IP Cores for Graph Kernels on FPGAs

Sanmukh R. Kuppannagari, Rachit Rajat, Rajgopal Kannan
Ming Hsieh Department of Electrical and Computer Engineering
University of Southern California
Los Angeles, California 90089
Email: {kuppanna, rrajat, rajgopak}@usc.edu

Aravind Dasu
Programmable Solutions Group
Intel Corporation
San Jose, California
Email: aravind.dasu@intel.com

Viktor K. Prasanna
Ming Hsieh Department of Electrical and Computer Engineering
University of Southern California
Los Angeles, California 90089
Email: prasanna@usc.edu

***Abstract*—Graphs are a powerful abstraction for representing networked data in many real-world applications. The need for performing large scale graph analytics has led to widespread adoption of dedicated hardware accelerators such as FPGA for this purpose. In this work, we develop IP cores for several key graph kernels. Our IP cores use graph processing over partitions (GPOP) programming paradigm to perform computations over graph partitions. Partitioning the input graph into non-overlapping partitions improves on-chip data reuse. Additional optimizations to exploit intra- and inter- partition parallelism and to reduce external memory accesses are also discussed. We generate FPGA designs for general graph algorithms with various vertex attributes and update propagation functions, such as Sparse Matrix Vector Multiplication (SpMV), PageRank (PR), Single Source Shortest Path (SSSP), and Weakly Connected Component (WCC). We target a platform consisting of large external DDR4 memory to store the graph data and Intel Stratix FPGA to accelerate the processing. Experimental results show that our accelerators sustain a high throughput of up to 2250, 2300, 3378, and 2178 Million Traversed Edges Per Second (MTEPS) for SpMV, PR, SSSP and WCC, respectively. Compared with several highly-optimized multi-core designs, our FPGA framework achieves up to 20.5× speedup for SpMV, 16.4× speedup for PR, 3.5× speedup for SSSP, and 35.1× speedup for WCC, and compared with two state-of-the-art FPGA frameworks, our designs demonstrate up to 5.3× speedup for SpMV, 1.64× speedup for PR, and 1.8× speedup for WCC, respectively. We develop a performance model for our GPOP paradigm. We then perform performance predictions of our designs assuming the graph is stored in HBM2 instead of DRAM. We further discuss extensions to our optimizations to improve the throughput.**

I. Introduction

Graphs are a power abstraction for representing real-world networked data in multiple scientific and engineering domains such as social networks, network biology, genome analysis, etc. [1]. Many graph processing frameworks targeting general purpose processors which provide high-level programming models for users to easily perform graph processing have been developed [1], [2], [3], [4], [5], [6], [7], [8], [9]. However, general purpose processors are not the ideal platform for graph processing due to their inefficient memory access granularity (cache line level granularity), and ineffective on-chip memory usage due to the poor spatial and temporal locality of graph algorithms [10], [11].

Therefore, dedicated hardware accelerators for graph processing have gained popularity recently [12], [11], [10], [13], [14], [15], [16], [17], [18]. With the increased interest in energy-efficient acceleration, Field-Programmable Gate Array (FPGA) has become an attractive platform to develop accelerators [19], [20]. State-of-the-art FPGA devices, such as Intel Stratix 10 MX [21], provide dense logic elements (702,720 Adaptive Logic Modules (ALMs)), abundant on-chip memory resources (2,810,880 ALM registers, 145 Mb of M20K) and 8 GB (2 stacks of 4 GB) of High Bandwidth Memory 2 (HBM2) that can provide a memory bandwidth which is as high as 512 GBps. Existing FPGA frameworks [22], [14], [13] for general graph algorithms are design based on the vertex-centric paradigm which access the edges of vertices through pointers or through vertex indices. This can result in massive random access to memory (HBM2 or external), thereby incurring significant communication latency and accelerator stalls [23].

In this work, we develop FPGA IP cores for several key graph kernels. We use our Graph Processing Over Partitions (GPOP) paradigm [24] to implement FPGA designs. GPOP performs graph processing over partitions of the graph to enhance on-chip data reuse and make effective use of the external memory bandwidth. We develop FPGA IP cores for four key graph kernels: Sparse Matrix Vector Multiplication (SpMV), PageRank (PR), Single Source Shortest Path (SSSP), and Weakly Connected Component (WCC). We also develop a performance model to estimate the performance improvements that can be obtained by using the high bandwidth HBM2 memory instead of the DDR4 DRAM as the memory to store the graph and discuss extensions to our optimizations which can further improve the throughput.

