On-The-Fly Parallel Data Shuffling for Graph Processing on OpenCL-based FPGAs

Xinyu Chen[1], Ronak Bajaj[1], Yao Chen[2], Jiong He[3], Bingsheng He[1], Weng-Fai Wong[1], Deming Chen[4]
[1]National University of Singapore, [2]Advanced Digital Sciences Center,
[3]Alibaba Group, [4]University of Illinois at Urbana-Champaign

***Abstract*—Graph processing has attracted much attention recently due to its popularity in many big data analytic applications. With high performance and energy efficiency, FPGAs can be an attractive architecture for graph processing. A number of techniques such as caching using block RAMs (BRAMs) to reduce random accesses of global memory and multiple processing element (PE) instances for high throughput have been explored. OpenCL-based FPGAs natively support a high-level programming paradigm, providing good programmability to developers. However, challenges remain because the run-time dependency introduced by multiple PEs usually cannot be handled efficiently by OpenCL's high-level control granularity. In this paper, we propose a novel on-the-fly parallel data shuffling technique that can be implemented in OpenCL to solve this problem. We have integrated our shuffling technique to an edge-centric graph processing framework which achieves a throughput of more than 1,000 million traversed edges per second (MTEPS) on PageRank, SpMV, BFS and SSSP applications and is even better than existing RTL-based designs.**

I. Introduction

Graph processing is widely used in data analytics, as it can be applied to a variety of application domains such as social networks, cybersecurity, and machine learning [1], [2]. A lot of work is being done to scale and improve the performance by exploiting the parallelism of underlying architectures such as CPUs, GPUs, and FPGAs. The challenge is that graphs are typically unstructured and irregular, making it difficult to extract parallelism [3], [4]. This also results in poor data locality for computation and randomized data access patterns [3], [4].

With the capability of high performance and energy efficiency, FPGAs have become an attractive platform for graph processing [5], [6]. Moreover, high-level synthesis (HLS)-based FPGA development decouples the low-level architecture details for the design and provides easier programmability at the application level [7]–[10]. With its good support for heterogeneous systems and parallel computing, OpenCL has become a popular programming language for HLS tools.

Some common techniques have been developed to solve the poor locality and parallelism issues of graph processing using FPGAs. In order to improve the **data locality**, Block RAMs (BRAMs) are used to cache the properties of vertices of the graph, as the accesses to BRAMs are of low latency and high throughput [6], [8]–[15]. Vertices of a large graph can be partitioned such that vertices of each partition fit into the available BRAMs [13]. To further improve the **parallelism**, multiple processing elements (PEs) are used to process multiple data items in their local BRAM caches [11], [12], [14]. Multiple unordered data words are read for processing and the target PE is determined by data words at runtime. This results in run-time data dependencies that can be either handled efficiently with fine-grained control logic, or avoided by pre-processing and rearranging the data. For the former, the fine-grained control logic required can often only be realized by hardware description languages (HDLs). HLS tools perform static analysis of the high-level description of the algorithms at compile time to generate hardware implementations, and are unable to extract the parallelism with run-time data dependency [16]. The alternative of pre-processing can have high overheads, and may compromise flexibility. In order to solve the problem of run-time data dependencies, this paper makes the following contributions:

- We present a novel OpenCL-based on-the-fly parallel data shuffling technique which handles the run-time data dependencies caused by dispatching data to multiple PEs efficiently.
- We implement an efficient pipelined edge-centric graph processing framework that integrates the proposed shuffling technique on OpenCL-based FPGAs. It has good scalability and can be easily adapted to other graph processing algorithms.
- Our design deliver a state-of-the-art performance in comprehensive evaluations. The framework achieves a throughput at more than 1,000 million traversed edges per second (MTEPS) on average, and up to 2.9× speedup over existing RTL-based works.

II. Background and Related Work

A. OpenCL for FPGA

Traditionally, HDLs like VHDL and Verilog have been used to describe designs for FPGAs. However, this is a time-consuming, error-prone and tedious process which requires in-depth understanding of underlying hardware. To make FPGAs easier to program, FPGA vendors and research communities have been actively working on developing HLS tools, which take a design description in high level languages (HLL) as input and generate a synthesizable hardware implementation for FPGAs [17]–[19]. OpenCL, by taking advantage of the highly abstracted programming model, enables developers to benefit from the efficiency of FPGAs without investing a lot of time and effort into HDL programming, and also expedites the development as well as design space exploration [20].

