2021 31st International Conference on Field-Programmable Logic and Applications (FPL) | 978-1-6654-3759-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/FPL53798.2021.00042

Graph Sampling with Fast Random Walker on HBM-enabled FPGA Accelerators

Chunyou Su*, Hao Liang†, Wei Zhang‡, Kun Zhao†, Baole Ai†, Wenting Shen†, Zeke Wang§
*‡Department of Electronic and Computer Engineering,
Hong Kong University of Science and Technology, Hong Kong SAR, China
†Alibaba Group, China
§Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China
*csuae@connect.ust.hk, ‡wei.zhang@ust.hk, §wangzeke@zju.edu.cn

***Abstract*—Graph neural networks (GNNs) have gained increasing popularity among researchers recently and have been employed in many applications. Training GNNs introduce a crucial stage called graph sampling. One of the most important sampling algorithms is Random Walk. However, Random Walk and many of its variants share and suffer from the same performance problem caused by random and fragmented memory access patterns, leading to significant system performance degradation.**

In this work, we present an efficient graph sampling engine on modern FPGAs integrated with in-package high bandwidth memory (HBM), which brings data closer and faster to the core logic. The hardware walker design is modular and easily scalable for massive parallelism, to fully utilize the available HBM channels. Our design also provides the flexibility to support Random Walk and two of its variants on both homogeneous and heterogeneous graphs. On real-world graph datasets, we achieve a 1.39×-3.74× speedup with a 2.42×-6.69× higher energy efficiency over highly optimized parallel baselines on a Xeon CPU. We also implement these algorithms on a NVIDIA Tesla V100 GPU and achieve comparable dynamic energy consumption.

***Index Terms*—graph sampling, random walk, HBM, FPGA**

I. Introduction

Recently, graph neural networks (GNNs) have been actively studied in both industry and academia [1–7, 9, 18, 22, 23]. Unlike image data in the form of tensors or text/speech data in the form of sequences, graphs are highly flexible and can be used to describe many real-world datasets such as social networks, academic graphs and knowledge graphs.

The powerful graph models have motivated researchers to propose various GNN models targeting numerous downstream applications [1–6, 9]. However, the irregular and randomized connections in graphs impinge the concurrent aggregation and combination of embedding vectors due to the load imbalance problem, which is truly the Achilles' heel in this field.

To address the issue, most of the cutting-edge GNN models choose to extract vectorized and regular data from raw graphs before later stages of learning [1–6]. That is, they look at the raw graph in a local point of view at a time, and aggregate information extracted from every local subgraph. In this way, the graph data for GNN learning is regularized, which reduces the difficulty for parallel execution.

The pre-processing phase of generating iterable subgraphs is also known as graph sampling. In some GNN models, these subgraphs are used for the aggregation and combination of feature vectors as needed [1–3], while other models deal with the sampling phase separately and feed the generated subgraphs into the subsequent training models [4–6].

Despite the success of sampling-based GNNs, implementing graph sampling is far from perfect regarding both throughput and energy efficiency. The main reasons are threefold. Firstly, and most importantly, traditional DDR memory is weak at concurrent memory accesses due to the limited number of independent memory channels, which leads to a throughput mismatch between the memory system and parallel processing units. Secondly, the irregular graph structure entails a unpredictable memory access pattern, which restricts the available memory bandwidth as the DDR memory is best at massive consecutive accesses. Finally, when a large amount of data is shared off-chip, the classic cache-based memory hierarchy cannot assist due to poor locality and invalid data exchanged frequently in and out of the cache.

Such observations reveal the limitations of the conventional cache-based Von-Neumann architecture for graph sampling tasks. Instead, domain-specific architectures are believed to be promising since flexible memory accesses are enabled through customized design. Specifically, FPGA platforms are favored by developers due to their post-fabrication reconfigurability and short time-to-market. Moreover, with the emergence of HBM-enabled FPGA platforms, breaking the memory wall through concurrent memory accesses becomes possible.

In this work, we propose a novel graph sampling accelerator based on HBM-enabled FPGAs. We focus on walker-based graph sampling methods (described later) since their performance suffers the most from the fragmented memory access pattern. Our main contributions are as follows.

- We propose a graph sampling accelerator targeting three walker-based sampling algorithms with hardware optimizations and implement it on an HBM-enabled FPGA platform.
- We evaluate the proposed accelerator on real-world graphs and achieve a **1.39×-3.74×** speedup with a **2.42×-6.69×** higher energy efficiency over parallel baselines implemented on a Xeon CPU. We also present an evaluation on the V100 GPU with HBM for comparison.
- We test the design on randomly generated graphs of varying graph size and alter the restart probability (explained later) to show the performance robustness.

II. Background

A. Walker-based Graph Sampling

Graph sampling algorithms can be generally categorized into *walker-based sampling* and *neighborhood-based sampling*. As the name implies, the basic of *neighborhood-based sampling* is to sample a subset of nearby nodes reachable in a few hops from the target node, which is widely adopted in graph convolutional network (GCN)-based models [1–3]. *Walker-based sampling* normally traverses the graph randomly from a starting node and terminates when a pre-defined path length (number of hops) is reached. In general, *walker-based sampling* is more commonly adopted in GNN models derived from the natural language processing (NLP) field, as can be found in [4–6].

