2018 IEEE Computer Society Annual Symposium on VLSI

Very Large-Scale and Node-Heavy Graph Analytics with Heterogeneous FPGA+CPU Computing Platform

Yu Zou and Mingjie Lin
Department of ECE, University of Central Florida, Orlando, FL, USA
yuzou@knights.ucf.edu, mingjie@eecs.ucf.edu

***Abstract*—We present a highly scalable approach to constructing a reconfigurable computing engine specifically optimized to perform sophisticated kernel computing on graph-structured data. We choose newly emerged graph convolutional networks (GCNs) as our key benchmark and develop a novel node-heavy edge-centric computing framework for very large-scale graph analytics. Unlike most existing studies, our design and implementation can handle extremely large graph size that well exceeds the on-chip memory capacity of any FPGA+CPU heterogeneous platform, thus can only be stored in hard drive. The most novel aspect of our approach is to enable a completely streaming mode of large vertex and edge data and perform a write-back message updating policy, therefore completely removing any redundant data accesses to IO-expensive hard drive. Additionally, our subgraph sorting scheme can effectively eliminate the performance bottleneck caused by preprocessing in the state-of-art computing framework X-Stream [1, 2].**

To validate our approach, we have implemented our proposed method with a KC705 Xilinx FPGA board and tested it with multiple real-world large-scale data sets. For the largest data set with 210,010 vertices and 1,349,400 edges, our platform achieves 1.87s in total latency, which is approximately 400 times faster than the baseline platform with the state-of-the-art approach [1].

***Keywords*-FPGA; graph analytics; heterogeneous computing;**

I. Introduction

Modern FPGA device, when coupled with the conventional CPU or GPU, provides an unique opportunity to implement application-specific heterogeneous computing that maximizes memory access performance, which is crucial in achieving high computing performance for many important memory-intensive and mission-critical computing applications. Unfortunately, with the advent of big-data graph-based analytics, the challenge of accomplishing high-performance computing with such heterogeneous computing systems (CPU+GPU+FPGA) has been greatly exacerbated by **large data sizes**, **irregular data placement**, and **complicated computing parallelism**. As such, despite holding great promise in leapfrogging our computing power and energy efficiency, heterogeneous computing systems, such as FPGA+CPU platforms, prove to be extremely challenging to be fully utilized in complex graph-based applications, such as data mining, collaborative filtering, and probabilistic graph reasoning. This may explain the fact that, while the research on heterogeneous computing for conventional matrix-based scientific-like applications is extensive, the work on how to conduct heterogeneous computing for emerging big-data graph-based applications is relatively scarce.

The main objective of this paper is to design and implement a highly scalable heterogeneous computing framework that is specifically optimized for very large-scale graph analytics. Our main focus is to achieve ultra-high data access performance when a hierarchy of disparate memory resources are involved. Specifically, we aim at a CPU+FPGA heterogeneous computing platform and a large class of graph analytics involving large data set and perform complex computations on graph-based data, such as the newly emerging graph convolutional neural network in artificial intelligence and machine learning area. More concretely, we attempt to optimize computing performance with the following techniques, which we consider to be essential for any kind of heterogeneous computing with graph-based data.

Prioritizing sequential memory accesses: It is well-known that a huge gap of accessing bandwidth exists between sequential and random accesses for any storage device. Therefore, to achieve high I/O performance, it is essential to sequentially stream data as much as possible no matter which computing framework is used. The importance of sequential accessing is even more crucial for our study because graph analytics applications typically embed large-scale data sets with irregular graph structures that tend to incur more random accesses.

Minimizing redundant data accessing: Again, due to the random nature of data layout in most graph analytics, accessing vertex set typically requires repeatedly accessing the same edge set, or vice versa. Both scenarios will inevitably cause redundant data accesses that seriously hinder the overall computing performance of the target application. In most cases, how to partition the overall graph is critical because the data size of a real-world graph often well exceeds an on-board storage capacity, thus causing so-called off-core graph problem.

Parallelizing processing cores: For most graph analytics applications, if either edge-centric or vertex-centric gather-and-scatter (GAS) computing framework is adopted, parallelizing processing cores is imperative to the overall processing performance. Unfortunately, due to the irregular memory access pattern, it is very challenging to achieve high aggregated throughput even with multiple processing cores because input data set of different processing cores can often overlap or cause conflict.

Minimizing sorting operations: Finally, when computing graph analytics, sorting operations are often used to promote sequential data accesses [3]. However, for large-scale graph-based data sets, sorting operations are quite expensive and also detrimental to effectively tackling dynamic graphs, where both edge and vertex sets can change at run time.