B. Graph Processing on FPGA

1) GAS model: The Gather-Apply-Scatter (GAS) model [21], [22] is widely used for graph processing frameworks [12]–[14], [23]. A simplified version of the GAS

model is proposed in [22]. It processes edges by propagating from source vertex to destination vertex and the process is broadly divided into three stages: *scatter*, *gather*, and *apply*. In the scatter stage, for each input edge, an update tuple is generated with the format of $\langle destination, value \rangle$. Then, all the update tuples are processed by accumulating the *value* to *destination* vertices in the gather stage. The final apply stage executes an apply function on all the vertices. In our design, we follow this GAS model.

2) Edge-centric and vertex-centric: Based on the access pattern of the edges in a graph, the GAS model can be classified into two categories: edge-centric processing (ECP) [24] and vertex-centric processing (VCP) [14]. For the VCP, edges are randomly accessed through vertices [25]. On the contrary, the ECP reads all edges sequentially, thus avoids costly random memory accesses [26]. The ECP often outperforms the VCP on FPGAs when the number of edges in a graph is much larger than the vertices [24] since the run-time is dominated by the portion of accessing edges from memory [12], [13]. We focus on ECP in our proposed design.

3) Graph Partitioning: In order to improve the locality of gather stage, BRAMs are used to cache the destination vertices [6], [11]–[14]. However, the BRAMs are insufficient to fit all the vertices of a graph. This requires partitioning of vertices of a graph such that each partitioned set can fit into the BRAM. Suppose a graph has vertex identifiers from 1 to V. If BRAMs available on the target FPGA device can cache U vertices, the vertex set can be divided into $\lceil V/U \rceil$ partitions. The i^{th} partition has the vertex set with identifiers ranging from $(i-1)U$ to iU.

4) Related work: A set of edge-centric graph processing works [11]–[13] has been proposed by Shijie et al. Edges are pre-sorted to minimize the row conflicts when accessing global memory. Dai et al. proposed a vertex-centric graph processing framework called ForeGraph [14] for multi-FPGA architecture. However, in their framework, each PE caches the same destination vertices, minimizing the partition size and increasing the overall data replication factor. Yao et al. proposed an accumulator to handle data conflict introduced by updating vertices [27]. All these studies take fine-grained control logic of HDLs or prepossessing to resolve the run-time data dependencies or data conflicts. Kapre et al. used network-on-chip (NoC) [28] as the communication scheme among graph processing soft cores [29] and built a vertex-based framework with a domain-specific language [23], [30]. However, we target OpenCL-based designs in which we would like to have (a) PEs that run independently, and (b) no NoC since even a light-weight NoC will result in unnecessary overhead and complexity for our case.

III. Motivations

In the GAS model, the gather stage consumes the update tuples to update the vertices cached in BRAMs. Multiple PEs are usually instanced to improve the throughput. Each PE accesses a subset of cached vertices such that PEs run independently and access data in parallel, consuming multiple

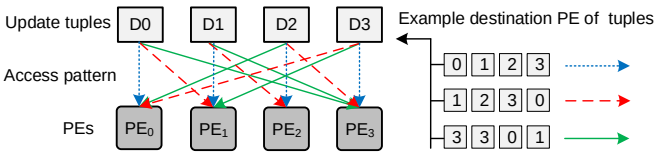


Fig. 1: Access patterns of update tuples.

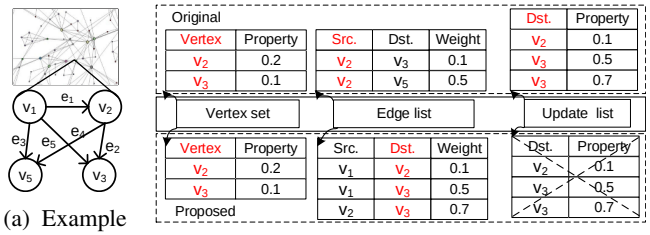
tuples per cycle and avoiding the conflicts. But it requires extra logic to determine the corresponding PE for each update tuple. Thus, run-time data dependency will be introduced since the target PE of each update tuple is determined by its destination vertex. This makes the access pattern of tuples unpredictable at the compile time as it depends on inputs. Fig. 1 shows three possible example patterns represented in the same colored arrows for a configuration with four PEs. Parallelism is hard to extract in this case and conservative hardware implementations are usually implemented to ensure the correctness [16], [31]. We found two solutions in the literature to tackle this problem [32], [33]; however, they do not solve the problem efficiently. Assume there are N tuples read per cycle and N PEs for processing. The throughput is determined by the efficiency of tuple assignment to PEs.