Generally, a specific *walker-based sampling* algorithm takes a graph ***G=(V,E)*** as the input, where ***V*** is the vertex set and ***E*** is the edge set. It also takes a subset of vertices from ***V***, each one of which would serve as the starting node of an independent random walk path. During the execution, the algorithm is typically given another two user-defined parameters, namely the number of walks w from each starting node and the length (#hops) l for each walk path.

The randomness of *walker-based sampling* is defined in each specific algorithm, which can be formalized as the transition probability describing the likelihood of transition from the current residing node to the next-hop node. Generally, the transition probability could be either independently distributed or conditionally distributed. In the former case, the probability at each hop is not affected by any previous state. In the latter case, the probabilistic decision is jointly decided by both the current and previous states. Here, we represent the transition probability in the form of first-order conditional probability $P_e(n_i|n_{i-1})$, where e is the outgoing edge from the i_{th} visited node n_i (i.e., e=(n_i, n_{i+1})) and $P_e(\cdot)$ is the transition probability defined by the sampling algorithm. Note that here the condition variable n_{i-1} should be treated as a comprehensive representation that carries composite information of the previous node, possibly including the node index, node type, the covered hop count, and so on.

B. Random Walk Sampling and its Variants

This section covers the classic Random Walk and two of its variants that we implement in this work. Interested readers can refer to [7] for more variants and their descriptions.

Random Walk (RW): The **RW** method refers to the classic random walk sampling used in DeepWalk [4]. We formalize it as the following transition probability:

$$P_e(n_i) = \frac{1}{|N(n_i)|}. \tag{1}$$

Here $N(n_i)$ is the set of vertices neighboring the current residing vertex n_i, and $|\cdot|$ returns its cardinality. The transition probability simply navigates a walker to a uniformly randomly selected neighbor in the next hop.

Random Walk with Restart (RWR): The **RWR** method has been widely used in *Personalized Page Rank* (PPR) to generate personalized importance vectors [8]. It is also adopted as the sampling algorithm in some GNN models like in [9]. As the name implies, a random walker in **RWR** is associated with a restart probability c. At each hop, a walker either returns to its starting node with probability c, or visits one of its neighbors with probability 1-c. The transition probability is

$$P_e(n_i) = \begin{cases} c, & \text{if } n_{i+1} = n_1 \\ (1-c)\cdot\dfrac{1}{|N(n_i)|}, & \text{if } n_{i+1} \in N(n_i). \end{cases} \tag{2}$$

For instance, the restart probability c is set to 0.15 in [8] and 0.5 in [9]. Unlike the case in [8], where the considered graph is super-large stored in distributed systems, we target mid-scale graphs in this work. For this reason, there is no need to break a long **RWR** path into multiple shorter ones where each walk is terminated after a returning decision. Instead, we adopt a fixed-length policy that allows a walker to continue after revisiting the starting vertex, until it reaches the pre-fixed walk length.

Meta-path: Unlike the aforementioned methods, the **Meta-path** [6] method aims to generate a series of random walks from a given heterogeneous graph following a pre-defined meta-path scheme $\mathcal{P} = T_1 \rightarrow T_2 \rightarrow \ldots T_i \rightarrow T_{i+1} \rightarrow \ldots T_l$, where $\{T_i\}$ is a sequence of node types associated with each hop. Let $\phi(\cdot)$ denote the node type mapping function, the transition probability can be expressed as

$$P_e(n_i) = \begin{cases} \dfrac{1}{|N_{T_{i+1}}(n_i)|}, & \text{if } \phi(n_{i+1}) = T_{i+1} \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Where $N_{T_{i+1}}(n_i)$ denotes the vertex subset of n_i's (with type T_i) neighbors that are of type T_{i+1}. Generally, a complete meta-path scheme for the entire walk can be obtained by repeating a short scheme. For example, in a citation graph, the A-P-A scheme indicates that a walker should visit an author node and a paper node in an interleaved manner.

C. Target Platform

The target platform is the Xilinx Alveo U280 board. It owns an 8 GiB HBM memory comprised of two HBM2 stacks, each with 8 memory channels (MCs) and 16 pseudo channels (PCs). It also has two 16 GiB DDR4 DIMM memory modules. We believe the hybrid memory setting is most suitable for *walker-based sampling*. First, the HBM devices expose concurrent memory interfaces to facilitate parallel walkers. Second, the capacity of traditional DDR banks is sufficient to accommodate sampled paths, and the sequential storing pattern of generated paths helps to efficiently utilize the DDR bandwidth.

In the HBM subsystem, every two pairwise PCs share a memory controller and correspond to 512 MiB storage. From the users side, each PC has an independent AXI3 interface [10]. To enable flexible HBM access, Xilinx also provides a built-in switch network IP to facilitate efficient interconnect across all 32 PCs. In this way, the 8 GiB HBM memory space is organized as a unified address pool and any AXI interface can be configured to access data residing in an arbitrary PC.

