pipelines the interconnection complexity of connecting each pipeline with all the hash blocks in its row/column leads to this behaviour. Therefore, the BRAM utilization is relatively higher in these two design points. Figure 6 (b) shows the resource utilization by fixing the number of pipelines to 8 and varying the size of sampled subgraph to 1000, 5000, 10000, and 50000 vertices. The URAM utilization increases almost linearly with the subgraph size, because the edge registers are mapped to URAM. We report the execution time and throughput performance in Table 1. The results clearly verify the scalability of our design with the number of pipelines, with each case supporting a maximum clock frequency of 300 MHz.

Table 1. Execution Time and Throughput

# Pipelines	Execution Time (μs)	Throughput (MEPS)
2	457.24	614.05
4	227.16	1236.01
6	151.49	1853.40
8	113.62	2471.14

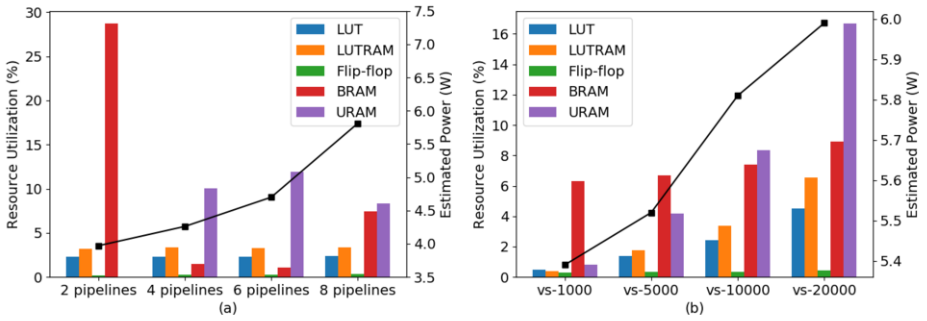

Figure 6. Resource Utilization and Estimated Power

5.3. *Comparison with State-of-the-art Multi-core Design*

We compare the performance of our 8 pipelines design with the highly optimized multi-core design of the same algorithm. In [7], the authors implement their design on a 16-core machine with 2 x 2.6GHz 8-core Intel Xeon E5-2650 processors (256KB L2 and 20MB L3 cache) and 128GB main memory with 2-way hyperthreading. We compare the performance in terms of throughput i.e. Million edges sampled per second (MEPS). As shown in Table 2, our design achieves a 3.63x improvement over the multi-core implementation. The majority of the speedup comes from the efficient and effective architecture of our design, as cost of the global synchronization, which is required on a multi-core design, is significantly reduced by the dynamic hash table. It is important to note that the design in [7] is tailored for sampling large size subgraphs (hundreds of thousands of vertices) unlike our design which is tailored for small size subgraphs.

Table 2. Comparison with multi-core design

Parameter	Multi-core Design	This Design (pipelines = 8)	Improvement
Throughput (MEPS)	680.87	2471.14	3.63

6. Conclusion

In this work, we presented an FPGA accelerated Totally Induced Edge Sampling (TIES) algorithm. We proposed a novel parallel hash table data structure that supports multiple concurrent read and write requests. Our design achieves a high throughput of 2471 MEPS which is 3.6x times better than the state-of-the-art design. Our design consists of multiple pipelines operating in parallel without stalling. The design is scalable with number of pipelines and number of sampled vertices with a sustained clock frequency of 300 MHz.

References

[1] Nesreen K Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1446–1455. ACM, 2014.

[2] Nesreen K Ahmed, Jennifer Neville, and Ramana Kompella. Network sampling: From static to streaming graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(2):7, 2014.

[3] Vito Giovanni Castellana, Alessandro Morari, Jesse Weaver, Antonino Tumeo, David Haglin, Oreste Villa, and John Feo. In-memory graph databases for web-scale data. *Computer*, 48(3):24–35, 2015.

[4] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[5] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[6] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[7] Kartik Lakhotia, Rajgopal Kannan, Aditya Gaur, Ajitesh Srivastava, and Viktor Prasanna. Parallel edge-based sampling for static and dynamic graphs. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 125–134. ACM, 2019.

[8] Jure Leskovec and Andrej Krevl. {SNAP Datasets}:{Stanford} large network dataset collection. 2015.

[9] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[10] Stanford. Stanford large network datasets. https://snap.stanford.edu/data/socnets.

[11] Usman Tariq, Umer I Cheema, and Fahad Saeed. Power-efficient and highly scalable parallel graph sampling using fpgas. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2017.

[12] Bin Wu, Ke Yi, and Zhenguo Li. Counting triangles in large graphs by random sampling. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2013–2026, 2016.

[13] Xilinx. Alveo u200 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u200.html.

[14] Xilinx. Ultraram. https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf.

[15] Xilinx. Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.

[16] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Accurate, efficient and scalable graph embedding. *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2019)*, 2019.

[17] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K Prasanna. An fpga framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 69–77. ACM, 2018.