Swift: A Multi-FPGA Framework for Scaling Up Accelerated Graph Analytics

Oluwole Jaiyeoba*, Abdullah T. Mughrabi†, Morteza Baradaran‡, Beenish Gul§, and Kevin Skadron¶
Department of Computer Science, University of Virginia
Email: *oj2zf@virginia.edu, †atmughra@virginia.edu, ‡morteza@virginia.edu, §bg9qq@virginia.edu, ¶skadron@virginia.edu

***Abstract*—Graph analytics are vital in fields such as social networks, biomedical research, and graph neural networks (GNNs). However, traditional CPUs and GPUs struggle with the memory bottlenecks caused by large graph datasets and their fine-grained memory accesses. While specialized graph accelerators address these challenges, they often support only moderate-sized graphs (under 500 million edges).**

Our paper proposes Swift, a novel scale-up graph accelerator framework that processes large graphs by leveraging the flexibility of FPGA custom datapath and memory resources, and optimizes utilization of high-bandwidth 3D memory (HBM). Swift supports up to 8 FPGAs in a node. Swift introduces a decoupled, asynchronous model based on the Gather-Apply-Scatter (GAS) scheme. It subgraphs across FPGAs, and each subgraph into intervals based on source vertex IDs. Processing on these intervals is decoupled and executed asynchronously, instead of bulk-synchonous operation, where throughput is limited by the slowest task. This enables simultaneous processing within each multi-FPGA node and optimizes the utilization of communication (PCIe), off-chip (HBM), and on-chip BRAM/URAM resources. Swift demonstrates significant performance improvements compared to prior scalable FPGA-based frameworks, performing 12.8 times better than the ForeGraph. Performance against Gunrock on NVIDIA A40 GPUs is mixed, because NVlink gives the GPU system a nearly 5X bandwidth advantage, but the FPGA system nevertheless achieves 2.6x greater energy efficiency.

***Index Terms*—Graph Analytics, High Memory Bandwidth (HBM), FPGA, Scalable Graph Processing, Accelerators**

I. Introduction

The growth of graph data in fields such as social network analysis, biomedical research, and graph neural networks (GNNs) [1]–[5] has created a greater need for efficient and scalable graph analysis [6]–[10]. However, existing hardware platforms face limitations in handling large-scale graphs. CPUs [7], [11]–[15] and GPUs [16]–[19] often result in fine-grained random memory accesses [20]–[25], which typically do not use memory bandwidth efficiently, limiting processing throughput.

The emergence of modern FPGAs, with High Bandwidth Memory (HBM) technology (up to $460\,\mathrm{GB/s}$) and custom parallel pipelines for data processing, presents an alternative to address the challenges associated with accelerating graph processing [26]–[34]. These state-of-the-art FPGAs offer significant parallelism compared to CPUs and GPUs, while maintaining a more efficient power profile. The inclusion of HBM allows graph data to be distributed across HBM channels and processed by specialized pipelines that achieve better memory access patterns, leveraging HBM's high memory bandwidth and enhancing parallelism. This feature makes HBM-enabled FPGAs a promising solution for graph processing.

From single to multi-FPGA graph processing: Prior research on single FPGA graph accelerators is comprehensive but typically supports graphs with fewer than 500 million edges [29], [32], [34]. To improve scalability, multi-FPGA methods often use a single FPGA solution replicated across multiple machines or rely on inter-FPGA interconnects [36], [37], [39]–[41], which can perform poorly in sharing fine-grained graph data. Such approaches can lead to inefficient processing, higher power consumption, increased memory usage, and complex inter-machine communication. Furthermore, scaling up a single-FPGA design within a machine is limited by the PCIe communication bandwidth.

Table I presents a performance comparison of the widely-benchmarked PageRank algorithm among various single- and multi-FPGA graph processing frameworks. To enable fair comparison, metrics such as interconnect bandwidth and memory bandwidth of the FPGAs are also provided. As shown, multi-FPGA frameworks like Foregraph [36] and FDGLib [39] often have lower throughput than single-FPGA frameworks such as ThunderGP [29], [35] across various graph algorithms and workloads. This is because PCIe-connected FPGAs (multi-FPGA frameworks) exhibit lower latency compared to network-connected FPGAs (single-FPGA frameworks) due to fast/high bandwidth on-chip HBM memory coupled with PCIe DMA, providing direct access to host memory without the need to navigate the network stack [42] [43] [44]. This challenges the presumed superiority of multi-FPGA setups. "Scaling up" refers to adding more compute power (FPGAs) to a single machine via PCIe, while "scaling out" involves adding more machines with the same compute power via a specialized network—in this context, adding more machines with a single FPGA.

Scaling up single machine multi-FPGA graph processing and addressing communication overhead: A key factor in the performance gap of multi-FPGA frameworks is the communication overhead among FPGAs. Frameworks like Foregraph [36] necessitate costly inter-FPGA communication for exchanging vertex property information at each graph iteration due to the memory-bound nature of graph processing. FPGA memory bandwidth, such as with High Bandwidth Memory (HBM) at up to $460\,\mathrm{GB/s}$, far exceeds that of inter-FPGA channels like PCIe at around $17\,\mathrm{GB/s}$

Swift: a decoupled Gather-Apply-Scatter graph execution model: In order to address the communication bottleneck

TABLE I: PageRank (PR) - Comparing Single-FPGA vs. Multi-FPGA "Scale Out" Graph Accelerators from Prior Art

Work	Lang.	Impl.	Eval.	Public	Platform	Mem (BW)	Interconnect/host (BW)	Throughput(GTEPS[a])	FPGA(nodes)	Reference
ReGraph	HLS	HW	✓		Alveo U280	460 GB/s	38 GB/s	8.037[b]	1	[29]
ThunderGP	HLS	HW	✓		Alveo U250	77 GB/s	38 GB/s	3.355[b]	1	[35]
GraphLily	HLS	HW	✓		Alveo U280	460 GB/s	38 GB/s	5.591[b]	1	[32]
ACTS	HLS	HW	✓		Alveo U280	460 GB/s	38 GB/s	8.557[b]	1	[34]
ForeGraph	HDL	Sim	-		Xilinx VC707	19.2 GB/s	98 GB/s	1.861, 3.675, 7.350, 9.8[d]	4, 8, 16, 32[e]	[36]
GraVF-M	Python	HW	✓		Microsemi	21.7 GB/s	5.85 GB/s	4.623	4[e]	[37]
GridGAS	HDL	HW	-		Xilinx KC705	-	3 GB/s	0.170[c]	-	[38]
FDGLib	HDL	HW/Sim	✓		Alveo U250	77 GB/s	12.25 Gbit/s	2.679, 5.916, 18.816, 33.026	4, 8, 16, 32[e]	[39]
Hadoop	HLS	HW	-		Alveo U250	19.2 GB/s	32 GB/s	0.046	16[e]	[40]
Swift	HLS	HW	✓		Alveo U280	460 GB/s	38 GB/s	13.168, 22.407	4, 8[f]	[current]