1) **Polling.** A naive solution is conventional polling [32]. Each PE iterates over update tuples one by one and only processes the tuple when destination belongs to it. Thus, the number of clock cycles required equals to the total number of tuples, which is N and the parallelism factor is limited to 1. It will introduce the “bubbles" since some cycles are wasted on checking invalid tuples.
2) **Convergence Kernel.** A recent work in [33] presented a convergence technique (named FPPA here). Each PE uses multiple read engines to read tuples in parallel, and then attempts to write only valid tuples to PE’s local buffer in parallel. A global write address is used to sync next available buffer location between read engines. When a tuple is valid, the corresponding read engine writes the tuple to current address and then increases the address by 1. Thus, data dependency raises since the write address for each read engine cannot be determined at compile time, resulting in unknown access patterns. This further leads to a conservative hardware implementation, with low parallelism. We experimentally demonstrate that the performance of this method is even worse than the Polling method in Section VI.

We present an OpenCL-based shuffling technique which handles this run-time data dependency efficiently, thus enabling each gather PE to run efficiently and independently, maximizing the parallelism. We also present a fully pipelined edge-centric graph processing framework with the integration of the proposed shuffling technique.

IV. System Overview

We propose an edge-centric graph processing framework with following key features: (1)improving the layout of the partitioned data to minimize the global memory access and (2) integrating with the proposed shuffling technique to achieve a highly efficient pipeline.



(a) Example partition

(b) Data layout change

Fig. 2: Data layout of a partition consisting of V_2 and V_3.

A. Improved data layout

Suppose the graph has E edges. In current popular edge-centric frameworks [13], [24], each partition maintains an edge list and an update list. The edge list consists of all the edges whose source vertex is in the partition's vertex set while the update list consists of all updates whose destination vertex is in the partition's vertex set. The scatter stage first processes all the partitions and stores the update tuples to global memory, which is then read by gather stage for further processing. Since the number of update tuples equals to the number of edges, it introduces at least $3E$ times global memory accesses. However, this is not optimal for FPGAs which have limited memory bandwidth to access external memory and cannot benefit from deep pipeline execution of FPGAs. Instead of storing both edge list and update list, we only store the modified edge list which consists of all edges whose destination vertex is in the partition's vertex set, rather than the source vertex of them. The corresponding changes for an example partition (Fig. 2a) are shown in Fig. 2b. With this layout change, the update tuples generated by scatter stage can be streamed to gather stage for further processing, without transferring back to global memory since the destination vertices of these tuples are in the cached vertex set. Finally, the number of global memory access is reduced by $3\times$ (from $3E$ to E) by eliminating update tuple retrieve to global memory.

B. Pipelined execution with shuffle

A high-level overview of our proposed system is shown in Fig. 3. Thanks to our data layout change, the scatter, shuffle and gather stages can be executed in a pipelined manner. For the processing of each partition, a set of vertices is loaded into local memory of PEs at the beginning and the scatter stage reads the edge list from global memory and works as producer, generating update tuples in parallel. Then it passes the tuples to the shuffle stage, which dispatches the tuples in parallel to their corresponding PEs in gather stage. The gather stage works as consumer, processing the update tuples generated by scatter stage. Taking advantage of implementing them in different OpenCL kernels, the stages run in a pipelined manner. After all the partitions are processed by the scatter and gather stages, the apply stage will further update the property of each vertex for the next iteration. The internal structure of this is discussed in detail in Section V. The shuffle stage is the most critical process for our proposed framework, as it connects the scatter and gather stages and deals with the data dependencies. We propose a decoder and filter based shuffling technique which

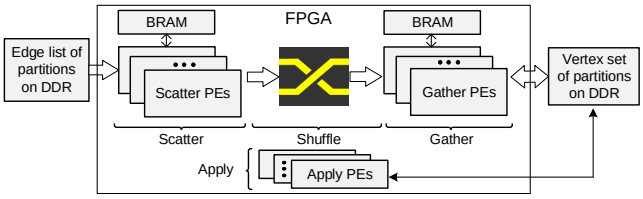


Fig. 3: System Overview.

provides a high parallelism, connects the scatter and gather stages efficiently, and satisfies the throughput requirement of the other parts. The details are described in Section V.

V. Implementation Details

We present the detailed design for all the stages of the proposed framework in this section. The architecture with functional components is shown in Fig. 4. Each individual component is connected by OpenCL channels which has the functionality as FIFOs.

A. Scatter

The inputs to the scatter stage are the edge lists from global memory, and the outputs are update tuples which are calculated by the specific functions of the algorithms. The scatter PE (sPE) is instantiated N times, to read multiple edges and generate N update tuples in parallel. We configure the suitable value for N so that the memory bandwidth is fully utilized.