The contributions of this work are as follows:

- We develop IP cores for four key graph kernels: SpMV, PR, SSSP, and WCC using our GPOP paradigm [24] to perform processing over graph partitions.
- We implement the IP cores on Intex Stratix 10 MX 2100 FPGA and achieve upto 35× performance improvement over optimized state-of-the-art multi-core based designs and 5.3× over optimized state-of-the-art FPGA based designs.
- We perform detailed performance modeling and prediction of using HBM2 instead of DDR4 to store the graph.

II. Background

We first briefly describe the four graph algorithms that we consider in this work and that are core kernels and building blocks in many different application. We then describe our GPOP paradigm to develop IP cores for the kernels.

A. Graph Kernels

1) Sparse Matrix-Vector Multiplication: Generalized Sparse matrix-vector multiplication (SpMV) iteratively computes $x^{t+1} = Ax^t$, where A is a sparse $H \times I$ matrix with row vectors A_i, x is a dense vector of size I, and t denotes the number of iterations that have been completed. To map SpMV into our processing paradigm, each non-zero entry of A is represented as a weighted edge, and each element of x is represented as a vertex.

2) Page Rank: PageRank (PR) is used to rank the importance of vertices in a graph [25]. It computes a weight for each vertex of the graph based on the weights on the vertices on the incoming edges. In other words, it calculates the likelihood that starting from a random vertex in the graph, we will reach this vertex. The algorithms starts with assigning same initial value to each vertex. Then, in each iteration, the PageRank value of each vertex v is updated based on Equation 1. Here d is a constant called damping factor, $|V|$ is the total number of vertices of the graph; v_i represents the neighbor of v such that v has an incoming edge from v_i; L_i is the number of outgoing edges of v_i.

$$PageRank(v) = \frac{1-d}{|V|} + d \times \sum \frac{PageRank(v_i)}{L_i} \quad (1)$$

3) Single Source Shortest Path: Single Source Shortest Path (SSSP) finds the shortest paths from a single source vertex to all the other vertices in a weighted graph. The algorithm proceeds as follows: the vertex attribute denotes the weight of the shortest path from the source vertex to itself in each iteration. It is initialized to ∞. We say that a vertex is active if its attribute was updated in the previous iteration. In the scatter phase of each iteration, all the active vertices send their updates to their neighbors using the outgoing edges. In the gather phase, each vertex that receives updates from neighbors updates its attribute if a shorter path to the source vertex is found. The algorithm terminates when none of the vertices are active.

4) Weakly Connected Component: A Weakly Connected Component (WCC) is a maximal subgraph in which there is a path between any two vertices. The problem of WCC is to find all such subgraphs. The algorithm runs as follows,: each vertex maintains a Connected Component (CC) identifier as the attribute to record the connected component it belongs to. The CC identifier of a connected component is the vertex with the smallest index. In the scatter phase, each active vertex sends its CC identifier to its neighbors. In the gather phase, a vertex updates its CC identifier to be the minimum of its current value and the ones its received. On termination, the vertices with same attributes form connected components.

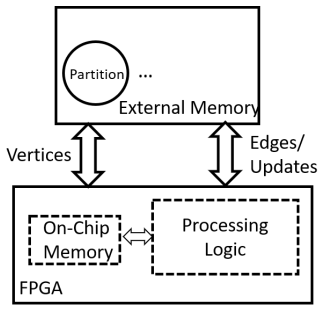

Fig. 1. Graph Processing Over Partitions

B. Graph Processing Over Partitions (GPOP) Paradigm

In graph processing over partitions (GPOP) paradigm [24], the computations are performed with partition as the center of computation as opposed to vertex or edge in the conventional techniques. We assume that the entire graph can fit into the external memory. The graph is partitioned such that each partition can fit into the on-chip memory. Each partition is fetched and one iteration of scatter-gather model of computation is performed as follows:

1) **Scatter:** In this phase, all the edges of the partition are processed to generate updates to the vertices.
2) **Gather:** In this phase, the updates generated in the scatter phase are applied to the vertices.