[a] (Giga Traversed Edges Per Second) measures graph processing performance.
[b] Geometric mean GTEPS[a] was calculated for all graphs with sizes below 300 million edges, as reported in their papers, using a single FPGA setup.
[c] Paper evaluation only for SSSP algorithm.
[d] Paper peak performance for PR on Twitter Graph.
[e] Single-FPGA per node - "scale out".
[f] Multi-FPGA per node - "scale up".

problem, we decouple the main stages of the Gather-Apply-Scatter (GAS) graph processing scheme. This separation allows pipelining and overlapping the GAS compute and memory operations on the multi-FPGA system, enabling higher throughput while processing large graphs. With Swift, our decoupled graph processing model, four operations can run simultaneously: 1) processing edges at a given region (*vertex intervals*) for a given iteration; 2) applying vertex updates to generate active frontiers at a second region. 3) exporting active vertices (*frontiers*) to remote FPGAs in a third, different region. 4) importing active frontiers from remote FPGAs in a fourth, different region. Swift's overlapping of GAS operations allows for higher utilization of available channels such as inter-FPGA communication channel (PCIe), intra-FPGA memory bandwidth (HBM), and on-chip BRAMs/URAMs. Furthermore, it improves throughput and conceals latency overheads. Swift adapts the open-source ACTS [34] FPGA accelerator for single-FPGA graph processing, and introduces *decoupled, asynchronous GAS processing* to overcome inter-FPGA communication latency and bandwidth limits, outperforming previous FPGA graph accelerator solutions.

Algorithm 1: Edge-centric Gather-Apply-Scatter Model

Data: Edges, vertices, and vertex properties
Result: Updated vertex properties (V_{prop})

$E(U,V)$: Edge E, where U=source vertex ID, V=destination vertex ID
E_{weight}: Edge weight of the edge E
$U(E)_{prop}$: Source vertex property
$V(E)_{prop}$: Destination vertex property
$V(E)_{temp\_prop}$: Temporary destination vertex property
res: Partial result (also known as *vertex update*) generated from processing edge E

```
foreach active Streaming Partition SP in graph do
    foreach outgoing edge E(U,V) in SP do
        if vertex U is active then
            res ← Process_Edge (E_weight, U_prop)
            V_prop ← Apply (V_temp_prop, res)
        end
    end
end
```

II. Background and Related Work

A. Gather-Apply-Scatter (GAS)

The Gather-Apply-Scatter (GAS) [6], [45], [46] model provides a high-level abstraction for various graph processing algorithms and is widely adopted by software-based [7], [9], [47]–[49] and accelerator-based frameworks [29], [35], [36], [50]–[53]. The two main variants of the GAS model are the vertex-centric and edge-centric approaches. Swift adopts the edge-centric variant, which facilitates high throughput streaming memory accesses, leveraging HBM's high memory bandwidth.

Algorithm 1 shows the pseudo code describing the Edge-centric Gather-Apply-Scatter graph processing model [15], [47]. As shown, this model employs streaming partitions by logically splitting the graph into intervals by source vertex IDs during pre-processing. Next, an input of an unordered set of directed edges is streamed and processed in the *Process_Edge* stage where edge data, source and destination vertex properties generate an update value (*res*). Only intervals with active vertices are processed, avoiding redundant reads to all edges. Furthermore, intervals are based on source IDs, and source vertex properties are read once from DRAM per iteration. In *Apply*, these updates are applied to destination vertices to compute new vertex properties. These functions iterate until a convergence criterion is reached.

B. ACTS: Near-Memory FPGA Graph Processing

ACTS [34] is a graph processing accelerator that utilizes the edge-centric GAS model on FPGAs and employs HBM to address the memory bandwidth bottlenecks of prior single-FPGA-based graph processing designs. The key idea behind ACTS is an online recursive partitioning mechanism that converts (via partitioning) the low-locality vertex updates generated from processing the edges of an active sub-graph, into high-locality vertex-update partitions in efficient time. This partitioning is done across the *destination* vertex IDs.

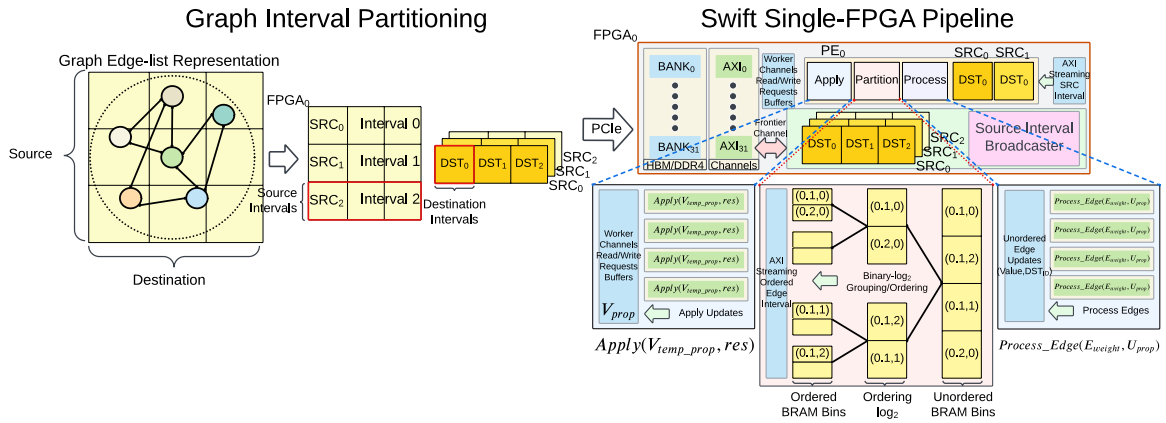


Fig. 1: How Swift (ACTS [34] based pipeline) handles Process Edge, Partition-Updates, and Apply Update operations.