B. Shuffle

The shuffle stage dispatches the unordered update tuples to corresponding gather PEs (gPEs). The shuffle stage is constructed by five modules, *PE Selection*, *Data Duplication*, *Validation*, *Decoder* and *Filter*. We define gPE together with validation, decoder, and filter modules for the same set of input data as a datapath, as shown in Fig. 4b.

1) PE Selection: The PE Selection unit determines which gPE the update tuple will be sent to, based on the destination of the update tuple. A radix-bit hash function is used for choosing corresponding PE. The id of the gPE (id_{gPE}) for the input tuple is decided by the least significant bits of the destination vertex identifier, $\log_2(pe\_number)$, where pe_number is the total number of gPEs. The update tuples will go to the gPE whose id_{gPE} is equal to $vid \pmod{(pe\_number)}$, where vid is the identifier of destination vertex of the update tuple. The execution of the radix-bit hash function costs only 1 clock cycle on FPGA. Also, id_{gPE} for each tuple is computed in parallel using N instances of the hash function and then is appended to the update tuple for the next stage.

2) Data Duplication: In order to balance the update tuple distribution, the Data Duplication module duplicates the input tuples and the attached hash value from PE Selection module to datapaths. This allows each Validation module to read data from its own copy and operate on the tuples independently. When a datapath has multiple valid tuples to read, data duplication allows other datapaths to start processing the next set of tuples, instead of waiting for the datapath to finish its processing.

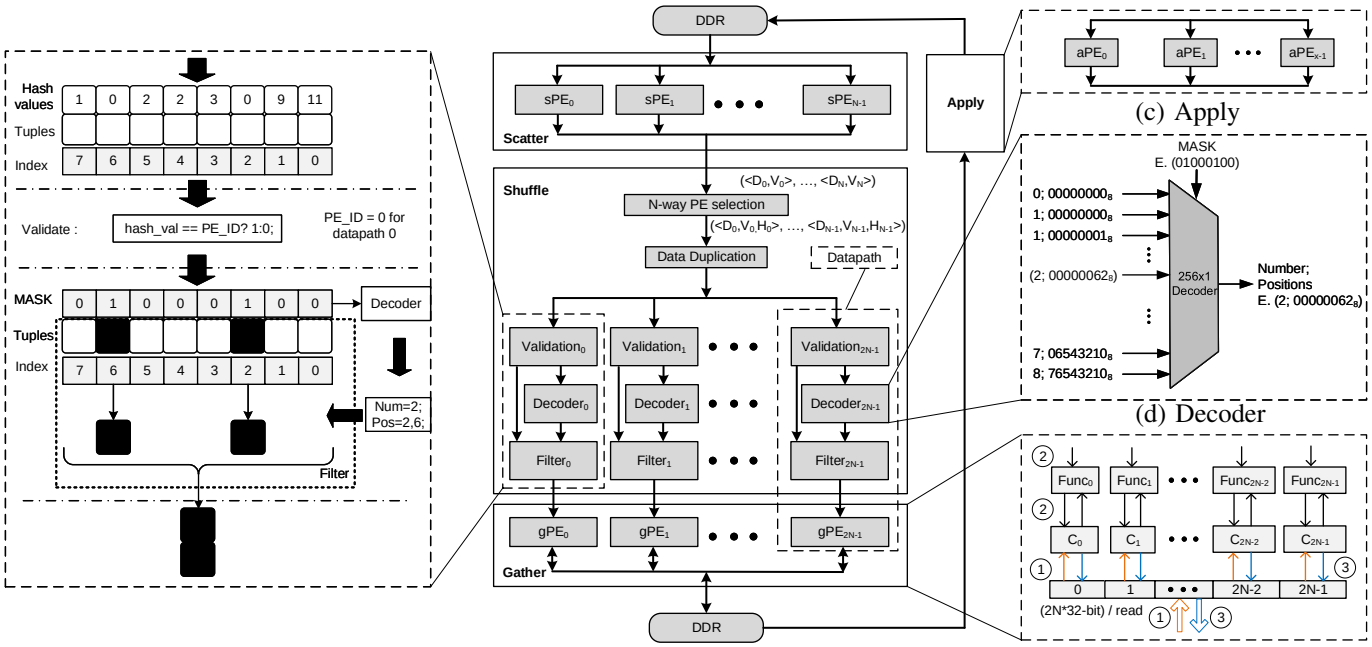


Fig. 4: Overall architecture with component details.