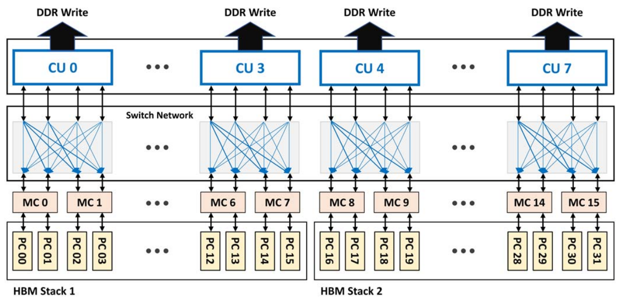

Fig. 1. Graph sampling accelerator with 8 CUs

III. System Design

The parallelizable nature of *walker-based sampling* inspires us to devise nested parallelism, which aims to fully utilize all HBM PCs. The outer parallelism comes from multiple computing units (CUs) that are allocated with different subsets of starting vertices. Within each CU, multiple walkers sample concurrently in one iteration, forming the inner parallelism.

A. System Overview

Fig. 1 shows the system diagram. Without loss of generality, each CU is connected with 4 AXI interfaces corresponding to 4 consecutive HBM PCs. All PCs operate in read-only mode and one of the 16 GiB DDR4 banks is responsible for storing the generated walks from multiple CUs.

A key observation here is that different configurations of the switch network would greatly affect the scalability and performance of memory access in HBM. Typical configurations include point-to-point (P2P) connection, 4×4 connection, 8×8 connection and all-to-all connection [11]. To avoid access interference among multiple PCs, the input graph is encouraged to be replicated in a distributed memory manner, where the extreme case is the P2P configuration. However, with more fine-grained configuration, the system scalability would be highly restricted, resulting in a smaller memory capacity reserved for each copy.

In this design, the 4×4 configuration is adopted as a trade-off between graph scalability and HBM performance. Specifically, any read port from a CU is allowed to access data in any one of the four associated PCs. In this way, the maximum size of the supported input graph is extended to ~1 GiB. As for the influence of the 4×4 interconnect on HBM performance, it is shown that accessing data from any of the four AXI interfaces is equally fast [12]. In the worst case, where simultaneous read requests from all the four AXI interfaces are served, the interference among multiple PCs might lead to slightly longer latency [13]. However, this cost is acceptable for larger graphs detailed by our sensitivity evaluations.

B. CU Architecture

As shown in Fig. 2, a CU is comprised of the processing logic, AXI read/write engines and memory primitives. The processing logic mainly includes the *Pseudo Random Number Generator* (**PRNG**) and the *Degree-aware Sampler* (**DAS**). In essense, the **PRNG** serve as the source of randomness and the DAS is responsible for randomly selecting a next-hop node from the neighbor list.

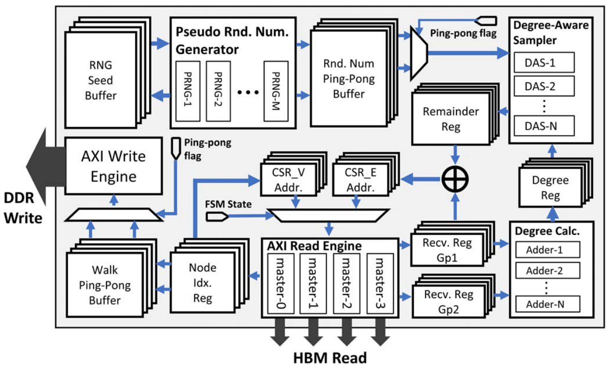

Fig. 2. Computing unit (CU) architecture

Concretely, we implement an RTL version of the *Complementary Multiply With Carry* (CMWC) PRNG following SDAccel examples [14]. Each **PRNG** generates one unsigned 32-bit random number at a time and updates the random seed in the **RNG Seed Buffer**. To reduce the resource cost, multiple walkers share one **PRNG**. However, this does not hurt randomness since we guarantee that the random number used by each walker is unique. A DAS fetchs a random number from the **Rnd. Num. Ping-Pong Buffer**, together with the degree of the current residing node, from the **Degree Reg.**, and outputs a remainder indicating the next-hop node to the **Remainder Reg.**. Each walker is associated with an independent **DAS** since in most cases their degrees are different.

The **AXI Read Engine** is responsible for interfacing with 4 associated PCs. Within the engine, each master retrieves the data requested by its corresponding walkers (walkers served by the same master are in one group) through merged multi-thread AXI read transactions, and stores it to the temporal **Node Idx. Reg.**, which will be concatenated and stored to the **Walk Ping-Pong Buffer**. When the walkers from the same group finish the current sampling iteration, the walk paths will be read out from **Walk Ping-Pong Buffer** and transferred to the global DDR bank through the **AXI Write Engine**.

The memory primitives include on-chip BRAM, URAM and registers for temporal data storage. Apart form the previously mentioned memory blocks, the **CSR_V Addr.** and **CSR_E Addr.** registers are responsible for forming the corresponding read address requested by each walker. The **Recv. Reg. Gp1** and **Recv. Reg. Gp2** registers are used to accomodate two consecutive elements from the CSR vertex array, which will be fed into the **Degree Calc.** block to derive the outgoing degree of the current node for each walker. In our parallel settings, the hardware resources required by each walker are identical, while the modular design does not prevent flexible resource sharing. For example, two walkers may share a BRAM by visiting its two true dual-port interface with non-overlapping addresses.