Specifically, the following steps are performed in each iteration of scatter-gather model:

- *Scatter:*
  - **C1:** Fetch partitions from the external memory into the on-chip memory.
  - **C2:** Stream the edges corresponding to the partitions from the external memory onto the device.
  - **E1:** For each edge, access its source vertex and produce an update. Update consists of a destination vertex and the value of update to be applied.
  - **C3:** Store the updates produced back into the external memory.
- *Gather:*
  - **C4:** Fetch partitions from the external memory.
  - **C5:** Stream the updates corresponding to the partitions back into the device.
  - **E2:** For each update, apply it to the destination vertex.
  - **C6:** Write the partitions (which consist of updated vertices) back into the external memory.

Figure 1 shows a high level overview of the processing paradigm. The graph partitions, consisting of vertices, are fetched one by one onto the on-chip memory. After a partition is fetched, the edges are fetched into the FPGA in a streaming manner and processed. The updates produced are written back to the external memory. Then, the updates are fetched back into the FPGA in a streaming manner and the corresponding vertices residing in the on-chip memory are updated. Finally, the partition is written back into the external memory.

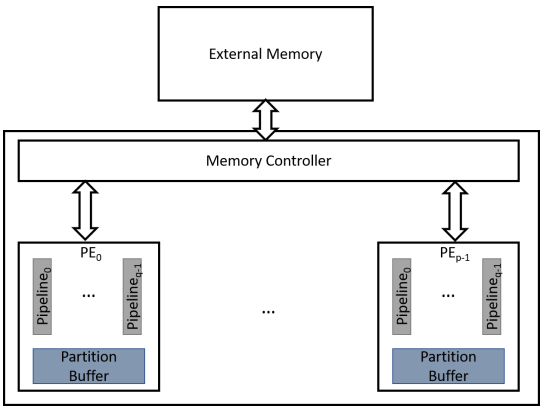

Fig. 2. Architecture Design

III. IP Cores for Graph Processing

A. Architecture Design

Figure 2 shows a high level architecture of the FPGA design. The external memory stores all the graph partitions. There are p processing engines (PEs) on the FPGA, which can be customized based on the target graph algorithm. These PEs process p distinct partitions in parallel. Each PE has an individual partition buffer constructed by on-chip memory and used to store the partition being processed by the PE. Each PE has q pipelines. In the scatter phase, the PEs read edges from the external memory and write back the updates produced into the external memory. In the gather phase, the PEs read updates from the external memory and write back the updated vertices.

B. IP Core Implementation

Given an algorithm in graph processing over partitions paradigm, our IP core implementation is essentially a mapping of the algorithm to the target FPGA architecture by customizing the parallel pipelines. We assume PEs to be homogeneous and capable of performing both the scatter and gather phase. The PEs are customized to implement the edge processing (scatter phase) and vertex attribute update (gather phase) pertinent to the algorithm being implemented.

C. Optimizations

1) Graph Partitioning: In each iteration, vertex data (attribute) need to be accessed repeatedly. To offer fine-grained single-cycle accesses to the data, we buffer the vertices in the on-chip RAMs. For large graphs, whose vertex array cannot fit in the on-chip RAMs, the graph is partitioned using a simple vertex-index-based partitioning approach [26]. As per this approach, assuming the partition buffers of a PE can store the data of m vertices, the input graph is partitioned into $k = \lceil \frac{|V|}{m} \rceil$ partitions, where $|V|$ denotes the total number of vertices in the graph. Thus, any at given time p partitions reside in the on-chip memory and are processed using the PEs in parallel.

2) Parallelization: **Inter-partition Parallelism:** Assuming the processing logic consists of p ($p \geq 1$) Processing Engines (PEs), we can independently process p partitions in parallel. The inter-partition parallelism of the design is denoted as p. When a PE completes the processing of a partition, it is automatically assigned another partition to process. **Intra-partition Parallelism:** Each processing engine employs parallel pipelines to concurrently process distinct edge (updates) during the scatter (gather) phase. Assuming each processing engine has q ($q \geq 1$) parallel pipelines, q distinct edges (updates) of the same partition can be concurrently processed by the processing engine during the scatter (gather) phase per clock cycle. The intra-partition parallelism of the design is denoted as q.