3) Validation: For the i^{th} datapath with gPE id id_{gPE_i}, a Validation module is instantiated to achieve this by comparing the hash value of incoming tuples with id_{gPE_i}. If the hash value of a tuple matches with id_{gPE_i}, the tuple is flagged as *valid* (1, *invalid* is 0 on the contrary). This is done in parallel for all the N input tuples using multiple instances of validation logic. The results are saved in a N-bit internal register, named as MASK, then attached with the tuples.

4) Decoder: The purpose of the Decoder is to determine the number of valid tuples and their positions based on the MASK value provided by the Validation module. We propose a multiplexer based solution as shown in Fig. 4d. Since there are N tuples read per cycle, a N-bit MASK code is generated from the Validation module for each datapath, resulting in 2^N combinations for the output. A 2^N entries multiplexer could select the desired output in a single clock cycle which takes the MASK value as the *select* input of the multiplexer. Position of each valid tuple is represented with a $\log_2 N$ bits number, and all of them are concatenated into a $N * \log_2 N$ bits wide number. In order to minimize the resource utilization of the multiplexer to support wider decoder and avoid the routing problem during the implementation on FPGA, we split the MASK value output from validation into multiple partitions and use multiple decoders with a smaller size. We use threshold N equivalent to 8 for splitting the decoder. An example of decoder is shown in Fig. 4d. The outputs of the decoder with the shown input are: number of valid tuples as 2 and the positions as 00000062_8.

5) Filter: Algorithm 1 shows the detailed operations of the filter with one decoder handling 8 tuples. The loop count is set as number of valid tuples. Given the positions, the filter fetches one valid tuple per cycle to the corresponding datapath, without wasting clock cycles on invalid tuples.

Algorithm 1 Filter Function

Require:
Input tuples, $tuples[8]$;
Number of valid tuples, num, (unsigned char);
Valid tuples positions, $positions$, (unsigned integer);
Ensure: Write all valid tuples to PE kernel;
(num, positions) = read channel (*decoder*);
tuples = read channel (*validation*);
for $i = 0$ to num **do**
    uchar *position* = (*positions* >> (*i* * 3)) & 0x7;
    uint2 *tuple* = *tuples*[*position*];
    write channel (*gPE*, *tuple*);
end for

6) Shuffle flow: All the modules of the shuffle stage are pipelined. An example for a set of 8 tuples is shown in Fig. 4a. The input to the datapath is a set of radix-bit hash values and update tuples. The Validation module computes an 8-bit MASK by comparing hash values with the id of the PE, which is 0. Using the MASK values, the Decoder module indicates that 2 tuples are valid for this datapath, and their positions are 2 and 6. Finally, the Filter module writes the valid tuples one-by-one to the gPE without “bubbles".

C. Gather

The gather stage processes the upcoming update tuples. It is constructed with M gPEs running in parallel and independently. Each of the gPE processes a specific set of vertices. The i^{th} gPE processes vertices for which $(vid \pmod U) \pmod{pe\_number}$ is equal to i, where vid is the vertex identifier and U is the total number of vertices the BRAM can cache. Fig. 4e shows the internal structure and dataflow of gPE. First, the vertex properties are loaded into the local cache (C_i) of gPEs. The gather function ($Func_i$)

TABLE I: Graph dataset.

Graphs	$\|V\|$	$\|E\|$	D_{avg}	D_{max}
rmat-19-32 (R19) [34]	524K	17M	32	90K
rmat-21-32 (R21) [34]	2M	67M	32	211K
mouse-gene (MG) [35]	43K	14.5M	670	8K
web-google (GG) [35]	875K	5.1M	11	6.4K
pokec (PK) [35]	1.6M	31M	37	20K
wiki-talk (WT) [35]	2M	5M	4	100K
live-journal (LJ) [35]	4.8M	69M	13	3K
twitter-2010 (TW) [35]	41M	1.4B	35	770K

block uses the value of update tuple to update vertex property in cache. After all the update tuples are processed, the updated vertices are stored back to global memory.

The function of the gPE depends on the algorithms. For each update tuple, the destination vertex property is first read then processed with the coming value, finally stored back. Then, next update tuple starts. Due to the read after write operation, the best achievable initiation interval (II) is 2 [17]. Thus, we need twice gPEs as many as sPEs ($M = 2N$), such that the scatter and gather stages have the same throughput. To achieve this, at the Data Duplication step of shuffle stage, tuples are duplicated $2N$ times, resulting in $2N$ datapaths (Validation, Decoder, Filter, and gPE modules).

D. Apply

The apply stage, as shown in Fig. 4c, updates all the vertex properties using multiple aPEs. The vertex properties are read and updated in parallel to fully utilize the memory bandwidth. The operations executed in apply stage depend on the specific graph processing algorithms.