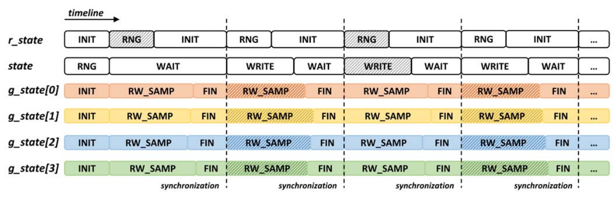

Fig. 3. Graph sampling control flow

C. Parallel Walkers Mapping

The most straightforward way of placing multiple walkers inside one CU is to assign each walker to an AXI read master, which brings 4 walkers in a CU. However, such an arrangement cannot guarantee high throughput since the HBM device and AXI protocol encourage burst-based massive consecutive data access. In our case, each walker merely requests one or two items at a time, which results in extremely low memory bandwidth since the clock cycles spent for receiving data cannot amortize the handshake cost. To address the problem, we leverage the multi-thread AXI read master to mitigate the communication overhead.

For a multi-thread AXI read transaction, the master firstly transfers the read addresses and their corresponding thread IDs in the read address channel. When the memory slave is ready, it returns the requested data together with thread IDs in the read data channel. Ultimately, the *read last* signal indicates the termination of one transaction.

In this design, we map each walker to an AXI read thread, thus a batch of read requests from multiple walkers can be merged into a single AXI read transaction (with different IDs), which significantly amortizes the handshake cost and increases the effective bandwidth obtained. In this way, the number of concurrent walkers within a CU is $\#threads \times 4$. The exact thread number is decided by many factors, one of which is the width of the ID signals ($arid$ and rid). According to the HBM controller documentation [12], the supported signal width can be up to 6 on the target platform, which entails a maximum of $2^6 = 64$ threads in total. However, this upper-bound is not achieved in actual implementation considering the overall hardware resource costs and timing difficulties. We will present a brief exploration of the design space later.

D. Control Flow

The control flow of the proposed accelerator is depicted by Fig. 3. As can be seen, the control flow is handled through six FSMs. Specifically, *state* stands for the main FSM state, *r_state* serves as the FSM state for PRNG, and *g_state[0-3]* are the FSM states for the four walker groups.

We adopt a 3-stage pipeline and double buffering design to hide clock cycles and enhance the system throughput. Here, the shaded boxes are used to indicate the data accesses to different ping-pong buffer groups to remove the data dependency.

To launch the pipeline, the PRNGs firstly generate a stream of random numbers (*RNG*) and put them in the on-chip buffer. After that, four walker groups start sampling (*RW_SAMP*) and the PRNG block starts generating another stream of random numbers for the next sampling iteration. When all the walker groups finish the current iteration, the main FSM activates the writing logic (*WRITE*) and the generated walks are written back to the DDR memory to spare the on-chip walk buffer. Meanwhile, the next sampling iteration is not stalled because of the ping-pong walk buffers. Such an arrangement not only hides the clock cycles for *RNG* and *WRITE*, but also keeps the AXI Read Engine busy almost all the time, which ensures high utilization of the HBM bandwidth.

It should be noted that each walker group possesses an independent sub-flow controlled by a separate FSM (*g_state*) as the 4 AXI interfaces within a CU are stand-alone. However, since all walker groups share a single AXI write master, they need to be synchronized at the end of each sampling iteration. More precisely, the main FSM would switch to *WRITE* after *g_state[0-3]* all reach the *FIN* state.

However, sharing one unified control flow among walkers in the same group is not ideal in the **RWR** case. Consider one individual walker within a group as an example. When a restart happens, the data fetched in the corresponding thread will be omitted and the bandwidth is wasted. To mitigate this issue, we modify the control logic to enable divergent progresses for concurrent walkers in the same group. During a restart of a walker, the hop counter increments and an extra memory access can be arranged if the next-hop decision is not restart.

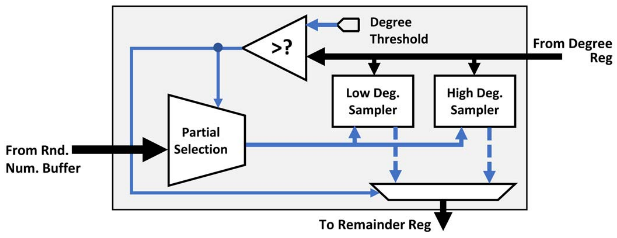

Fig. 4. Degree-aware sampler

E. Degree-aware Sampler

The basic sampling operator used in *walker-based sampling* is the uniform random sampler, which means selecting one of the adjacent nodes from the neighbor list uniformly at random. In practice, it can be achieved by calculating the remainder, where the dividend is a random number and the divisor is the out-degree of the current residing node.

In a CPU/GPU based implementation, the remainder calculation induces the time-consuming integer division instruction. However, unlike general-purpose processors that are driven by instructions, FPGAs are intrinsically data-driven and the sampling logic can be optimized with customized circuits. We propose a DAS based on the classic division by subtraction algorithm, which is illustrated in Fig. 4.

The key observations behind the DAS are as follows. First, the running time of the sampler is closely related to the bit-width of the dividend, which is a pseudo random number in our case. Second, the running time of a general divider is long in case of the minimum possible divisor. If we were able to have knowledge about the approximate range of the divisor in advance, the running time could be further reduced.

In the proposed DAS, we first set a degree threshold, which is compared against the actual degree before division. The degree threshold is customized by the input graph to balance the path latency between the low-degree sampler and high-degree sampler. This can be achieved by estimating the median degree according to the minimum degree and maximum degree recorded when constructing the CSR vertex array with neglectable overhead.