Through this, ACTS improves both read and write bandwidth performance, even as graph size increases. Consequently, ACTS achieved an average speedup of 1.5×, with a peak speedup of 4.6× compared to Gunrock [16], a state-of-the-art GPU-based graph processing accelerator, on the NVIDIA Titan X GPU. Furthermore, ACTS demonstrates an average speedup of 3.6×, with a peak speedup of 16.5× over GraphLily [32], a modern FPGA-based graph accelerator utilizing HBM. These speedups are found in their paper. We therefore use this as the starting point for Swift's multi-FPGA solution.

C. ForeGraph: Scalable FPGA Graph Processing

ForeGraph [36], [39] tackles scaling by using the Catapult torus interconnect in an FPGA simulated environment; however, it cannot scale beyond 48 nodes or maintain optimal performance as the number of nodes increases. As the number of FPGA nodes increases, the interconnet becomes a bottleneck, limiting scalability and degrading performance.

III. Swift

A. Graph Processing Decoupled Pipeline

The Swift graph processing accelerator builds upon ACTS by further decoupling its pipeline into five distinct stages: process-edge, partition-updates, apply-updates, import-frontier and export-frontier operations. This decoupling allows Swift to hide latency by exploiting overlap among operations, speeding up overall execution time. Figure 1 illustrates the connection among three key stages—Process-edge, Partition-updates, and Apply-updates—involved in graph partitioning within an FPGA. Each stage interfaces with the HBM channels to receive specific data: edges for Process-edge, vertex updates for Partition-updates, and both vertex updates and properties for Apply-updates. The output data from each stage serves as the input for the subsequent stage, creating a continuous processing flow.

- **Process-edge:** This operation generates vertex-update messages from the active sub-graph (source intervals). As shown in Figure 1, edges are read from HBM into *EdgeProperty* Buffers (BRAM), and their source vertex properties are read into *VertexProperty* Buffers (URAMs). Edges of active vertices are processed, and a user-defined edge function generates vertex-update messages, following the proposed scheme in Algorithm 1. The vertex updates are buffered in DRAM for the partition-updates operation. The vertex-update tuple is formatted as (*Value, Dst*), where *Dst* is the destination vertex, and the value is the message.
- **Partition-updates:** The partition-updates stage, introduced in the ACTS paper [34], addresses the challenge of random accesses and low spatial locality in vertex-updates generated from the process-edges stage. The operation is online and happens on the device side to further decompose the vertex updates generated from the process-edges. Partitioning enhances memory locality by converting low locality vertex updates (from edge processing) into high locality, enabling efficient use of fast URAMs for updating vertices. Due to the initial static partitioning of the graph, edge and vertex layouts within each HBM channel are optimized for online partitioning, which is confined to each HBM channel. The partition-updates operationconverts low-locality vertex updates into fine-grained, high-locality vertex-update partitions. The vertex updates generated from the process-edges operation are loaded into fast on-chip URAMs and BRAMs, and then partitioned using FPGA logic into high-locality partitions. This allows updates, represented by key-value pairs, to leverage the Ultra-RAM (URAM) multi-port parallelism and high capacity in Xilinx [54] FPGAs when applied to destination vertices. However, with large graphs, URAM capacity is still limited, which can lead to partitioning overheads when swapping vertex updates. As shown in Figure 1, a recursive BRAM tree ($log_2(Dst)$) manages DRAM access latency with multilevel passes as updates move between BRAM and HBM, improving vertex-update locality with each level, thus reducing DRAM access latency. This makes it preferable to conventional bucket partitioning, especially for large graphs with low spatial locality. By breaking the task into recursive steps and buffering intermediate partial-partitioned results in HBM, the recursive BRAM tree strategy improves overall partitioning throughput and efficiency. The number of passes is the logarithm of the range of destination vertex IDs. This ensures efficient data

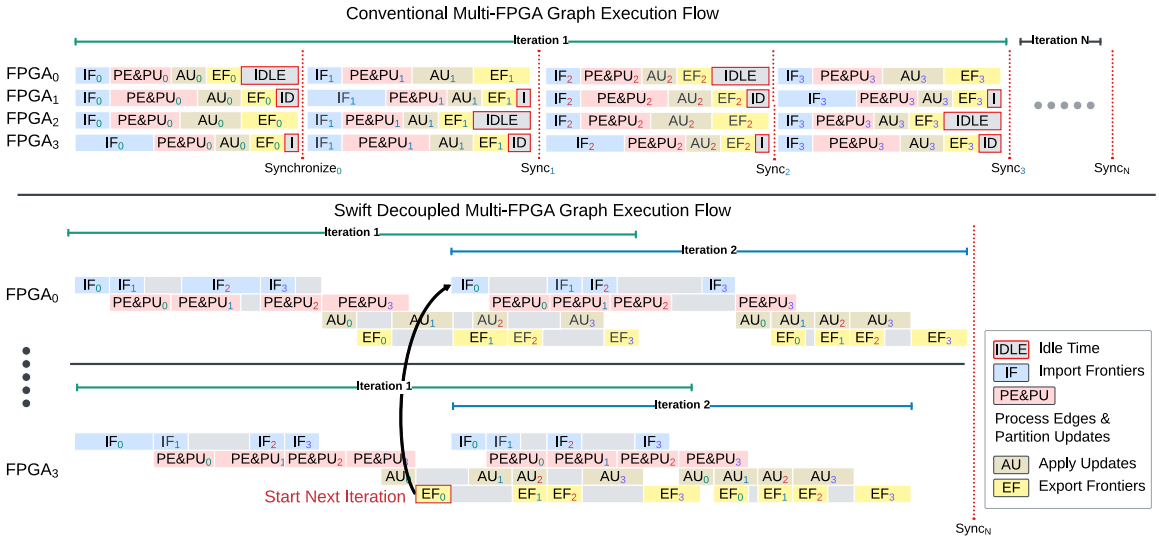


Fig. 2: Swift decoupled multi-FPGA graph execution flow compared to prior art - decoupled operations execute asynchronously on the graph with no bulk synchronization.

transfer from BRAM to HBM, surpassing bucket partitioning.