Our proposed computing framework with a heterogeneous CPU-FPGA platform can be readily applied to many other node-centric or edge-centric graph computing tasks. We claim the following contributions:

1) We designed and implemented, to our knowledge, the first functional heterogeneous CPU+FPGA computing

[1] All source code of our hardware implementation and detailed performance models can be found at https://github.com/yzou93/ngraph_fpga.

2159-3477/18/$31.00 ©2018 IEEE
DOI 10.1109/ISVLSI.2018.00121

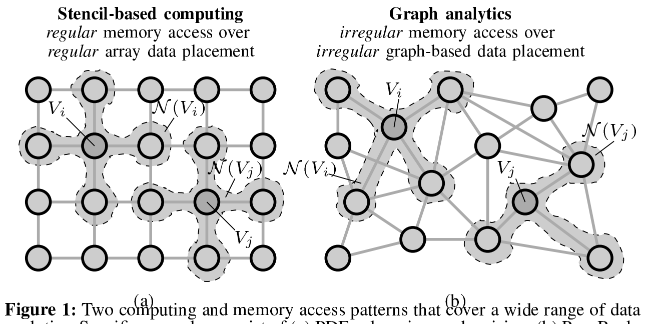
Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2020 at 01:03:51 UTC from IEEE Xplore. Restrictions apply.

platform specifically optimized for a wide range of large-scale and node-heavy graph analytics applications. With very limited hardware resource (≈10% of a Kintex xc7k325t-2ffg900c FPGA chip), our heterogeneous FPGA+CPU computing platform can achieve orders-of-magnitude performance improvement over the state-of-the-art approach for a very large-scale graph analytics application.

2) Although we adopted the basic idea of 2D graph partitioning in GridGraph [4], there are two key innovations. First, each edge partition is sorted according to destination vertex. This change is essential to effective message combining and parallelizing processing cores. Second, unlike the most of other studies, our message data take a write-back policy instead of write-through. This change is important to avoid redundant data loading and reduce data writing. Finally, we stream node data directly from hard drive instead of the standard edge streaming [5, 4, 6, 1, 7]. This proves to be critical for node-heavy graph analytics.

3) Unlike the majority of existing studies, we tackle the case of very large-scale graph analytics, where the size of graph data set well exceeds the capacity limit of on-chip memory and has to be stored in hard drive. As such, optimizing I/O performance to the secondary storage is crucial. We have developed a data streaming scheme for node-heavy data such that no redundant I/O access is needed.

II. Node-Heavy Graph Processing Framework



Stencil-based computing
regular memory access over
regular array data placement
Graph analytics
irregular memory access over
irregular graph-based data placement
V_i
$\mathcal{N}(V_i)$
$\mathcal{N}(V_j)$
V_j
(a)
(b)

Figure 1: Two computing and memory access patterns that cover a wide range of data analytics. Specific examples consist of (a) PDE solver, image denoising. (b) PageRank, collaborative filtering, loopy belief propagation, and Gibbs sampling.

To highlight our computing challenges and present a larger perspective, we abstract a wide range of computer analytics tasks into two categories. Specifically, we depict stencil-based codes as iterative kernels that update array elements according to some fixed pattern in Fig. 1(a), which are most commonly found in computational fluid dynamics (CFD), solving partial differential equations (PDE), and image processing. In contrast, Fig. 1(b) depicts the general form of graph analytics applications, the underlying data layout becomes a generalized form of graph, and the shape of vertex neighborhood varies throughout iterations. This kind of data irregularity can be widely found in collaborative filtering, graph mining, and computer vision, etc. Note that, because all computing patterns depicted in Fig. 1 are defined in a data-centric way in terms of neighborhoods of data structure elements, data locality, essential to achieving high computing performance and energy efficiency, becomes quite challenging to exploit without sophisticated accessing schemes, especially when constrained by memory ports and bandwidth. Our work in this paper specifically aims at accelerating very large-scale graph analytics depicted in Fig. 1(b) and particularly focuses on optimizing the I/O performance for such computing tasks.