3) Update Combination: In the scatter phase, the total number of produced updates, which need to be written into the external memory, in the worst case is equal to the number of edges $|E|$. In order to reduce the data communication for writing the updates to the external memory, we combine the updates that have the same destination vertex. To enable this, the edges of each partition are sorted based on the destination vertices. Due to this optimization, in the scatter phase, the updates with same destination vertex are produced consecutively. We then combine the consecutive updates that have the same destination vertex as a single update. This also reduces the number of updates to be processed in the gather phase, thereby reducing the data communication of the gather phase as well.

IV. IP Core Evaluation

A. Experimental Setup

We implemented the graph kernel IP cores using our GPOP paradigm (Section II-B) on Stratix 10 MX 2100 FPGA. This device has 702,720 Adaptive Logic Modules (ALMs), 2,810,880 ALM registers, 145 Mb of M20K. Four DDR4 chips are used as external DRAM, each chip having a bandwidth of 25 GB/s. Post-place-and-route simulations were performed using Inter Quartus Prime Pro 18.1. Experiments were conducted on the real life graphs mentioned in Table I.

TABLE I
Real life graph datasets

Dataset	# Edges $\|\|E\|\|$	#Vertices $\|\|V\|\|$	Description
WK [27]	5M	2.4M	Wikipedia Network
LJ [28]	69M	4.8M	Social Network
TW [29]	1468.4M	41.6M	Twitter Network
CA	5.5M	2.0M	Road Network
RMat24	263.0M	16.8M	Synthetic Network

B. Performance Metrics

The following metrics were used for evaluation:

- Resource utilization: Percentage of FPGA resources utilized by the design. Resources include M20K utilization, logic utilization and DSP utilization.
- Power consumption (Watt): Power consumption of the design.
- Execution time (ms): Total execution time required for each iteration.
- Throughput (MTEPS): Number (in Millions) of edges traversed per second.

C. Resource Utilization and Power Consumption

To saturate the external memory bandwidth the number of PEs was set to 4 (p = 4), the number of pipelines in each PE was set to 8 (q = 8), the interval size to 128K (m = 128K). The clock speed of 200 MHz, 185 MHz, 183 MHz and 140 MHz was achieved for SpMV, PR, WCC and SSSP respectively. The junction temperature was set to 50C. Table II shows the resource utilization (in%) and power consumption (Watt).

TABLE II
RESOURCE UTILIZATION

Algorithm	ALM	Register	DSP	M20K	Power
SpMV	23	26	5	41	12.7
PR	19	20	5	42	12.4
WCC	20	19	0	39	12.2
SSSP	20	18	0	38	12.5

D. Throughput and Execution Time

Table III mentions the throughput and execution time observed in our experiments. High throughput of 2250 MTEPS, 2300 MTEPS, 3106 MTEPS and 2178 MTEPS was observed for SpMV, PR, WCC and SSSP respectively.

E. Effect of Update Combination

The effectiveness of update combining and filtering optimization to reduce the data communication between external memory and FPGA is shown by comparison with a baseline design which has the partition and data layout optimization but does not have the update combining and filtering optimization. The number of write updates are reduced by a factor of up to 11X, 12X, 18X and 22X for SpMV, PR, SSSP and WCC respectively.

F. Comparison with state-of-the-art designs

1) Comparison with multi-core designs: We compare our design against multiple state of the art multi-core designs such as X-Stream [3], GraphMat [4], NXgraph [5], GraphX [30] and PowerGraph [6]. 32-core AMD Opteron 6272 processor with 25 GB/s DRAM bandwidth was used in X-Stream [3]. A system with hexa-core Intel i7 processor with 160 GB/s DRAM bandwidth was used in NXgraph [5]. A cluster with 16 computing nodes where each node has 8 cores was used in GraphX [30]. A system with 24-core Intel Xeon E5-2697 processor with 80 GB/s DRAM bandwidth was used in GraphMat [4]. PowerGraph [6] uses 64 node cluster of Amazon EC2 where each instance has two quad core Intel Xeon X5570 processor with 23GB RAM. The comparison results are summarized in Table IV. Our FPGA based design compared to state-of-the-art multi-core based designs achieve up to 3.5X, 16.4X, 20.5X and 35.1X improvement for SSSP, PR, SpMV and WCC respectively. The power consumption of multi-core designs (usually $>$ 80 Watt) is also much more that the power consumption of our FPGA based design ($<$ 20 Watt). Hence, from energy efficiency point of view our design achieves even larger improvements.