To illustrate the advantage of the recursive BRAM tree over conventional bucket-based partitioning, assume we have (N) vertex updates in HBM from processing active graph edges. Conventional bucket partitioning reads chunks of updates into the FPGA, splits them into (P) buckets based on destination vertex IDs, and writes them back to HBM. As the graph size increases, both (N) and (P) grow, requiring more partitions to maintain locality, which increases DRAM access latency and degrades performance. In contrast, the recursive BRAM tree strategy splits partitioning into successive steps, reducing latency and performance degradation. In each pass, after the buckets are filled, they are streamed into HBM, using its full bandwidth. Then in the next pass, each bucket is read back from memory (also streaming), and partitioned again, until sufficient locality is achieved. Although it may require a logarithmic number of passes for large graphs, the recursive BRAM partitioning allows for better performance than prior art, because it maintains high locality in the HBM accesses.

- **Apply-updates:** When receiving a vertex update, the apply updates stage resolves this update to its destination vertex using a user-defined Apply function. This apply operation generates an active frontier property. Because the earlier (i.e., Partition-updates) operation outputs vertex-update chunks with high BRAM locality, the Apply operation can benefit from fast URAM memory. This is because several high-locality vertex-update partitions (generated from the partition-updates stage) and their corresponding destination vertex properties are streamed into independent high-speed URAMs, each connected to a separate apply-update logic. Therefore, several updates can be applied concurrently, allowing parallelism. In this way, the Apply operation can benefit from fast URAM memory.
- **Import-frontier and export-frontier operations:** A multi-FPGA graph processing context requires periodically exchanging graph data between FPGAs. Export-frontier operations send active frontiers from a given FPGA to its remote neighbors via PCIe through the host. These active frontiers are gathered and merged at the remote FPGA end using the import-frontier operation. Host-FPGA communication uses a host-managed shared buffer for DMA transfers over PCIe. This buffer moves active vertex properties (frontiers) between host memory and FPGA's HBM during export/import operations (Section III). The DMA engine handles memory transfers, reading from host memory (H2C) and writing to FPGA, and vice-versa.

B. Understanding the Swift Pipeline

The Swift pipeline leverages the time window between edge processing within an FPGA and its next iteration to overlap with other intra-FPGA computation and inter-FPGA communication, using separate FPGA resources concurrently. Key to our flow model is that regions within the active sub-graph (vertex intervals) can start the next operation in the pipeline once dependencies are met. This contrasts to the bulk-synchronous model adopted by various prior art that require each operation to finish on the entire sub-graph before proceeding to the next. This allows for overlapping operations on two levels:

- **Inter-FPGA:** Overlapping computation (within FPGAs) with communication operations (between FPGAs).
- **Intra-FPGA:** Operations within the same FPGA, hiding expensive, throughput-limiting operations within each other in the processing pipeline and improving throughput.

Figure 2 shows Swift's decoupled execution flow versus the conventional bulk-synchronous model. In the conventional model, stages happen sequentially, starting only after the previous one is completed. For example, exporting active frontiers to remote FPGAs (export-frontiers stage) occurs only after applying updates to the active sub-graph (apply-updates stage). Similarly, processing edges (process-edges stage) occur

only after receiving all import frontiers. In Swift's model (Figure 2), a decoupled flow exploits potential overlaps within and between FPGAs. The graph is divided into partitions, each assigned to an FPGA. Within each FPGA, partitions are divided by source vertex IDs into vertex intervals, illustrated in Figure 2 during pre-processing. Graph layout details are in Section IV-B. For simplicity, four vertex intervals are shown. Each interval goes through five stages as in Section III-B. Unlike the conventional model, there's no bulk-synchronous constraint. An interval can start its next operation as soon as its dependencies are satisfied. This is explained in Section III-C.

To better understand the Swift decoupled pipeline, let us look at each overlapping feature when processing graphs:

• **Overlap between computation within an FPGA and communication between FPGAs:** In Figure 2, $FPGA_0$ starts processing edges using the process_edge operation (denoted by PE_0) in src interval 0 as soon as its active frontiers are imported from remote FPGAs. This happens concurrently with src interval 1 importing its active frontiers (IF_1). Similarly, Dst interval 0 in $FPGA_0$ exports active frontiers to remote FPGAs (EF_0), concurrently with interval 1 generating vertex updates (AU_1), allowing computation overlap within an FPGA and communication between FPGAs.

• **Overlap between multiple operations within the same FPGA:** For example, import-frontier operation for interval 1 in iteration 2 runs concurrently with process_edge and partition-updates for interval 0, and apply-updates for interval 3, overlapping operations within each FPGA and keeping HBM, URAM, and compute resources simultaneously busy.

Due to strict dependencies, some FPGA operations cannot overlap. Partition-updates and apply-updates are such operations. Apply-updates can begin only after vertex updates from process-edges are partitioned online. Additionally, process-edges and partition-updates can be merged into a single step.

C. Swift Flow Example

This section will demonstrate how Swift operates within FPGA hardware, using a cluster of four FPGAs as an example. For simplicity, we will focus on the operations within a single FPGA ($FPGA_0$.) The workload graph assigned to each FPGA is first divided into vertex intervals based on vertex IDs, as shown in Figure 2 and 3. A vertex interval consists of vertices along with their incoming edges. Inside each FPGA, five execution modules carry out the following operations: **Process-edges (PE_M)**, **Partition-updates (PU_M)**, **Apply-updates (AU_M)**, **Export-frontier (EF_M)**, and **Import-frontier (IF_M)**.

At any given time, each vertex interval can be in one of three states: "ready-for-process," "ready-for-export," and "ready-for-import." An interval in the "ready-for-process" state indicates that all dependencies needed for the process-edges operation on that interval have been met, allowing the process-edges module to execute that interval directly. The same principle applies to the "ready-for-export" and "ready-for-import" states. The modules continuously check the state of vertex intervals to carry out computation, export, and import operations.

The steps below demonstrate the Swift execution flow in FPGA hardware, focusing on one FPGA ($FPGA_0$).

1) **Initialization:** During processing initiation, all vertex intervals containing active vertices in $FPGA_0$ to $FPGA_3$ are set to the ready-to-process state.

2) **Process-edges and Partition-updates:** The process-edge module (PE_M) in $FPGA_0$ activates in the ready-to-process state, executing process-edge and partition-update operations on all vertex intervals. Concurrently, the partition-updates module (PU_M) partitions low-locality vertex updates into high-locality vertex-update partitions.