A. Edge-Centric Iterative Computing Framework

We follow the standard notations of graph theory, i.e., G, V, and E denote a given graph, its vertex set, and its edge set, respectively. Furthermore, $n = |V|$ represents the total number of vertices, and $m = |E|$ represents G's total number of edges with $e_{i,j}$ denoting an edge with source vertex v_i and destination vertex v_j. Under the computing framework of graph analytics, both vertices and edges can potentially contain computing-related information denoted with $D(v_i)$ or $D(e_k)$ respectively. In this work, we focus on discussing only vertex-centric and edge-centric frameworks. In particular, we adopt the Gather-Apply-Scatter (GAS) graph computation abstraction introduced in PowerGraph [6]. Specifically, the GAS consists of three conceptual phases to define a specific graph computing task: *Gather*, *Apply*, and *Scatter*. During each phase, all vertices in $G(V,E)$ execute in parallel and each vertex v_i exchanges information only with its neighborhood $\mathcal{N}_i$, where $i = 1, 2, , \cdots, n$. In the gather phase, for any given vertex v_i, information stored in all its incoming edges is combined through a predefined and application-specific function $\mathcal{F}(v_i, \mathcal{E}_{i,\text{incoming}})$ is performed, where $\mathcal{E}_{i,\text{incoming}}$ denotes v_i's all incoming edges. In the apply phase, the previously resulted $\mathcal{F}$ value will update $D(v_i)$, i.e., $D(v_i) \leftarrow \mathcal{F}(v_i, \mathcal{E}_{i,\text{incoming}})$. Finally, the scatter phase uses the updated value of v_i to update the data stored on all its outgoing edges $\mathcal{E}_{i,\text{outgoing}}$. Note the GAS computing model is so general that almost all important graph analytics applications can be cast into this framework. For example, three quite different algorithms, PageRank, graph-coloring, and single source shortest path algorithm, can be readily computed with the GAS [6].

Given the GAS computing framework, for large-scale graph processing, there are essentially two types of computing models: vertex-centric vs. edge-centric. The well-known vertex-centric computational paradigm has been recently incorporated into distributed processing frameworks to address computing challenges of irregular memory accesses [7, 8]. This is especially true when real-world billion-node graphs well exceed the memory capacity of typical computing machines and therefore can not be effectively tackled by the mainstream computing framework such as MapReduce, which performs notoriously poorly for iterative graph algorithms such as PageRank. X-Stream [1] pioneered the concept of edge-centric computing framework to scatter-gather process large-scale out-of-core graphs. Although vertex-centric approach [7, 8] is more prevalent for most graph-based problems, the edge-centric approach can completely avoid random access into the set of edges, instead stream them from storage. Therefore, for most graphs that the edge set is much larger than the vertex set, streaming the edges is often advantageous compared to accessing them randomly. Unfortunately, streaming edges will generate the cost of random access into the vertex set. The X-Stream mitigates this issue by partitioning both vertex set and edge set such that each vertex partition fits in the main memory and edges appear in the same partition as their source vertex, and then processing the graph one partition at a time, first reading in its vertex set and then streaming its edge set from storage. In this study, we assume that both edge and vertex data sets are too big to be stored in on-chip or DDR memory. In particular, each graph node contains a large amount of data, thus node-heavy. We will adopt the edge-centric computing

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2020 at 01:03:51 UTC from IEEE Xplore. Restrictions apply.

model and directly stream vertex data set from hard disk.

B. Graph Storage and Partitioning Strategy

Our work focuses on very large-scale graph analytics, meaning that the storage size of a graph, including edge data set and vertex data set, well exceeds the capacity of the on-chip block RAMs and the on-board DRAM, therefore can only be stored in the hard drive. One crucial design decision is how to partition the complete graph into subgraphs with manageable data sizes and still facilitate efficient memory accesses. As in GridGraph [4] and FPGP [5], we first partition the complete vertex set V into P partitions denoted as V_i, $i = 1, 2, 3, \cdots, P$. Subsequently, the edge set E can be partitioned into P^2 subsets $E_{i,j}$, $i, j = 1, 2, \cdots, P$, where $E_{i,j}$ contains all edges with the source vertex u and the sink vertex v such that $u \in V_i$ and $v \in V_j$. Furthermore, the specific choice of P depends on the on-chip memory capacity. Typically, we make sure that one vertex partition can be stored comfortably in the given on-chip block memory.

We adopt the same partitioning method as GridGraph [4] and FPGP [5]. As shown in Fig. 2, all the nodes are partitioned into P disjoint intervals. The parameter P is chosen such that each source node interval can fit the FPGA on-chip block memory. All the edges are partitioned into P^2 number of partitions, E_{ij} contains all the edges whose source nodes belong to interval I_i and destination nodes belong to interval I_j. Each edge partition is associated with a message partition. The message partition stores the generated messages when processing the corresponding edge partition. To reduce disk I/O accesses, in the preprocessing step, edges within each edge partition are sorted in ascending order of destination. The order of generated messages is the same as the order in which edges are streamed. So the generated messages within each message partition are also sorted by destination. A message combiner is used to combine messages with the same destination to one message and write back to hard drive. On average, for each edge partition, the message combiner reduces the number of written messages from $\frac{|E|}{P^2}$ to $\frac{|V|}{P}$.