VI. Evaluation

A. Experimental Setup

Our experiments are conducted using Intel FPGA SDK 16.1 for OpenCL, on a Terasic DE5-Net board (named DE5) which includes an Altera Stratix V GX FPGA and two independent banks of DDR3. All the presented results are collected from on-board implementations. The graph dataset used are described in Table I which contains both synthetic [34] as well as real-world graphs. We implement four common algorithms: Page Rank (PR), Sparse Matrix-vector Multiplication (SpMV), Single Source Shortest Path (SSSP), and Breadth-first Search (BFS) and use fixed-point data type for calculations. The number of million traversed edges per second (MTEPS) [14] is used as the evaluation metric.

B. Shuffle Efficiency

We first evaluate the efficiency of our proposed shuffle technique. In this experiment, the input of the shuffle is directly connected to the memory interface and the output is connected to different BRAM blocks independently. We use Knuth shuffle [36] to create the uniformly distributed update tuples with random destinations. We fix the frequency of the implementation in this evaluation to 200Mhz to approximate the impact when shuffle is used as an intermediate stage in the real applications. The full memory bandwidth is utilized to read the input data, which is a 512-bit data input per clock cycle thus it is 12.5GB/s for reads. The larger the tuple size is, the fewer tuples need to be shuffled per cycle. The theoretical throughput of number of tuples processed is calculated using Equation 1.

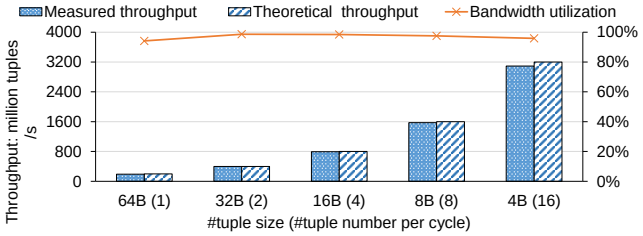


Fig. 5: Throughput and memory utilization of shuffle.

$$throughput = \frac{memory\_bandwidth}{tuple\_size} \quad (1)$$

We use a 2^8 entries multiplexer for the experiments with tuple number of 1, 2, 4, and 8, and use two 2^8 entries multiplexers to shuffle 16 tuples per cycle. As shown in Figure 5, the measured throughput of different tuple sizes is very close to the theoretical estimation. Our shuffle can efficiently dispatch unordered tuples to multiple datapaths regardless the data dependency.

C. Compare with Other Shuffling Solutions

We evaluate the solutions discussed in Section III with different tuple sizes, shown in Figure 6. In polling [32], only one tuple can be validated in each clock cycle. Hence, throughput is determined by the clock frequency. The FPPA [33] still retains the data dependency, thus, leads to a high II. Our experimentally collected II for the FPPA is 284, 79, 9 and 1 cycles for shuffling 8, 4, 2 and 1 tuples to the PEs, respectively. Our proposed solution resolves these issues efficiently. Therefore, the II of the shuffle stage remains at 1 for all tuple sizes.

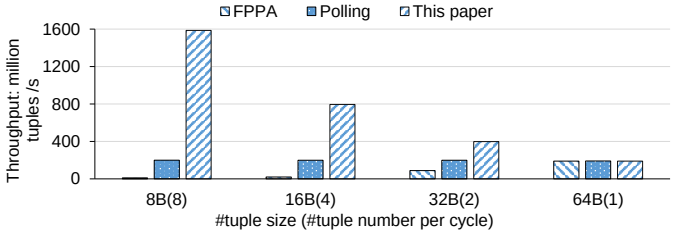


Fig. 6: Performance comparison with other shuffling solutions.

D. Benefit of Shuffling Technique

Taking advantage of our efficient shuffle technique, the run-time data dependency is handled efficiently and gPEs run in a parallel and independent manner. We calculate the performance for four algorithms on all graphs in the dataset with 1, 2, 4, 8, and 16 PEs and present the results for Live-journal graph in Figure 7. Similar trends were oberseved for all other graphs. As shown in Figure 7, the throughput improves significantly with increase in number of PEs for all the evaluated algorithms. There is almost 20× speedup with 16 PEs when compared to 1 PE. The additional speedup comes from the frequency improvement with multiple PEs by taking advantage of the smaller BRAM size for each of the PE that leads to a better timing for the FPGA implementation.