TABLE III
THROUGHPUT AND EXECUTION TIME

Algorithms	Dataset	T_{exec} per iteration (ms)	Throughput (MTEPS)
SpMV	WK	5.0	1004
	LJ	36.2	1906
	TW	652.5	2250
	CA	2.8	1964
	RMat24	143.5	1832
PR	WK	4.9	1032
	LJ	35.4	1951
	TW	638.3	2300
	CA	2.9	1885
	RMat24	151.7	1735
WCC	WK	50.5	1523
	LJ	451.3	3039
	TW	7231.8	3106
	CA	1600.1	3378
	RMat24	1196.8	2422
SSSP	WK	36.4	1509
	LJ	845.9	2178
	TW	7966.9	2008
	CA	1590.4	1708
	RMat24	1316.1	1693

TABLE IV
COMPARISON WITH STATE OF THE ART MULTI-CORE DESIGN

Algorithms	Dataset	Approach	Throughput (MTEPS)	Improvement
SpMV	LJ	[3]	93	1.0X
		This Paper	1906	20.5X
PR	LJ	[3]	119	1.0X
		[4]	1530	12.9X
		This Paper	1951	16.4X
	TW	[6]	408	1.0X
		[5]	716	1.8X
		[4]	815	2.0X
		This Paper	2300	5.8X
WCC	LJ	[3]	187.6	1.0X
		This Paper	3039	16.2X
	TW	[30]	88.5	1.0X
		This Paper	3106	35.1X
SSSP	CA	[4]	488	1.0X
		This Paper	1708	3.5X
	RMat24	[4]	1151	1.0X
		This Paper	1693	1.47X

TABLE V
COMPARISON WITH STATE-OF-THE-ART FPGA BASED DESIGNS

Algorithms	Dataset	Approach	Throughput (MTEPS)	Improvement
SpMV	WK	[14]	190	5.3X
		This Paper	1004	
PR	WK	[12]	965	1.07X
		This Paper	1031	
	LJ	[12]	1193	1.64X
		This Paper	1951	
	TW	[12]	1856	1.24X
		This Paper	2300	
WCC	TW	[12]	1727	1.8X
		This Paper	3106	

2) Comparison with FPGA design: We compare our design against two state-of-the-art FPGA frameworks, GraphOps [14] and ForeGraph [12]. GraphOps is a hardware library

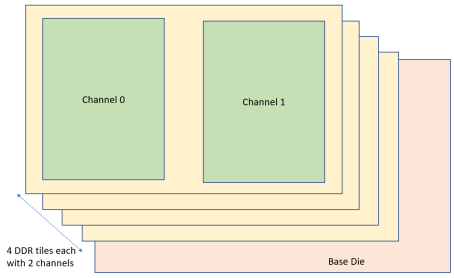

Fig. 3. A single stack of HBM2

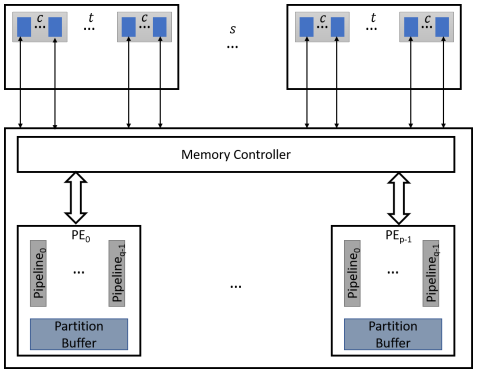

Fig. 4. FPGA connected to HBM2 with s stacks, t tiles/stack and c channels/tile. Total (64 bit wide) channels $s \times t \times c$.

for creating FPGA accelerator for graph analytics targeting Xilinx Virtex-6 (FPGA) and Intel Xeon X5650 (CPU) based heterogeneous architecture. ForeGraph is a graph processing framework which uses multiple interconnected FPGAs. Our design achieves upto 5.3X, 1.64X and 1.8X for SpMV, PR, WCC, respectively.