C. Data Accessing Scheme

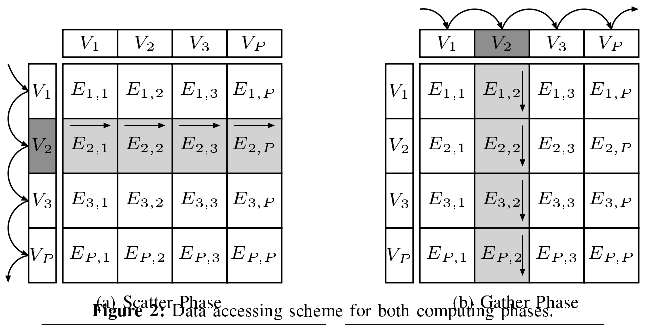


Figure 2: Data accessing scheme for both computing phases.

```
Scatter Phase:

Input: edges $E$, vertices $V$
Output: messages $M$
for $i$ = 1 to $P$ do
    Load $I_i$
    for $j$ = 1 to $P$ do
        $M_{i,j}$ = calculate($I_i$, $I_j$, $E_{i,j}$)
        Save $M_{i,j}$
    end for
end for
```

```
Gather Phase:

Input: messages $M$
Output: updated vertices $V$
for $i$ = 1 to $P$ do
    $I_i$ = 0
    for $j$ = 1 to $P$ do
        $I_i$ = update($I_i$, $M_{j,i}$)
    end for
    save $I_i$
end for
```

Figure 3: Node-heavy graph processing flow.

When processing each edge partition, we adopt the edge-centric graph computing model [1]. In the scatter phase of each iteration, a source vertex partition V_i is loaded into the on-chip block memory. Edge partitions E_{i1} to E_{ij} are streamed in from the PC host. Subsequently, all generated messages are streamed back to the PC host and stored into their corresponding message partitions. This process is repeated P times until all P of source vertex partitions and P^2 of edge partitions are traversed. In the gather phase, all message partitions are streamed in the order of destination. P number of message partitions whose destination nodes belong to V_i destination vertex partition are streamed consecutively to FPGA. After processing all P message partitions, the vertex partition V_i is completely updated and written back to the hard drive. This process is repeated P times until all the P^2 message partitions are traversed. On average, for each message partition, the message combiner reduces the number of message reads from $\frac{|E|}{P^2}$ to $\frac{|V|}{P}$.

III. Hardware Architecture and Implementation

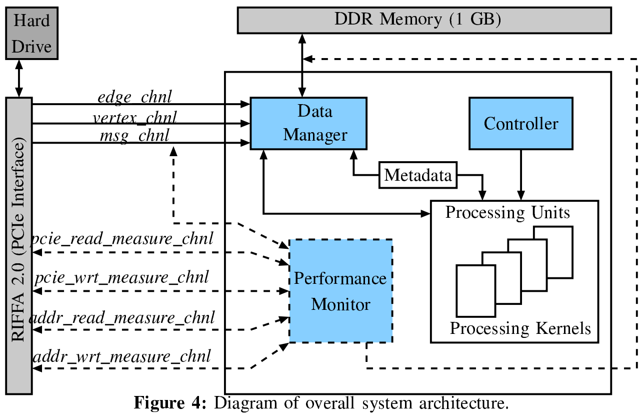


Figure 4: Diagram of overall system architecture.

Overall Architecture—The overall architecture of our implementation is shown in Fig. 4. There are three layers of memory hierarchy: hard drive, DDR memory, and on-chip BRAM (block RAMs). Initially, all data including both edge set and vertex set reside in the hard drive, which has the largest capacity, constant parameters are stored in on-board DRAM. As computation progresses, data will gradually be moved into the accelerator. The hard drive is off-FPGA-board and connected to the FPGA chip through a RIFFA interface [9], which essentially is a hardware wrapper around the PCIe protocol stack IP issued by Xilinx. In contrast, the DRAM is off-chip but on-FPGA-board and communicates with the FPGA device through the Xilinx Memory IP core, a combined pre-engineered controller and physical layer (PHY) for interfacing Kintex architecture to DDR SDRAM. Both the hard drive and the DDR memory are connected to a specifically designed data manager, which coordinates all data movements in our system and directly communicates with all processing cores. In order to enable high throughput data movement and data streaming, we implement three separate I/O channels between the RIFFA interface and the data manager: edge data channel, vertex data channel, and message data channel. To implement control path, we have a controller to coordinate all logic operations and data movements. Inside the processing unit, to improve the overall throughput, we pipeline multiple copies of identical processing kernels. As shown in Fig. 5, the vertex data set

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2020 at 01:03:51 UTC from IEEE Xplore. Restrictions apply.

will be first streamed into on-chip memory, and then edge data set will be streamed in from the hard drive through the RIFFA PCIe interface. In scatter phase, vertex data can be fetched within one or two clock cycles in a random access pattern when processing each edge.