3) **Apply-updates** After generating and partitioning vertex updates, the apply-updates module (AU_M) processes each partition to create active frontiers and flags the intervals as ready-for-export.

4) **Export-frontiers:** The export-frontier module (EF_M) starts exporting active frontiers to the host CPU when triggered by the ready-for-export flag. This enables overlap between apply-updates and export-frontier operations until all vertex interval frontiers are processed. Figure 2 illustrates this with AU_0, AU_1, AU_2, AU_3 overlapping EF_0, EF_1, EF_2, EF_3.

5) **Import-frontiers:** Active frontiers associated with vertex intervals are marked ready-for-import by the export-frontier module. Thus, the import-frontier module (IF_M) in remote FPGAs can overlap their operations. This is shown by the overlap of EF_0, EF_1, EF_2, EF_3 from iteration 1 with IF_0, IF_1, IF_2, IF_3 from iteration 2 in Figure 2.

6) **Cycle Continuation:** This cycle continues until the algorithm converges (i.e., no more active frontiers) or until each vertex interval has completed a given number of iterations.

IV. Graph Partitioning and Workload Balancing

A. Graph Partitioning

Figure 3 illustrates the layout of a graph within Swift FPGA cluster. The graph is initially partitioned by its destination vertex IDs across different FPGAs. Graph partitioning is performed on the host side as a pre-processing step, as represented by different colors. Since Swift is designed for static graphs (i.e., graphs with a fixed topology), this is treated as a one-time cost that can be amortized over multiple iterations. Each data type is partitioned differently. As in some prior work, vertex properties are represented using two dimensions: source/destination. Each FPGA holds a full copy of the source vertex properties, while destination vertex properties are distributed across all HBM channels and all FPGAs, as are edges, which are partitioned by destination vertex IDs. Each processing element is connected to one HBM channel, processing edges and destination vertex properties in that channel. Each destination range and its incoming edges are assigned to a unique FPGA. Within each FPGA, the graph partition is further divided by source IDs. Each FPGA in the cluster has a dedicated HBM channel, the "frontier HBM," which accommodates active frontiers imported from the communication channel. The remaining HBM channels in the FPGA, the "worker HBMs," each store a segment of the graph's destination vertices and their incoming edges. Each processing

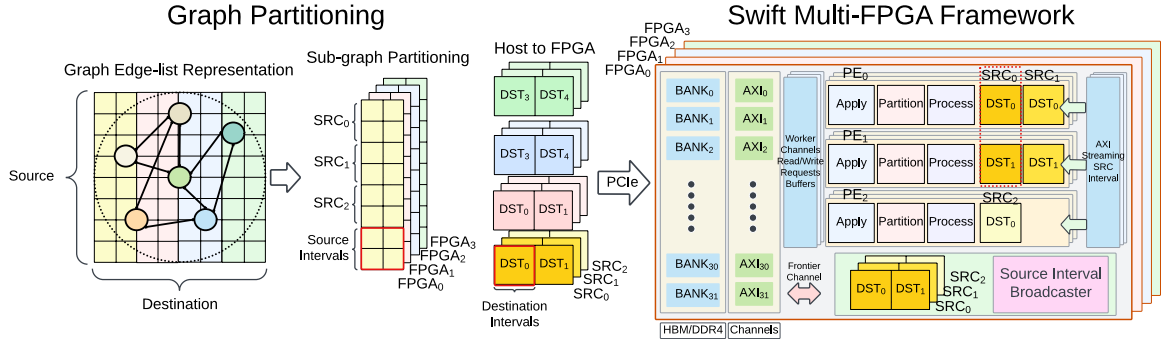


Fig. 3: Swift framework ensures the graph is partitioned into load-balanced intervals distributed across FPGAs and Processing Elements (PEs), and then processed asynchronously.

element (PE) is linked to a worker HBM and handles the edges within that specific channel. To prevent graph data duplication and maintain storage efficiency, unique edges and vertices are distributed across HBM channels.

The vertices within each HBM are categorized into vertex intervals, with the range being $\frac{V}{NUM\_PEs}$, where V represents the vertex properties that can fit in URAM. NUM_PEs denotes the number of processing elements in the cluster. Consequently, the combined range of vertex intervals across all PEs in the cluster is V.

B. Workload Balancing

Optimizing performance in a cluster-scale environment with HBM-enabled FPGAs requires an efficient workload placement strategy that leverages parallelism at multiple levels. The first level of parallelism comes from the independent FPGAs in the cluster. Each FPGA has 32 independent HBM channels, adding a second level of parallelism. Swift configuration allows up to 128 Processing Elements (PEs) to operate independently in a 4-FPGA cluster. The challenge is to prevent any straggler PE from becoming a bottleneck, which requires a graph placement strategy that ensures uniform workload balance. Prior schemes distributes the graph across machines using a pre-processing step, maintaining balance but sacrificing throughput due to the graphs' unstructured nature [55], [56]. This resulted in the creation of cutting edges across various machines, disrupting the sequential ordering of vertex IDs. As a result, vertex translation was necessary for storage efficiency, and it also introduced communication bottlenecks between FPGAs. Swift avoids translations at receiver FPGAs by consistently referencing vertices using global IDs across all FPGAs. Additionally, it enforces a vertex-interval-based strategy for workload placement. This means that all vertices and edges within a vertex interval are placed across the entire cluster before moving to the next interval. This approach allows imported active frontiers to fit into low-latency URAM, enhancing throughput. In summary, our proposed strategy for placing graph workloads balances workload distribution and optimizes throughput in a cluster-scale, HBM-enabled FPGA environment. We achieve efficient graph processing without sacrificing overall performance by addressing translation bottlenecks and leveraging parallelism.