V. Modeling Performance Benefits of Using HBM2 for IP Cores

In this section, we develop a performance model for GPOP on FPGA connected to an external memory. We also model the HBM2 memory. We then use our performance model to estimate the benefits that can be achieved by using HBM2 to store the graph instead of DDR (Figure 4).

A. Architecture Parameters

We assume word granularity for processing and storage (64 bits in our implementations) and define the following architecture parameters: p: Number of Processing Elements (PE), q: Number of parallel pipelines in each PE, p_l: partition latency i.e. latency to read first vertex of partition plus latency of processing first edge (fetching plus exiting the pipeline) B: Size (words) of local on-chip memory, H_b: Bandwidth (words/cycle) of external memory, t: time period of one cycle, F_s: reduction factor of bandwidth for sequential update, F_r: reduction factor of bandwidth for random update.

B. Algorithmic Parameters

We define the following algorithmic parameters: P_n: number of partitions of the graph, P_s: size of each partition (words), I: number of iterations of scatter gather.

C. HBM2 Model

HBM2 is a 3D stacked DRAM with much higher performance compared to DRAM memory. We assume that there are s stacks of HBM2 where each stack has t tiles (DRAM Die). Each tile, supports c channels or $pc = 2 * c$ pseudo-channels. Each pseudo channel has a width of a word - 64 bit. With $t = 4$ tiles, $c = 2$ channels, and data rates of 2 Gbps per signal, the total bandwidth that can be achieved from a stack is 256GBps. An FPGA can support upto $s = 2$ stacks thus providing an aggregate bandwidth of 512 GBps. For random accesses, a bandwidth of around 50% of the peak can be sustained [31]. A high level overview of a single stack of the HBM2 memory is shown in Figure 3. Additionally, we assume the read/write latency to be 20 ns (worst case: computed using Table 64 in [32]).

Data Layout: The vertices and edges are accessed sequentially. Therefore, an optimal data layout will simply distribute the data across s stacks and across the t tiles of each stack such that in each cycle c words can be accessed using the available c channels from each tile.

D. Performance Estimation

We estimate the benefits of replacing one or four (used in our implementation) DDR4 chips with one or two HBM2 (a chip can support two HBM2 - Section V-C) as the external memory using our performance model. One DDR4 has a maximum bandwidth of 21.32 GBps while one HBM2 has a maximum bandwidth of 256 GBps. Assuming each edge takes 8 bytes, the maximum bandwidth values of these devices in terms of number of edges comes out to be 2.665 and 32 Giga edges, respectively. We assume that the entire graph can fit into the external memory. We also assume that both vertices and edges are laid out optimally in the external memory to achieve the peak bandwidth via preprocessing. Further, we assume random access is needed for writing the updates. We also assume that update combination reduces the write update messages by $\rho = 0.2$ (20%) (Section IV-E). Thus, The total time required by our algorithm to perform each iteration of scatter-gather is given as:

$$\frac{F_s|V|}{H_b} + \frac{F_r|V|}{H_b} + 2tp_l\frac{P_n}{p} + \max\{\frac{(2-2\rho)F_r|E|}{H_b} + \frac{F_s|E|}{H_b}, \frac{t|E|}{pq}\} + \max\{\frac{(1-\rho)F_s|E|}{H_b}, \frac{t|E|}{pq}\}$$

Here $\frac{F_s|V|}{H_b} + \frac{F_r|V|}{H_b}$ corresponds to reading/writing the partitions assuming sequential read and random writes, $2tp_l\frac{P_n}{p}$ is the sum of partition latencies incurred at scatter and gather phase, $\max\{\frac{(2-2\rho)F_r|E|}{H_b} + \frac{F_s|E|}{H_b}, \frac{t|E|}{pq}\}$ corresponds to computation/communication time in scatter phase with sequential edge reads and random optimized update combination