Note that, to facilitate performance measurements, we have an additional circuit block for monitoring performance that records all I/O activities of PCIe read and write operations. These measurements allow us to analyze our data performance in detail and indirectly illustrate the source of performance gain. When measuring the final performance, we remove this performance monitoring block to conserve hardware usage and keep computational fidelity.

Host Implementation—There are eight threads working in parallel on the PC host. Thread 0 is responsible for retrieving vertices from the hard drive and sending data to FPGA through channel 0 in scatter phase. In gather phase, thread 0 receives updated vertices from hardware. Thread 1 sends edge data to FPGA. In the scatter phase, thread 2 receives generated messages from FPGA and stores the messages to the corresponding file. In the gather phase, thread 2 in-order retrieves message partitions from hard drive and streams message data to FPGA. Thread 3 retrieves pre-stored metadata information for each edge partition and streams to FPGA. Thread 4, 5, 6, 7 are used to receive performance measurement data for monitoring data bandwidth. They are only used to analyze the bandwidth utilization, in the real platform, these threads will be removed.

Data Controller—Data controller is responsible for data transfer between the host and FPGA through PCIe. We use RIFFA to handle the communication [9]. RIFFA supports PCIe Generation 2 and 128 bit data interface. Up to 12 channels can be used to fully utilize the bandwidth. In the architecture we use eight channels. For consistency, in the following, "send" and "receive" are from the aspect of FPGA. Channel 0 is used to receive vertex data from host and send calculated vertex data for the next iteration back to host. Channel 1 receives edge data from host. To improve the streaming performance, edge transfer and message transfer use different channels. Channel 2 sends messages, which are generated in the scatter phase, back to the host and receives messages in the gather phase. Channel 3 is used to transfer metadata for each edge partition. When processing each edge partition, only one metadata is used to indicate the number of edges will be streamed in and the number of messages should be streamed out, so the metadata will not bottleneck the processing throughput. Also, putting metadata into hardware memory (either DRAM or on-chip memory) will constrain the scale of data which the system can process, since for P number of node intervals, P^2 number of metadata are needed. Channel 4, 5, 6, 7 are used to transfer performance measurement data to monitor the bandwidth of the system, and they are only for comparison analysis. In the final system, these four channels are removed. In our design, for simplicity, we did not fully utilize all the 12 channels. More channels should cause higher bandwidth utilization. For example, edge data can be scattered to multiple threads, and each thread sends a part of edge data through an individual channel. This will increase the interface width equivalently and consequently increase the PCIe bandwidth utilization.

Processing Pipelining— In our design, we split the data processing into three stages, read, processing, and write.

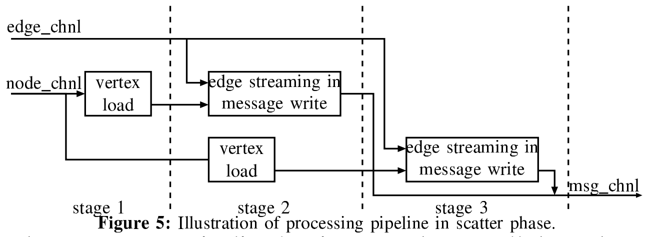


Figure 5: Illustration of processing pipeline in scatter phase.

Three stages are pipelined to improve the overall throughput as shown in Fig. 5. Double buffer is also used to improve throughput of vertex data read.

Message Combiner & Preprocessing—When processing each edge partition, message combiner combines generated messages with the same destination to one message in order to reduce PCIe accesses. In preprocessing, all edges are grouped into P^2 edge partitions. Within each edge partition, edges are sorted by destination. The number of edges and the number of messages are logged to a metadata file. Sorting complexity is $\mathcal{O}\left(P^2 \cdot \frac{|E|}{P^2} \cdot \log \frac{|E|}{P^2}\right)$.