If the current degree is low, the sampling will be handled by the low degree sampler, otherwise by the high degree sampler. Either sampler is essentially an unsigned integer divider, with the output being the remainder. The difference mainly lies in the bit-width of the input operands. In the low degree sampler, the dividend is a random number with a shorter bit-width as a small value can ensure the randomness and naturally leads to fewer subtractions, while in the high degree sampler, the dividend is of longer bit-width to ensure sufficient subtractions. However, since the divisor is large, the number of subtractions is noticeably reduced. Consequently, the running time of the sampler decreases in both cases owing to fewer subtractions.

IV. Experiments

A. Methodology and System Settings

HW & SW Settings: The target platform is a Xilinx Alveo U280 FPGA plugged to a PCIe slot. The host PC owns a Xeon Sliver 4110 @2.1 GHz CPU with 256 GiB DDR4 memory running on a Ubuntu 18.04 LTS system. We implement the proposed design in Verilog HDL by the Vitis 2020.2 toolkit.

Input Graph: As most of the real-world graphs are sparse, we adopt the *compressed sparse row* (CSR) for graph storage. Each undirected edge is stored twice to avoid sink nodes. For efficient data organization, nodes from the AMiner graph are re-indexed to ensure a compact CSR vertex array. Actually, the overhead of such pre-processing is neglectable compared to other commonly used tricks like edge sorting.

The basic features of the adopted real-world graphs are as follows. The Google graph [15] used in both **RW** and **RWR** sampling is a homogeneous web graph from the Google programming contest 2002. It has 875,713 web-page nodes and 5,105,039 edges indicating the hyperlinks in between. The AMiner graph [16] used in **Meta-path** sampling is a heterogeneous academic graph, which is comprised of 3,883 conference nodes (C), 1,693,531 author nodes (A) and 3,194,405 paper nodes (P). We directly access it from [6] to keep it consistent with our evaluation. For **Meta-path** sampling, a *C-A-C* scheme is adopted as the practice in [6]. The original edges include P-A links and P-C links and each paper node owns a single P-C link as it belongs to a sole conference. On this basis, a bipartite graph between conference nodes and author nodes is built by constructing the C-A CSR and A-C CSR. The ultimate number of C-A/A-C edges is 9,323,739.

HBM PC Mapping: We adopt a static memory mapping policy to specify at which PC the input data resides. Consider ***CU0*** and its associated ***PC0 - PC3*** as an example. For **RW**, the CSR vertex array is mapped to ***PC0*** and the remaining

TABLE I: Experimental parameter settings

Algorithm	Parameter	Generated Size
RW	$w = 32, l = 80$	8.35 GiB
RWR	$w = 32, l = 80, c = 0.15$	8.35 GiB
Meta-path	$w = 1,024, l = 200$	2.98 GiB

TABLE II: Parallelism settings and resource utilization

Algorithm	#CUs×#gps/CU ×#wkrs/gp	LUT	REG	BRAM	URAM	DSP
RW	$8\times4\times16 = 512$	25.05 %	15.89 %	31.60 %	20.00 %	1.42 %
RWR	$8\times4\times16 = 512$	27.49 %	16.51 %	31.60 %	20.00 %	1.42 %
Meta-path	$7\times4\times16 = 448$	51.72 %	20.40 %	41.00 %	35.00 %	1.24 %

PC1 - PC3 are reserved for the CSR edge array, as in most cases $|E|>>|V|$ (although only ***PC1*** is occupied). For **RWR**, each PC keeps a replica of the Google graph as the aforementioned complicated control logic leads to significant frequency drop and performance degredation. As for **Meta-path** on the AMiner graph, we notice that the C-A CSR vertex array is short but frequently visited. Thus, we map it to ***PC0***, and load them to the on-chip URAM as the static cache. The remaining A-C CSR vertex array, C-A CSR edge array and A-C CSR edge array is mapped to ***PC1***, ***PC2*** and ***PC3***, respectively.

B. Experimental Configurations

Parameter Settings: To avoid possible influence brought by a biased selection of starting vertices, each vertex is equally likely to be set as the starting vertex by iterating over the whole vertex set V. For **Meta-path** sampling, the iteration is performed on the conference vertex set V_C. The algorithm-specific experimental parameter settings are shown in Table I. For **RW** and **RWR**, the walk length l follows the practice in [7]. To make full use of the available parallelism and assume the generated walks could fit in one DDR4 memory bank of 16 GiB, we set w to be 32. The parameters for **Meta-path** are almost identical to those in [6], except that we set w to be 1,024 instead of 1,000 to make it a multiple of 32.