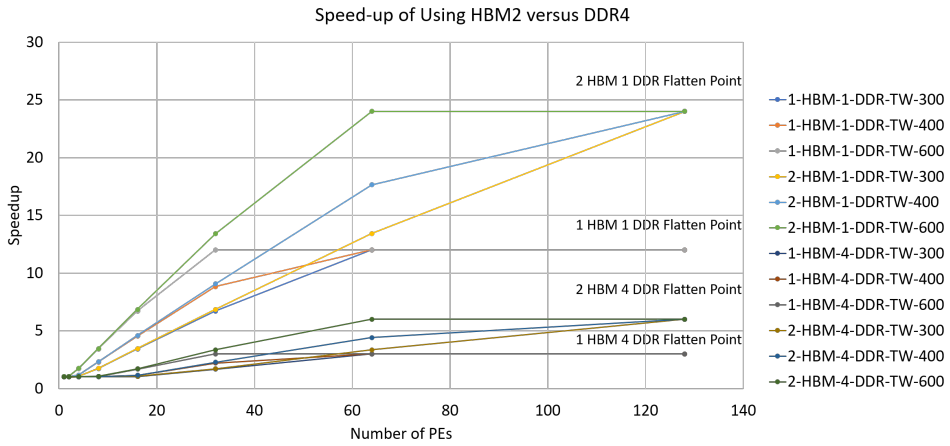

Fig. 5. Speedup obtained by using HBM2 to store graph instead of DDR

writes and $\max\{\frac{(1-\rho)F_s|E|}{H_b}, \frac{t|E|}{pq}\}$ corresponds to computation/communication time in gather phase. Note that F_s is always used for reads as we assumed sequential reads and F_r is always used with writes as we assumes the writes to be random. The values of F_s and F_r are around 1.1 and 2 respectively, signifying that in sequential reads the effective bandwidth is usually 90% while is random writes the effective bandwidth is usually 50%.

We vary the frequency of FPGA and determine minimum number of parallel pipelines ($p \times q$) required to saturate the bandwidth (Table VI). Thus, performance speedup can be obtained if the designs running at 300 MHz (600 MHz) with 1 or 4 DDR4 have more than 8 (4) or 33 (17) PEs, respectively.

TABLE VI
MINIMUM # OF PIPELINES TO SATURATE MEMORY BANDWIDTH

Frequency	DDR4	4 DDR4s	HBM2	2 HBM2
300 MHz	8	33	95	193
600 MHz	4	17	48	96

Figure 5 shows the speedup (ratio of Equation V-D) one can achieve by replacing 1 or 4 DDR4s with 1 or 2 HBM2 for varying number of pq for LJ graph dataset and varying the device frequency. We observe four performance speed up flattening points: 2 HBM replacing 1 DDR (2 HBM - 1 DDR): $24.01\times$, 1 HBM - 1 DDR: $12.01\times$, 2 HBM - 4 DDR: $6.0\times$, and 1 HBM - 4 DDR: $3.0\times$. These flattening points are essentially the ratio of the bandwidths of HBM and DDR.

E. Improving Performance with HBM2

Our IP cores utilize around 25% logic and 50% on-chip memory resources to implement 48 parallel pipelines at 200 MHz (Section IV-C). Our implementations were customized to fully utilize the bandwidth provided by 4 DDRs. Hence, it is to be expected that as per our performance model, no speed up will be obtained by replacing 4 DDRs with HBM2. However, if increasing design frequency to 300 MHz will lead to a speedup of $1.67\times$ (against a design running at 300 MHz with 4 DDR4s). Similarly, if our current IPs used only 1 DDR (instead of 4), at 300 MHz frequency, a speedup of $6.71\times$ is expected.

Increasing the number of pipelines will increase the speedup further. As evident from Table VI, 48-95 pipelines are needed to fully saturate the HBM2 bandwidth. The bottleneck in increasing the number of pipelines is the on-chip memory utilization. This can be addressed by reducing the size of partitions, thereby, increasing their number P_n by some constant α. This will lead to an increase of $2cq_l(\alpha-1)\frac{P_n}{p}$ in processing time while simultaneously reducing the processing time by $\frac{(\alpha-1)2c|E|}{\alpha pq}$. Setting $\alpha = 2$ to reduce the on-chip memory utilization to 25%, this will not increase the overall execution time as long as $P_n = \frac{p|V|}{B} \leq \frac{|E|}{2q_l q}$. For $pq = 32$, partition latency $q_l = 15$ (5 pipeline states, 10 fpga cycles for HBM2 read/write access latency), $|V| \leq \frac{|E|}{2}$ (as per datasets), $B = 2M$ vertices, the execution time will not increase. Thus, assuming linear scaling, we can obtain 193 (48×4) pipelines which will allow us to saturate bandwidth of 2 HBM2 modules and achieve upto $24.01\times$ speedup.