IV. Experimental Setup and Implementation

To solidify our study, we choose the newly emerging Graph Convolutional Neural Networks (GCNs) [10] as our target application. GCN generalizes classical CNNs to handle graph data such as molecular data, point cloud and social networks. Its applications include 1) semi-supervised document classification in citation networks, 2) semi-supervised entity classification in a bipartite graph extracted from a knowledge graph, 3) an evaluation of various graph propagation models, and 4) a run-time analysis on random graphs. Solving GCN not only consists of handling all challenges associated with standard graph analytics but also poses a set of unique challenges induced by the huge data size of vertex set. All data sets we test are listed in Table I [10] and quite representative in terms of memory access irregularity typically found in graph-based analytics. Mathematically, the graph propagation rule of the GCN model can be described as $h_i^{(l+1)} = \sigma(\sum_{j \in N_i} \frac{1}{c_{ij}} h_j^{(l)} W^{(l)})$ and $c_{ij} = \sqrt{d_i d_j}$, where W is a trained weight matrix, d_i denotes the degree of node v_i, and h_i is a feature vector of node v_i [10]. For example, in the documentation classification, each document is described in a bag-of-words feature vector. In the first layer, each bag-of-words feature vector is a sparse vector. However, in the second and following layers, each vector is a dense vector with each element a 32 bit floating number. Therefore, sparse graph processing methods lose the generosity in this problem, even it could benefit when processing the first layer. To be general, we treat each feature vector in the first layer as a dense vector even most of elements are zero. Also, each feature is stored as a 32 bit floating point number.

A. Experimental Setup

Before delving into specifics, we introduce the primary targeting hardware system—a modern integrated FPGA platform depicted in Fig. 6. It is heterogeneous in the sense that such chip platform possesses multiple levels of memory hierarchy with various bandwidth and capacity. Many existing chip platforms fit with the abstraction in Fig. 6. The experiments were conducted on a KC705 evaluation Board with a Xilinx Kintex xc7k325t-2ffg900c FPGA chip.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2020 at 01:03:51 UTC from IEEE Xplore. Restrictions apply.

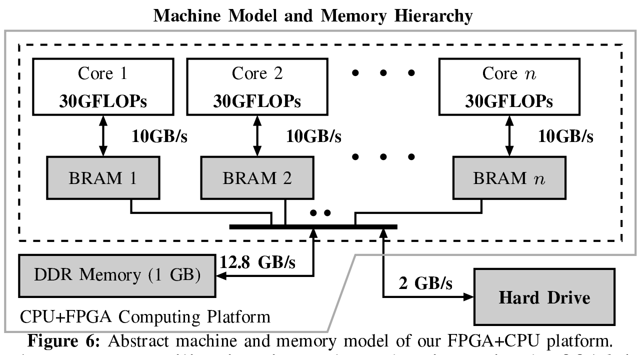


Figure 6: Abstract machine and memory model of our FPGA+CPU platform.

The resource utilization is evaluated using Vivado 2016.4. The processing logic runs at 100MHz clock frequency and RIFFA logic runs at 250MHz clock frequency [9]. We ran three experiments for 4 different datasets [10]. The statistics of each dataset is listed in Table I. In experiments we only focus on the memory accesses, since in the graph convolution network, each node will do a complex matrix multiplication, and matrix multiplication acceleration is a problem-specific optimization. To be general, in the following experiments, we keep the memory access pattern but skip the computation part. We will discuss how to accelerate graph convolution network in our future work.

For the baseline implementation, we extend the FPGP [5] graph processing platform to process node-heavy graph problems. FPGP makes an assumption that edges are stored in hard disk and nodes are stored in DRAM. We assume edges and vertices are all stored in hard drive while keeping the memory access pattern and computing strategy the same as FPGP. The baseline implementation works as described in Algorithm 1 of FPGP [5].

B. Performance Measurement: Latency and I/O Bandwidth

For performance measurements, we consider 4 data sets listed in Table I with variable sizes. The first three data sets—*Citeseer, Cora, and Pubmed*—are collected real-world data. In order to stress-test our computing platform, we also create, with the same principle as in [10], a synthetic *X-Large* data set with more than one million edges. For all these test cases, we use the same FPGA implementation that consumes 13% of LUT, 3% of LUT-RAM, 8% of FFs, and 75% of BRAM resource available on a Kintex xc7k325t-2ffg900c FPGA chip. For comparison, our baseline implementation uses 12% of LUT, 2% of LUT-RAM, 8% of FFs, and 75% of BRAM for the same FPGA device. Also, for each experiment, we adjust P to fully utilize the FPGA on-chip block memory.

As shown in Table I, for each dataset, we measure the latency to finish each iteration. For all the four data sets, the speedup of our node-heavy graph processing platform over the baseline ranges from 10 to 50 times. The biggest contributing factor, we believe, is due to the fact that, during each iteration, the baseline framework needs $P|V|\alpha_{in}$ number of node accesses, while our implementation only needs $|V|\alpha_{in}$ number of accesses. Also, our node-heavy graph processing framework will generate a lot of messages which will decrease the speedup. So theoretically the speedup of our design compared with the baseline implementation should be smaller than but close to P, which is consistent with our measured results on the hardware.