TABLE II: Graph datasets under evaluation

Dataset	Symbol	#Vertices	#Edges	Type
Indochina	IND	7.4M	194M	Real
Twitter	TW	41.6M	1.4B	Real
Sk-2005	SK	50.6M	1.9B	Real
Uk-2005	UK	39.5M	936M	Real
Soc-sinaweibo	SN	58.7M	523M	Real
Webbase-2001	WB	118M	1.0B	Real
RMAT_8	R8	8.39M	1.07B	Syn
RMAT_16	R16	16.8M	1.07B	Syn
RMAT_32	R32	33.6M	1.07B	Syn

V. Performance Evaluation

A. Experimental Methodology

1) Graph Algorithms and Datasets: We studied three commonly used graph algorithms: Pagerank (PR), Sparse Matrix-Vector Multiplication (SpMV), and Hyperlink-Induced Topic Search (HITS) to explore their distinct contributions within Swift. These algorithms capture memory access patterns that are common to various other graph algorithms. Our experiments involved using both synthetic and real-world datasets as shown in Table II. We choose these datasets because they express diverse cache behaviors. The synthetic datasets were generated from the RMAT graph generator [58], while the real-world datasets were obtained from the University of Florida's Sparse Matrix Collection [59]. Because of the limited HBM memory capacity (8GB per FPGA), we have postponed the exploration of very large graphs for future research. Future HBMs are expected to deliver up to 32GB, allowing for much larger graphs to be run.

2) Acceleration Environment, Design, Baselines, and Performance Metrics: In our study, we conducted a comprehensive comparison of Swift with several state-of-the-art clusterscale systems, including ForeGraph (FPGA-based), PowerGraph (CPU-based), TurboGraph (FPGA-based), FPGP (FPGA-based), FDGLib (FPGA-based), and Gunrock (GPU-based). We compared Swift with Gunrock, as it was open-sourced. The complete implementation of Swift, including I/O and FPGA kernel invocation costs, was carried out using four Xilinx Alveo Ultrascale+ FPGA Accelerator Cards. These cards are

TABLE III: Benchmark Tools and Hardware Specifications

Benchmark	Devices per Node	Architecture	≈BW[a] (GB/s)	BW[b] (GB/s)	BW[c] (GB/s)	Runtime Power (W)	Freq[d] (MHz)	LUT[e]	FF[e]	BRAM[e]	URAM[e]
Swift (FPGA)	↑ 4, 8	Alveo U280	1840	1380	68 (PCIe)	100 W to 200 W	150 MHz	3480K (65.4%)	2880K (25.4%)	8004 (49.3%)	3072 (80.0%)
Gunrock (GPU) [16], [57]	↑ 4	Tesla A40	3072	3072	448 (NVLink)	480 W to 720 W	1305 MHz				

[a] Memory BW: Off-chip DDR4/HBM memory bandwidth for four (4) cards.
[b] Total Effective BW: PCIe/NVLink bandwidth between the FPGA/GPU respectively.
[c] Total Communication BW: Maximum bandwidth the algorithm can use upon deployment.
[d] Max clock freq: Maximum on-chip clock frequency per card.
[e] Total LUT, FF, BRAM and URAM utilization across 4 FPGAs

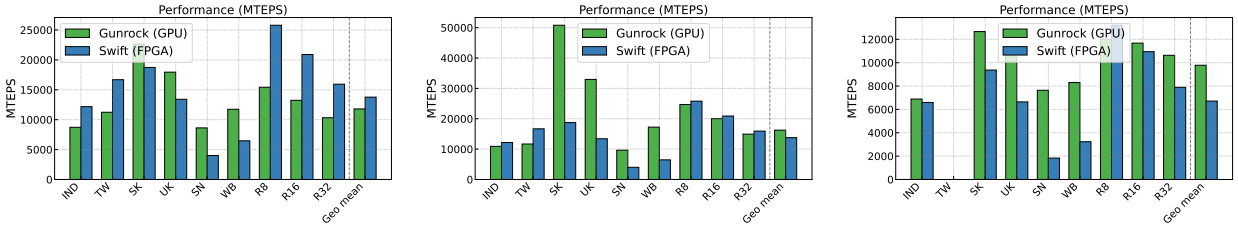


Fig. 4: Performance Comparison of Gunrock (GPU) and Swift (FPGA) for PageRank (left), SpMV (middle), and HITS (right) using for 16 iterations (trials) 4 FPGAs/GPUs.

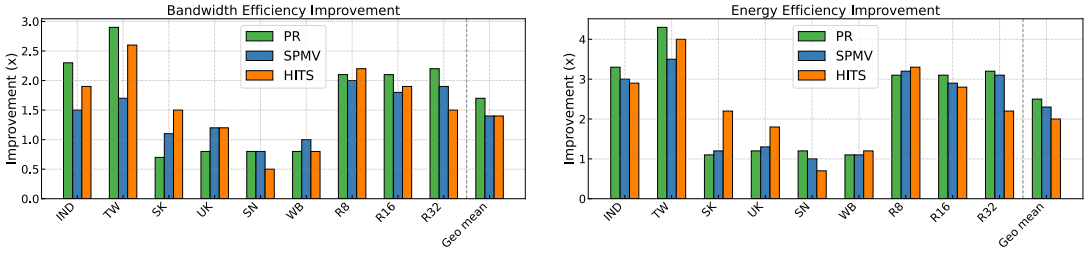
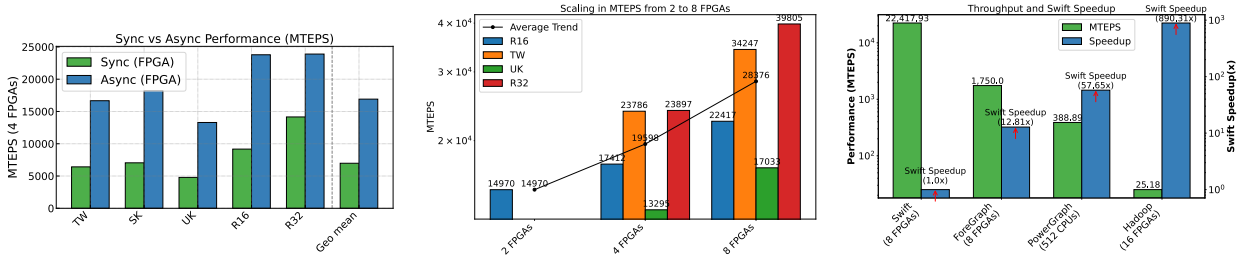


Fig. 5: Efficiency improvement for Swift over Gunrock — PR, SPMV, and HITS.