In conclusion, HBM2 will lead to increased speedup with as low as 8 pipelines running at 300 MHz or 4 pipelines at 600 MHz. For 2 HBM2 modules, the speedup will flatten after 193 pipelines running at 300 MHz or 96 pipelines at 600 MHz.

VI. ACKNOWLEDGEMENTS

This work was supported in part by Intel Strategic Research Alliance and in part by National Science Foundation under award number 1911229.

VII. CONCLUSION

In this paper we implemented 4 key graph kernels using GPOP paradigm on Stratix 10 MX FPGA device. Graph partitioning was used to increase the parallelism and to allow efficient on-chip buffering. Data communication was reduced using an efficient update method. Compared to state-of-the-art multi-core designs a speedup of $20.5\times$, $16.4\times$ and $35.1\times$ was observed while compared with state-of-the-art FPGA designs a speedup of $5.3\times$, $1.64\times$ and $1.8\times$ was observed for SpMV, PR and WCC respectively. We also developed a performance model and estimated the performance benefits our designs can achieve by using HBM2 as external memory.

REFERENCES

[1] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[2] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a {PC}," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 31–46.

[3] 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*. ACM, 2013, pp. 472–488.

[4] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[5] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "Nxgraph: An efficient graph processing system on a single machine," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 409–420.

[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)*, 2012, pp. 17–30.

[7] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on gpus," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 463–474.

[8] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *ACM SIGPLAN Notices*, vol. 51, no. 8. ACM, 2016, p. 11.

[9] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a gpu," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 233–245.

[10] 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)*. IEEE, 2016, pp. 103–110.

[11] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[12] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 217–226.

[13] 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*, 2016, pp. 105–110.

[14] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 111–117.

[15] J. Zhang, S. Khoram, and J. Li, "Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 207–216.

[16] H. Giefers, P. Staar, and R. Polig, "Energy-efficient stochastic matrix function estimator for graph analytics on fpga," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–9.

[17] J. Zhang and J. Li, "Degree-aware hybrid graph traversal on fpga-hmc platform," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 229–238.

[18] S. Khoram, J. Zhang, M. Strange, and J. Li, "Accelerating graph analytics by co-optimizing storage and access on an fpga-hmc platform," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 239–248.

[19] S. R. Kuppannagari, R. Chen, A. Sanny, S. G. Singapura, G. P. C. Tran, S. Zhou, Y. Hu, S. P. Crago, and V. K. Prasanna, "Energy performance of fpgas on perfect suite kernels," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[20] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *international symposium on computer architecture*, vol. 42, no. 3, pp. 13–24, 2014.

[21] Intel stratix 10 mx fpgas. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html

[22] 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 *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 25–28.

[23] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A reconfigurable computing approach for efficient and scalable parallel graph exploration," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, 2012, pp. 8–15.

[24] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, "Gpop: a cache and memory-efficient framework for graph processing over partitions," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, 2019, pp. 393–394.

[25] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." 1999.

[26] R. A. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on*, 2014, pp. 549–559.

[27] J. Leskovec, D. P. Huttenlocher, and J. M. Kleinberg, "Predicting positive and negative links in online social networks," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 641–650.

[28] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[29] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. AcM, 2010, pp. 591–600.

[30] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 599–613.

[31] D. Kehlet, O. Tan, and L. Landis, "Intel stratix 10 mx fpgas revolutionizing system memory bandwidth," in *FCCM Workshop on Inte FPGA New Technology Showcase: Chiplets, High-Bandwidth memory, and eASICS*, 2019.

[32] Jedec standard no 235a. JEDEC. [Online]. Available: https://composter.com.ua/documents/JESD235A.pdf