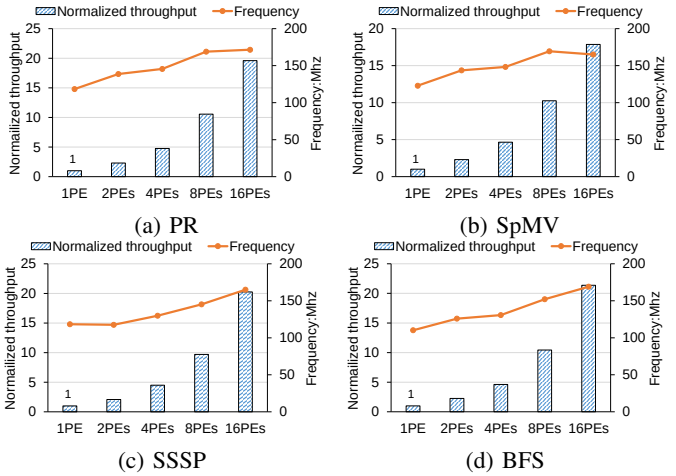


Fig. 7: Throughput with different number of PEs, normalized to throughput of 1 PE.

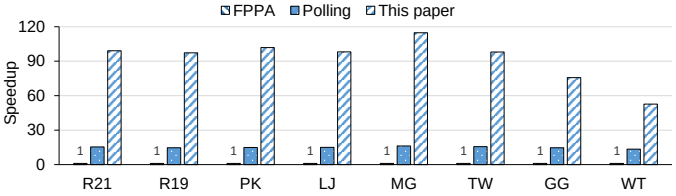


Fig. 8: Speedup of PageRank with 16 PEs on different shuffling solutions.

With the shuffling technique, the proposed graph processing framework demonstrates very good scalability as the number of PEs increases. We also compare the end-to-end performance of our proposed framework by using the FPPA and Polling instead of the proposed shuffling technique. We observe the same trend for all the four evaluated algorithms, and present the results for PageRank in Figure 8. The speedup is up to $100\times$ of FPPA, and $6\times$ of Polling.

E. Overall Evaluation

The throughput per iteration of four algorithms with our proposed framework, with 16 PEs, are shown in Table II. The resource utilization for each of them are presented in Table III. With 16 datapaths design, the shuffle stage uses 2.9% of total BRAMs and 15.4% of total ALUT. Since highly imbalanced graphs may influence the throughput of framework, we also study the performance on synthetic graphs (generated using Boost library [37]) with different power-law distribution factor, α. Each graph has 500K vertices and 16 million edges. The throughput for PageRank with the proposed framework by varying the value of α from 0 to 3, in increments of 0.5, is shown in Table IV. Based on results presented, we make the following observations. First, our designs achieve a high throughput of more than 1,000 MTEPS on average. Second, PR and SpMV cost more BRAM and DSP resources when compared to BFS and SSSP due to their more complex apply functions. Third, all the algorithms are implemented with well utilized BRAMs to provide on-chip data cache capacity for the vertex set. Fourth, our framework is robust to power-law distribution in graphs, i.e., it performs consistently well even for highly imbalanced graphs.

TABLE II: Throughput of different algorithms.

Algo.	R19	R21	MG	PK	GG	WT	LJ	TW
PR	1136	1109	1223	1165	839	584	1111	1064
SpMV	1066	1042	1143	1085	803	551	1052	1002
BFS	1229	1219	1271	1152	878	579	1090	914
SSSP	1252	1250	1288	1178	908	619	1129	932

TABLE III: Resource utilization and frequency.

Algo.	Freq.(Mhz)	BRAM	Logic	DSP
PR	171.5	2,329 (91%)	125,811 (54%)	8 (3%)
SpMV	165.4	2,394 (94%)	125,854 (54%)	8 (3%)
BFS	172.6	2,030 (79%)	127,085 (54%)	0 (0%)
SSSP	170.6	1,926 (75%)	123,457 (53%)	0 (0%)

F. Comparison with State-of-the-art Designs

As a sanity check, we choose the state-of-the-art graph processing frameworks implemented with RTL as our comparison targets. Since the edge-centric processing model is memory bounded, we compare with implementations which have similar memory bandwidth with our platform. We compare with state-of-the-art edge-centric designs for BFS and SSSP [24]. The proposed OpenCL based designs have throughput improvement of up to $2.9\times$ compared to state-of-the-art designs, demonstrating the efficiency of our proposed techniques, as shown in Table V.

VII. Conclusion

In this paper, we have identified the run-time data dependency introduced by dispatching data to multiple PEs as the main obstacle to achieve high throughput on graph processing problems using OpenCL-based FPGAs. We propose a novel OpenCL-based data shuffling method to handle this data dependency efficiently. With the proposed data shuffling technique, we also present an edge-centric graph processing framework with high pipeline execution efficiency. The experimental studies show that our approach achieves throughput that are comparable or even better than RTL-based design in the previous studies.