Design Space Exploration: Owing to the modular design, the accelerator can be easily scaled out and the overall parallelism is upper-bounded by the hardware resource budget. However, higher parallelism does not guarantee better performance, since: (1) The throughput of the HBM memory controller does not scale up linearly with an increased number of AXI read threads; (2) Meeting timing constraints becomes more difficult under higher resource utilization. Specifically, as mentioned in [17], when the on-chip memory in SLR0 is used up, the URAM/BRAM cells from SLR1/SLR2 are invoked, which would significantly increase the length of the critical path. Given the absence of an accurate existing model and limited design search space, we conduct a brute force search on several design candidates, and report the optimal parallelism settings and their corresponding hardware utilization in Table II. Specifically, the second column lists the overall number of concurrent walkers as the production of $\#CUs$, $\#groups\ per\ CU$ and $\#walkers\ per\ group$. As can be seen, the best design point is reached when the induced hardware resources can be roughly covered by the user budget in SLR0. Higher parallelism actually leads to degraded performance due to lower clock frequency. The accelerator for **RW**, **RWR** and **Meta-path** runs at 187 MHz, 125 MHz and 119 MHz, respectively.

C. Baseline Setup

We implement several baseline implementations of the three sampling algorithms for comparison, including:

- A parallel CPU baseline implemented in the Pytorch Geometric [18] graph learning framework (marked as **PyG**).
- A parallel GPU baseline using CUDA (marked as **V100**).
- A parallel GPU baseline using CUDA with limited 512 current random walkers (marked as **V100-512**).

The **PyG** baseline is evaluated on an Intel Xeon E5-2682v4 @2.5 GHz CPU with a 4-channel 128 GiB DDR4 2400 memory. We widely test various batch size settings to find the optimal ones, namely 512, 256 and 16 for **RW**, **RWR** and **Meta-path**, respectively. The parallelism of **PyG** lies in multiple workers/threads in the *DataLoader* class: each worker loads a batch and returns it once the data is ready. We use 32 workers to get the state-of-the-art result, and the number of workers is equal to the logic core of the target Xeon CPU.

The GPU baselines are implemented with CUDA 11.0 on the NVIDIA Tesla V100 GPU with a 16 GiB HBM2 memory running at 877 MHz. For the **V100** baseline, the profiling summaries report 100 %, 100 %, 50 % theoretical warp occupancy and 92.89 %, 92.26 %, 48.38 % achieved occupancy for **RW**, **RWR** and **Meta-path**, respectively. This means that the achieved peak number of concurrent threads per *streaming multiprocessor* (SM) is 2048, 2048 and 1024 for **RW**, **RWR** and **Meta-path**, respectively, with 84 SMs in total.

To evaluate the CPU power, we leverage the *Running Average Power Limit* (RAPL) [19] feature to get estimated results as the actual power is difficult to be obtained on a packed CPU chip. We modify the script from [20] to read the *model specific registers* (MSR) once per second and make sure the measurement is able to cover a complete lifetime of the baseline executions, where the precise timestamps are recorded by the system UNIX time. The run-time GPU power is captured by the *nvidia-smi* command every 20 ms.

D. Results

Figure 5 shows the execution time of the covered three sampling algorithms, where the data transfer time from the on-board FPGA/GPU memory to the host PC memory is excluded. As can be seen, both **V100** and our design achieve significant speedup over the **PyG** baseline owing to the adopted HBM. Consequently, the proposed design achieves a **1.39 ×-3.74 ×** speedup over **PyG**. Among all the parallel implementations, **V100** achieves the best result. However, when we limit the overall parallelism of the V100 implementation to be 512 concurrent walkers, it is observed that the proposed accelerator outperforms the GPU baseline (**V100-512**) significantly. In fact, when full parallelism is achieved on V100, although most warps are stalled due to cache miss, divergent control flow among warps allows overlapping between computation and memory access, and brings more data per unit time on average. However, the essential memory bottleneck limits the achievable parallel efficiency and GPU performance.

Figure 6 shows the energy consumption breakdown of different implementations on the target three algorithms, where the static energy consumption is estimated by the production of no-load power and execution time, with the dynamic energy consumption being the remaining part. Obviously, the traditional CPU-DRAM-based system solution (**PyG**) is at a great disadvantage in terms of energy efficiency for this kind of application. It is observed that the proposed accelerator achieves a **2.42 ×-6.69 ×** higher energy efficiency over **PyG**. Overall, the **V100** implementation achieves the highest energy efficiency. However, the proposed design achieves comparable energy efficiency in terms of dynamic energy consumption. Regardless of the node technology difference (12 nm for V100 and 16 nm for Alveo U280), the proposed architecture could still be promising as our hardware utilization is quite low and a properly tailored accelerator requires much lower static energy.

E. Sensitivity Test

We conduct sensitivity tests to evaluate the design robustness. One of the tests considers the effect of the input graph size on **RW**. The graphs for the tests are generated with a uniform degree of 6, and the size of the vertex set ranges from 1.6M to 25.6M (thus, the graph size ranges from 43 MiB to 684 MiB). The memory mapping is the same as the previous **RW** setting. When the CSR edge array exceeds the capacity of a single PC, it can be split into two or three PCs thanks to the built-in interconnect. Nodes with an index smaller than 800,000 comprise the starting vertex set; thus, the amounts of overall workload for different input graphs are identical. As shown in Fig. 7, the execution time of **PyG** climbs rapidly after the input graph size exceeds a threshold between the **6.4M** group and the **12.8M** group.