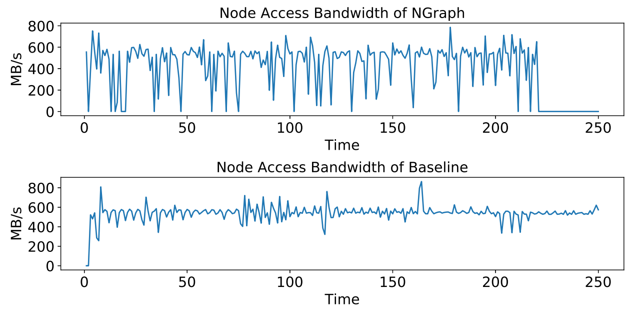


Figure 7: Experimentally measured PCIe I/O traffic patterns for (a) Node-heavy graph processing implementation, and (b) Baseline implementation.

To experimentally validate the above hypothesis, we measure the PCIe bandwidth usage for our node-heavy graph processing framework and the baseline. As shown in Fig. 7 for "Cora" dataset, both designs are bottlenecked by node access bandwidth. Unlike the baseline, the node bandwidth utilization of our design is not always high, because FPGA needs to send generated messages back to host through PCIe after processing each edge partition. More importantly, the total amount of PCIe data traffic is much larger for the baseline implementation than for our processing framework. As such, compared with our design, the baseline takes much longer to finish the processing. To further explain our significant performance gain, we also measure the effect of message combining caused by pre-sorting. We found that, on average, we can reduce the total number of messages by about 2 times by combining. We also found that the benefit of message combining is significantly affected by both the graph sparsity and the partition factor P.

V. Performance Modelling and Analysis

P	Number of vertex partitions
α_{in}	Input vertex data size (in Byte)
α_{out}	Output vertex data size (in Byte)
β	Edge data size (in Byte)
$\lvert V\rvert$	Number of vertices
$\lvert E\rvert$	Number of edges
M_{BRAM}	Storage capacity of FPGA on-chip block memory (in MB)

Table II: Notations of disk I/O model

For node-heavy and large-scale graph analytics, I/O performance to the secondary storage will be crucial to the overall computing performance of a heterogeneous computing platform. In this section, we develop two disk I/O models for our implementation and the baseline implementation respectively for performance comparison. Note that, although we choose hard drive as our secondary storage in developing our performance model, our mathematical model can be readily extended to treating the on-board DRAM as the secondary storage. We use the notations listed in Table II to formulate and derive our performance model. In this study, P, the total number of vertex partitions, is determined by fulfilling the requirement that each vertex partition doesn't exceed the memory capacity of on-chip block RAMs, i.e., $M_{\text{BRAM}} \geq \frac{|V|\alpha_{in}}{P}$. Given that the graph under our consideration is node-heavy and very large-scale, typical P value is large, 100s or even 1000s. Also, when partitioning the graph, we attempt to balance each vertex partition such that each resultant edge set contains almost the same number of edges.

As detailed in Section II-B, our computing algorithm follows the GAS (gather-apply-scatter) principle with two data accessing phases. During the scatter phase, as shown in Fig. 2(a), we need to enumerate through all vertices and

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2020 at 01:03:51 UTC from IEEE Xplore. Restrictions apply.

Dataset	Nodes (Size)	Edges (Size)	# Features Per Node	NGraph Latency	Baseline Latency	# Partitions (P)	Speedup
Citeseer	3327 (49.28 MB)	12431 (99.45 KB)	3703	46.979 ms	453.6 ms	16	10
Cora	2708 (15.52 MB)	13264 (106.11 KB)	1433	0.11 s	4.64 s	64	45
Pubmed	19717 (39.43 MB)	108365 (866.92 KB)	500	0.18 s	5.65 s	39	31
X-Large	210010 (411.23 MB)	1349400 (9321.23 KB)	500	1.87 s	12.6 min	411	404

Table I: Dataset statistics, latency, and speedup of our implementation vs. baseline.