(a) 4-FPGAs performance (b) Geo mean across graph data sets (c) PR with Twitter Graph

Fig. 6: (a) Swift PageRank (PR) performance improvement (synchronous vs asynchronous) and (b) multi-FPGA scalability. (c) shows Swift MTEPS and speedup vs. ForeGraph [36], PowerGraph [45], and Hadoop [40] on Twitter Graph.

equipped with HBM (High Bandwidth Memory) capable of delivering up to 460GB/s per FPGA. Gunrock, on the other hand, was tested on four NVIDIA A40 GPUs with HBM2 memory supporting 696GB/s per GPU (as shown in Table III). Communication among FPGAs in Swift occurs via PCIe, with data routed through the host. Our model uses PCIe's duplex feature for simultaneous read/write to optimize transfer, using Gen3 x16 PCIe, which delivers up to 17 GB/s. The RTL code was generated from the C++ HLS source using the Xilinx HLS tool, and the design was synthesized and run on the Xilinx Alveo FPGA board using the Xilinx Vitis tool. It is important to note that Vitis was only able to synthesize up to 24 Processing Elements (PEs), resulting in a clock frequency of 150 MHz. Further improvements to the synthesis, to enable more PEs and a higher clock frequency, are ongoing work, but this configuration already shows the potential of Swift. Timing measurements for both Swift and Gunrock begin once the graph is loaded onto the accelerator and end upon completion of kernel processing, capturing all exchanges during execution. The graph loading time is excluded, as it is a one-time cost.

Data movement is facilitated by the host using the DMA engine, which reads graph data from the host's allocated

memory (for H2C) and writes it directly into the FPGA's HBM memory during import, and vice versa for export. The synchronization process is overlapped with the FPGA kernels performing graph processing across multiple FPGAs. This overlap is achieved through double buffering, out-of-order command queuing, asynchronous event handling, and non-blocking calls, all implemented on the host side.

B. Results

1) Throughput: Figure 4 and 5 present a comparative analysis of Swift with prior accelerators, leading to several noteworthy observations.

- Swift exhibits mixed performance compared to Gunrock in Figure 4. Some datasets such as SK-2005 and UK-2005 are characterized by high regularity and cache hit rates, and have significant benefits from the advanced caching mechanism of the GPU. In contrast, Swift demonstrates superior throughput with relatively unstructured datasets over Gunrock. We could not collect results for HITS on Swift due to out-of-memory error.
- It's important to note that the evaluation of Gunrock is on a GPU cluster using A40 GPUs with NVlink; the A40 offers higher off-chip memory bandwidth (768 GB/s vs. 345 GB/s) and NVlink offers higher inter-device bandwidth (112 GB/s vs. 17 GB/s) compared to the Alveo U280 FPGAs. To evaluate the benefit the GPU system derives from NVLink vs. the slower PCIe interconenct, we evaluated Gunrock on PageRank on R8 with NVlink disabled. Without NVlink, Gunrock is 4.8X slower than with NVlink, similar to the bandwidth difference. *This suggests that with a similar high-speed interconnect, the multi-FPGA system would consistently outperform Gunrock for all our algorithms and datasets.*
- Swift exhibits superior performance over prior multi-FPGA-based (Foregraph, Hadoop with FPGAs) and CPU-based (Powergraph) clusterscale graph accelerators. As shown in Figure 6c, Swift outperforms Foregraph by up to 12x, Hadoop by 890x, and Powergraph by 57x. This superiority can be attributed to two main factors:
  - Swift effectively manages random accesses related to vertex-to-vertex communication within each FPGA by restructuring vertex updates during processing and leveraging fast URAMs to perform apply-update operations (refer to section III-A)
  - Swift's decoupling strategy enables tight interleaving between computation (within FPGAs) and communication (between FPGAs), as well as between computation operations within the same FPGA. This reduces idle times.
- We compared Swift against the bulk-synchronous GAS approach (where no overlapping exists) to gain insights into the impact of our decoupling approach and better quantify the performance impact of overlapping communication with computation during graph processing. The result are plotted in Figure 6a. To achieve this we turned off the asynchronous behavior in Swift to enforce that each iteration completes a bulk-synchronous step before the next commences. The results prove that Swift's decoupling mechanism provides about 2-3X improvement to throughput.

2) Energy & Bandwidth Efficiency: Swift (FPGA-based) and Gunrock (GPU-based) were run on different platforms with different characteristics. The Alveo U280 FPGA has an off-chip memory bandwidth of 460GB/s, while the Tesla A40 GPU supports up to 768GB/s. To compare, we use bandwidth efficiency (MTEPS/bandwidth), and energy efficiency (MTEPS/Watt) We query GPU power using Nvidia-smi and FPGA using Xilinx's xbutil. Based on our observations, Swift experiences about 1.5X better bandwidth efficiency than Gunrock and about 2X better energy efficiency. Further profiling of power consumption in Swift revealed that as much as 80% of Swift's overall power is used by the HBM while only about 20% is spent in on-chip FPGA activity.

3) Scalability: Figure 6b shows the throughput for a number of datasets plotted across an increasing number of FPGAs, to gain insights into how Swift scales. Some datasets (TW, UK & R32) were too large to fit in a 2-FPGA setup and their 2-FPGA numbers were omitted. As shown, Swift's throughput increases relatively linearly as more FPGAs are added. This linear stability facilitated by the workload balancing mechanism (Section IV-B) allows a graph workload to be uniformly distributed across the different FPGAs in the cluster.

VI. Conclusions

The paper introduces Swift, a clusterscale graph accelerator for FPGAs with HBM. Swift leverages the open-source ACTS [34] framework and addresses key challenges not present in single-FPGA accelerators, in particular the limited bandwidth of FPGA-to-FPGA communication and inefficiency in prior workload balancing strategies. To overcome these challenges, Swift allows overlapping of crucial graph processing primitives, such as edge processing within a local FPGA, importing of active frontiers from remote FPGAs, and exporting of active frontiers to remote FPGAs. This approach maximizes communication bandwidth across PCIe, off-chip (HBM/DDR), and on-chip (SRAM), effectively concealing inter-FPGA communication with intra-FPGA computation. Swift outperforms prior FPGA-based frameworks. Results compared to Gunrock on a multi-GPU system are mixed, because the GPU system benefits from 5X higher inter-card bandwidth due to NVlink, but still achieves over 2X greater energy efficiency. If the FPGA system had a similar high-bandwidth interconnect, it should consistently outperform the GPU.

Acknowledgements

This work was funded in part by PRISM, one of seven centers in JUMP 2.0, an SRC program sponsored by DARPA; the NSF I/UCRC MIST Center, and Booz Allen Hamilton under contract FA-8075-18-D-0004. We also thank the anonymous reviewers for their helpful suggestions.