VIII. Acknowledgement

This work is supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore. This work is also partly supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme, and the SenseTime Young Scholars Research Fund. We also thank Intel for hardware accesses and donations.

TABLE IV: Throughput of PageRank.

α	0	0.5	1	1.5	2	2.5	3
Throughput	1224	1226	1219	1221	1242	1238	1237

TABLE V: Comparison with state-of-the-art implementations.

Algo.	Graph	Others	Throughput	Ours	Impro.
PR	LJ	ForeGraph [14]	1193	1110	0.93×
	WT	[11]	279	584	2.09×
	GG	[11]	317	838	2.64×
SpMV	WT	GraphOps [38]	190	551	2.90×
SSSP	WT	[13]	657	618	0.94×
	LJ	[13]	872	1129	1.29×

REFERENCES

[1] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *PVLDB*, 2012.

[2] Graph 500. [Online]. Available: https://graph500.org/

[3] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, 2007.

[4] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Commun. ACM*, 2016.

[5] I. Kuon, R. Tessier, J. Rose *et al.*, "FPGA architecture: Survey and challenges," *Foundations and Trends® in Electronic Design Automation*, 2008.

[6] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph processing framework on FPGA a case study of breadth-first search," in *FPGA*, 2016.

[7] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen, "High level synthesis of complex applications: An h.264 video decoder," in *FPGA, 2016*.

[8] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs," in *FPGA, 2019*.

[9] Y. Chen, S. T. Gurumani, Y. Liang, G. Li, D. Guo, K. Rupnow, and D. Chen, "FCUDA-NoC: A scalable and efficient network-on-chip implementation for the CUDA-to-FPGA flow," *VLSI*, 2016.

[10] Y. Chen, T. Nguyen, Y. Chen, S. T. Gurumani, Y. Liang, K. Rupnow, J. Cong, W. Hwu, and D. Chen, "FCUDA-HB: Hierarchical and scalable bus architecture generation on fpgas with the FCUDA flow," *TCAD*, 2016.

[11] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing memory performance for FPGA implementation of pagerank." in *ReConFig*, 2015.

[12] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An FPGA framework for edge-centric graph processing," in *CF*, 2018.

[13] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on FPGA," in *FCCM*, 2016.

[14] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-FPGA architecture," in *FPGA*, 2017.

[15] S. Huang, M. El-Hadedy, C. Hao, Q. Li, V. S. Mailthody, K. Date, J. Xiong, D. Chen, R. Nagi, and W.-m. Hwu, "Triangle counting and truss decomposition using FPGA," in *HPEC, 2018*.

[16] Z. Li, L. Liu, Y. Deng, S. Yin, Y. Wang, and S. Wei, "Aggressive pipelining of irregular applications on reconfigurable hardware," in *ISCA*, 2017.

[17] Intel. Intel FPGA SDK for opencl pro edition programming guide. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html

[18] Xilinx. Vivado high-level synthesis. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[19] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *FPGA*, 2011.

[20] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *TCAD*, 2016.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs." in *OSDI*, 2012.

[22] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *SOSP*, 2015.

[23] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An FPGA framework for vertex-centric graph computation," in *FCCM, 2014*.

[24] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *SOSP*, 2013.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *SIGMOD, 2010*.

[26] S. Zhou and V. K. Prasanna, "Accelerating graph analytics on CPU-FPGA heterogeneous platform," in *SBAC-PAD*, 2017.

[27] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," in *PACT*, 2018.

[28] N. Kapre and J. Gray, "Hoplite: A deflection-routed directional torus noc for fpgas," *TRETS, 2017*.

[29] N. Kapre, "Custom fpga-based soft-processors for sparse graph acceleration," in *ASAP, 2015*.

[30] M. Delorimier, N. Kapre, N. Mehta, and A. Dehon, "Spatial hardware implementation for sparse graph algorithms in graphstep," *TAAS, 2011*.

[31] N. Kapre and H. Patel, "Applying models of computation to opencl pipes for fpga computing," in *IWOCL*, 2017.

[32] B. K. Bergen, M. G. Daniels, and P. M. Weber, "A hybrid programming model for compressible gas dynamics using opencl," in *ICPP, 2010*.

[33] Z. Wang, J. Paul, B. He, and W. Zhang, "Multikernel data partitioning with channel on opencl-based FPGAs," *TVLSI*, 2017.

[34] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *JMLR*, 2010.

[35] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[36] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *ICDE*, 2013.

[37] Boost C++ Libraries. [Online]. Available: http://www.boost.org/

[38] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *FPGA*, 2016.