The **V100** baseline shows a slightly increased execution time with the growth of the graph size. In fact, we observe the drop in the L2 cache load hit rate from 8.67 % to 3.31 % from the report of the NVIDIA Nsight Compute profiling tool. The increased execution time of our design on large graphs is due to the access interference among multiple PCs. Particularly, for the **1.6M, 3.2M** and **6.4M** groups, the spent time remains unchanged. However, when the size is increased to **12.8M**, the consumed time increases by around 0.9 seconds, as the CSR edge array exceeds a single PC's capacity and is stored in two PCs, which brings a more complicated operation mode in the underlying switch network. This also applies to the **25.6M** case, where the execution time is further increased since the CSR edge array occupies all 3 of the allocated PCs.

We also investigate the influence of restart probability on **RWR**. The memory mapping follows the above-mentioned **RWR**'s practice. The restart probability ranges from 0 to 1 with the interval being 0.1. The maximum probability is set to 0.996, otherwise there will be no randomness at all and the generated walks will be meaningless. The detailed results are shown in Fig. 8. To conclude, the execution time significantly drops with a higher restart probability for **PyG**, as the chance

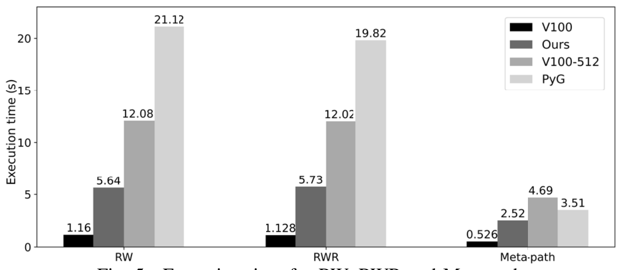

Fig. 5. Execution time for RW, RWR and Meta-path

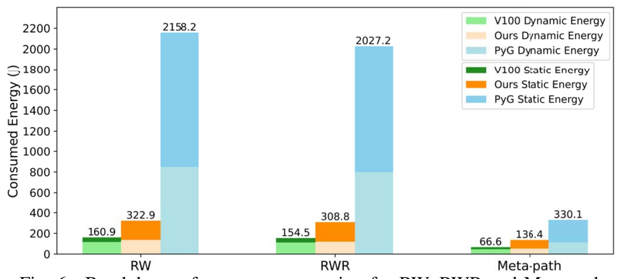

Fig. 6. Breakdown of energy consumption for RW, RWR and Meta-path

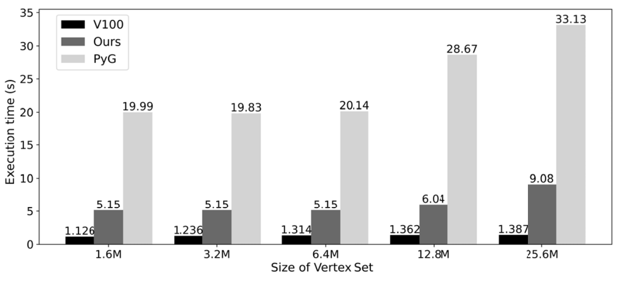

Fig. 7. Sensitivity test on graph size

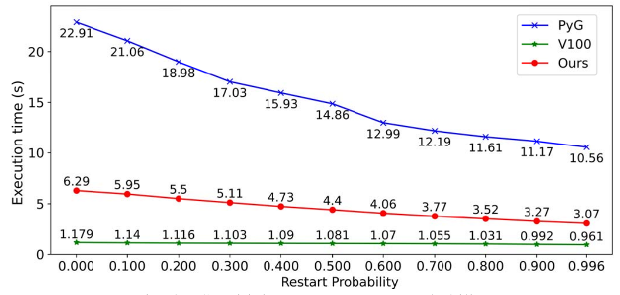

Fig. 8. Sensitivity test on restart probability

of data reuse increases. Owing to the above mentioned modfied control logic, our design achieves good linearity, as depcited by the red line. The **V100** implementation is less sensitive to the restart probability, which we believe is due to the GPU's SIMT model. Within one warp, if any walker (thread) refuses to restart, the other returning walkers (threads) will also be stalled until all the data required by the current warp is ready.

V. Related Work

HBM on FPGA: Shuhai [13] is a pioneer work for benchmarking HBM on FPGAs. It shows that HBM favors a large burst size and good memory locality. It also mentions that the interference among HBM channels leads to decreased throughput, which is consistent with our sensitivity evaluation. The authors of [17] studied the benefits of HBM on FPGAs and show the performance improvement on database applications. **HBM-Connect** [21] boosts two memory bound applications on the U280 board by inserting a customized interconnect and rearranging memory access. However, the method is not applicable to our case as our memory accesses to the same PC are not on consecutive addresses.

Graph Learning Frameworks: PyG [18] and **DGL** [22] are two of the most popular GNN frameworks, which aim to provide fast implementation of GNN models for developers. To deploy GNN on large graphs, Zhu et al. proposed *AliGraph* [23], a comprehensive graph learning platform for distributed systems targeting real-world graphs. For the sampling phase in GNN models, all the aforementioned frameworks deploy it on general-purpose CPUs while GPU implementations for sampling are not quite ready.

Graph Sampling on CPUs: KnightKing [7] is the most relevant work to our design with the main differences being twofold. First, KnightKing targets distributed systems and large graphs, while our design targets a single-node system with the potential to scale out. Second, KnightKing utilizes the rejection sampling method to reduce the computation complexity, while our work mainly focuses on the throughput and energy efficiency. Theoretically, the two optimizations are orthogonal and the combination of both is possible.