write out messages into their outgoing edges. As such, we sequentially load all vertex partitions and, for each loaded vertex partition V_i with the data size $\frac{|V|\alpha_{in}}{P}$, we stream all edge subsets containing all edges that have vertices in V_i as their sources, i.e., $E_{i,j}$, where $j = 1, 2, \cdots, P$, from PC host. This procedure ensures that all vertices are only loaded once, therefore significantly reducing I/O traffic. Without message combining, the number of generated messages equals the number of edges. Each message carries the data to be used to update the destination vertex partition in the gather phase. In the GCN test case, only accumulate operation is needed when using messages to update destination nodes in the gather phase. Therefore each message carries the same size of data as the output vertex size α_{out}. So the total disk I/O accesses spent on edge streaming-in and message streaming-out will be $\frac{|E|\beta}{P^2} + P \cdot \frac{|E|\alpha_{\text{out}}}{P^2} = \frac{|E|\beta}{P^2} + \frac{|E|\alpha_{\text{out}}}{P}$. According to Algorithm 3, this process will be repeated P times until all the edge partitions can get streamed. Therefore, the total disk I/O needed in the scatter phase can be written as $P\left(\frac{|V|\alpha_{\text{in}}}{P} + \frac{|E|\beta}{P^2} + \frac{|E|\alpha_{\text{out}}}{P}\right)$. In the gather phase, messages associated with edge partitions get processed in the order of the destination partitions they belong to. As depicted in Fig. 2(b), after P number of message partitions $M_{1,i}$ to $M_{P,i}$ are streamed in, the vertex partition V_i is completely updated and streamed out to hard drive. This process is repeated P times until all the P^2 message partitions get processed. The total number of hard drive accesses in the gather phase is $P\left(P \cdot \frac{|E|\alpha_{\text{out}}}{P^2} + \frac{|V|\alpha_{\text{out}}}{P}\right)$. Combining the I/O accesses in both gather and scatter phases, the total disk accesses of our design in each iteration can be expressed as $|E|\beta + |V|\alpha_{\text{in}} + (2|E| + |V|)\alpha_{\text{out}}$. With the similar procedure, we can also derive the baseline I/O access model for our baseline System, which is similar to [4] and FPGP [5]. The total amount of disk I/O accesses for this baseline implementation can be expressed as $P|V|\alpha_{\text{in}} + |E|\beta + |V|\alpha_{\text{out}}$.. Here, we omit details of model derivation for simplicity. Comparing the above two equations, it is clear that our method can significantly reduce the total number of I/O accesses by approximately P times, as long as $P > \frac{2|E|\alpha_{out}}{|V|\alpha_{in}} + 1$. This reduction in I/O accesses clearly translates into the significant performance gain presented in Table I for all benchmark data sets. There are two contributing factors to our performance improvement. First, unlike in the baseline implementation, we adopt the well-known GAS principle for the iterative graph analytics instead of the stream-and-apply approach. We observe that the use of message array can effectively eliminate the redundant loading of vertex data set. Second, unlike in all previous studies, we perform the sorting within each vertex partition, which facilitates the message combining during scatter phase and consequently significantly reduce the total number of messages written into the hard drive. In fact, because P is proportional to $|V|\alpha_{in}$, when processing a real-world large-scale node-heavy graph on a CPU+FPGA heterogeneous computing platform, P will almost always be a very large number.

VI. Conclusion

Many important memory-intensive and mission-critical applications pose huge computing challenges due to their **large data sizes**, **irregular data placement**, and **complicated computing parallelism**. Our study has experimentally demonstrated that emerging FPGA+CPU/GPU heterogeneous platform presents an unique opportunity to maximize memory access performance for such applications through implementing application-specific computing machine.

References

[1] A. Roy, I. Mihailovic, and W. Zwaenepoel, “X-stream: Edge-centric graph processing using streaming partitions,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, (New York, NY, USA), pp. 472–488, ACM, 2013.

[2] A. Gudimella, R. Story, M. Shaker, R. Kong, M. Brown, V. Shnayder, and M. Campos, “Deep reinforcement learning for dexterous manipulation with concept networks,” *CoRR*, vol. abs/1709.06977, 2017.

[3] S. Zhou, C. Chelmis, and V. K. Prasanna, “High-throughput and energy-efficient graph processing on fpga,” in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 103–110, May 2016.

[4] X. Zhu, W. Han, and W. Chen, “Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning.,” in *USENIX Annual Technical Conference*, pp. 375–386, 2015.

[5] G. Dai, Y. Chi, Y. Wang, and H. Yang, “Fpgp: Graph processing framework on fpga a case study of breadth-first search,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 105–110, ACM, 2016.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. 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)*, (Hollywood, CA), pp. 17–30, USENIX, 2012.

[7] V. Kalavri, V. Vlassov, and S. Haridi, “High-level programming abstractions for distributed graph processing,” *CoRR*, vol. abs/1607.02646, 2016.

[8] R. McCune, T. Weninger, and G. R. Madey, “Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing,” *ACM Comput. Surv.*, vol. 48, pp. 25:1–25:39, 2015.

[9] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, “Riffa 2.1: A reusable integration framework for fpga accelerators,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, p. 22, 2015.

[10] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *CoRR*, vol. abs/1609.02907, 2016.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2020 at 01:03:51 UTC from IEEE Xplore. Restrictions apply.