VI. Conclusion and Future Work

We presented a graph sampling accelerator on an HBM-enabled FPGA. It achieves a **1.39×-3.74×** speedup and a **2.42×-6.69×** higher energy efficiency over highly optimized CPU baselines on three walker-based sampling algorithms. We also implemented the covered three algorithms on an NVIDIA V100 GPU and compared the performance of these two HBM-enabled platforms. Overall, the V100 GPU provides a better performance at much higher parallelism, and our design achieves comparable energy efficiency in terms of dynamic energy consumption. Moreover, under the same parallelism settings, our design can outperform a V100 GPU and provides higher speedup over CPU-DRAM-based implementations.

In future work, we will optimize the timing to improve the clock frequency. We noticed some recent work on optimizing the floorplanning and placement on multi-die FPGAs [24], which could potentially help to boost the system performance. Furthermore, we will explore the HLS-based development flow to facilitate easy deployment.

Acknowledgement

We thank Jie Sun from Zhejiang University for his advice on the V100 GPU baseline implementation and the anonymous reviewers for their insightful comments and remarks. We thank Xilinx for the generous board donation. This research was partially supported by ACCESS — AI Chip Center for Emerging Smart Systems, Hong Kong SAR and RGC GRF funding No. 16215319.

References

[1] W. Hamilton, Z. Ying, and J. Leskovec, Inductive representation learning on large graphs, in *Proc. NIPS*, 2017, pp. 10241034.

[2] J. Chen, T. Ma, and C. Xiao, FastGCN: Fast learning with graph convolutional networks via importance sampling, in *Proc. ICLR*, 2018, pp. 115.

[3] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, Layer-dependent importance sampling for training deep and large graph convolutional networks, in *Proc. NIPS*, 2019, pp. 1124711256.

[4] B. Perozzi, R. Al-Rfou, and S. Skiena, DeepWalk: Online learning of social representations, in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2014, pp. 701710.

[5] A. Grover and J. Leskovec, Node2vec: Scalable feature learning for networks, in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2016, pp. 855864.

[6] Y. Dong, N. V. Chawla, and A. Swami, metapath2vec: Scalable representation learning for heterogeneous networks, in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2017, pp. 135144.

[7] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, KnightKing: A Fast Distributed Graph Random Walk Engine, in *Proc. 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 524537.

[8] Q. Liu, Z. Li, J. Lui, and J. Cheng, Powerwalk: Scalable personalized pagerank via random walks with vertex-centric decomposition, in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, 2016, pp. 195204.

[9] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, Heterogeneous graph neural network, in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2019, pp. 793803.

[10] ARM. 2011. *AMBA AXI and ACE Protocol Specification AXI3, AXI4 and AXI4-Lite, ACE and ACE-Lite.* www.arm.com

[11] C. Riley. "Basic Tutorial for Maximizing Memory Bandwidth with Vitis and Xilinx UltraScale+ HBM Devices," Nov. 2019, Accessed: Mar. 20, 2021. [Online]. Available: https://developer.xilinx.com/en/articles/maximizing-memory-bandwidth-with-vitis-and-xilinx-ultrascale-hbm-devices.html.

[12] "AXI HBM IP Documentation by Xilinx," Accessed: Mar. 20, 2021. [Online]. https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.

[13] Z. Wang, H. Huang, J. Zhang, and G. Alonso, Shuhai: Benchmarking High Bandwidth Memory on FPGAs, in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* IEEE, 2020.

[14] "Pseudo Random Number Generator by Xilinx," Accessed: Mar. 20, 2021. [Online]. https://github.com/Xilinx/SDAccel_Examples/tree/1e273f6ef01073f878a4c2b5ca4d6ad5aec7e616/acceleration/prng

[15] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters, Internet Mathematics, vol. 6, no. 1, pp. 29123, 2009.

[16] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su, Arnetminer: Extraction and mining of academic social networks, in *Proc. 8th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2008, pp. 990998.

[17] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis and G. Alonso, High Bandwidth Memory on FPGAs: A Data Analytics Perspective, in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL).* IEEE, 2020.

[18] M. Fey and J. E. Lenssen, Fast graph representation learning with PyTorch Geometric, in *ICLR Workshop on Representation Learning on Graphs and Manifolds,* 2019.

[19] H. David, E. Gorbatov, Ulf R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. in *Proc. 16th ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, 2010, pp. 189194.

[20] S. Desrochers, C. Paradis, and V. M. Weaver, A validation of dram RAPL power measurements, in *Proc. 2nd International Symposium on Memory Systems*, 2016, pp. 455470.

[21] Y. Choi, Y. Chi, W. Qiao, N. Samardzic and J. Cong, HBM Connect: High-Performance HLS Interconnect for FPGA HBM, in *2021 29th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 116-126.

[22] M. Wang et al., Deep Graph Library: A Graph-Centric, HighlyPerformant Package for Graph Neural Networks, *arXiv:1909.01315[cs, stat]*, Aug. 2020, Accessed: Mar. 20, 2021. [Online]. Available: http://arxiv.org/abs/1909.01315.

[23] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, Aligraph: A comprehensive graph neural network platform, in *Proc. 45th International Conference on Very Large Data Bases (VLDB)*, 2019, pp. 20942105.

[24] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang and J. Cong, "AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs," in *2021 29th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 81-92.