REFERENCES

[1] J. A. Ang, B. W. Barrett, K. Wheeler, and R. C. Murphy, "Introducing the Graph 500," 2010.

[2] J. Lee, H. Kim, S. Yoo, K. Choi, H. P. Hofstee, G.-J. Nam, M. R. Nutter, and D. Jamsek, "ExtraV: boosting graph processing near storage with a coherent accelerator," Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137776

[3] "Graph 500 | large-scale benchmarks," 2020. [Online]. Available: https://graph500.org/

[4] 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," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: https://dl.acm.org/doi/10.14778/2212351.2212354

[5] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph Neural Networks: A Review of Methods and Applications," *arXiv:1812.08434 [cs, stat]*, Jul. 2019, arXiv: 1812.08434. [Online]. Available: http://arxiv.org/abs/1812.08434

[6] R. R. McCune, T. Weninger, and G. Madey, "Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Distributed Graph Processing," *arXiv:1507.04405 [cs]*, Jul. 2015, arXiv: 1507.04405. [Online]. Available: http://arxiv.org/abs/1507.04405

[7] 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*, ser. SIGMOD '10. Indianapolis, Indiana, USA: Association for Computing Machinery, Jun. 2010, pp. 135–146. [Online]. Available: https://doi.org/10.1145/1807167.1807184

[8] H. Jin, P. Yao, X. Liao, L. Zheng, and X. Li, "Towards dataflow-based graph accelerator," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 1981–1992.

[9] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, "GraphLab: a new framework for parallel machine learning," in *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'10. Arlington, Virginia, USA: AUAI Press, Jul. 2010, pp. 340–349.

[10] Y. Low and Gonzalez, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, 04 2012.

[11] K. Lakhotia, R. Kannan, and V. Prasanna, "Accelerating PageRank using Partition-Centric Processing," *arXiv:1709.07122 [cs]*, Aug. 2018, arXiv: 1709.07122. [Online]. Available: http://arxiv.org/abs/1709.07122

[12] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 631–643. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/malicevic

[13] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv:1508.03619 [cs]*, May 2017.

[14] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '13. Shenzhen, China: Association for Computing Machinery, Feb. 2013, pp. 135–146. [Online]. Available: https://doi.org/10.1145/2442516.2442530

[15] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, 2015, pp. 375–386. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/zhu

[16] 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 *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. Barcelona, Spain: Association for Computing Machinery, Feb. 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2851141.2851145

[17] Y. Zhang, F. Mueller, X. Cui, and T. Potok, "GPU-Accelerated Text Mining," Mar. 2011. [Online]. Available: https://hgpu.org/?p=3174

[18] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: vertex-centric graph processing on GPUs," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, ser. HPDC '14. Vancouver, BC, Canada: Association for Computing Machinery, Jun. 2014, pp. 239–252. [Online]. Available: https://doi.org/10.1145/2600212.2600227

[19] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?" in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2014, pp. 140–149.

[20] S. Beamer, K. Asanovic, and D. Patterson, "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server," in *2015 IEEE International Symposium on Workload Characterization*, Oct. 2015, pp. 56–65.

[21] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 234–248.

[22] A. T. Mughrabi, M. Baradaran, A. Samara, and K. Skadron, "ECG: Expressing Locality and Prefetching for Optimal Caching in Graph Structures." IEEE Computer Society, May 2024, pp. 520–525. [Online]. Available: https://www.computer.org/csdl/proceedings-article/ipdpsw/2024/646000a520/1YTstNCnXCo

[23] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "P-OPT: Practical Optimal Cache Replacement for Graph Analytics," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2021, pp. 668–681, iSSN: 2378-203X.

[24] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 373–386, iSSN: 2378-203X.

[25] M. Baradaran, A. Ansari, M. Sadrosadati, and H. Sarbazi-Azad, "Energy consumption analysis of instruction cache prefetching methods," in *2023 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, 2023, pp. 60–67.

[26] R. Shi, K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "Exploiting HBM on FPGAs for Data Processing," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 36:1–36:27, Dec. 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3491238

[27] A. K. Jain, C. Ravishankar, H. Omidian, S. Kumar, M. Kulkarni, A. Tripathi, and D. Gaitonde, "Modular and Lean Architecture with Elasticity for Sparse Matrix Vector Multiplication on FPGAs," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2023, pp. 133–143, iSSN: 2576-2621. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10171480

[28] C. M. Siefert, S. L. Olivier, G. R. Voskuilen, and J. S. Young, "Observed Memory Bandwidth and Power Usage on FPGA Platforms with OneAPI and Vitis HLS: A Comparison with GPUs," in *High Performance Computing: ISC High Performance 2023 International Workshops, Hamburg, Germany, May 21–25, 2023, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, Aug. 2023, pp. 620–633. [Online]. Available: https://doi.org/10.1007/978-3-031-40843-4_46

[29] X. Chen, Y. Chen, F. Cheng, H. Tan, B. He, and W.-F. Wong, "ReGraph: Scaling Graph Processing on HBM-enabled FPGAs with Heterogeneous Pipelines," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2022, pp. 1342–1358. [Online]. Available: https://ieeexplore.ieee.org/document/9923781

[30] K. Li, S. Xu, Z. Shao, R. Zheng, X. Liao, and H. Jin, "ScalaBFS2: A High Performance BFS Accelerator on an HBM-enhanced FPGA Chip," *ACM Transactions on Reconfigurable Technology and Systems*, Feb. 2024, just Accepted. [Online]. Available: https://dl.acm.org/doi/10.1145/3650037

[31] A. T. Mughrabi, M. Ibrahim, and G. T. Byrd, "QPR: Quantizing PageRank with Coherent Shared Memory Accelerators," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2021, pp. 962–972, iSSN: 1530-2075.

[32] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, Nov. 2021, pp. 1–9, iSSN: 1558-2434. [Online]. Available: https://ieeexplore.ieee.org/document/9643582

[33] A. Shekar, M. Baradaran, S. Tajdari, and K. Skadron, "Hashmem: Pim-based hashmap accelerator," 2023.

[34] W. Jaiyeoba, N. Elyasi, C. Choi, and K. Skadron, "ACTS: A Near-Memory FPGA Graph Processing Framework," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association

for Computing Machinery, Feb. 2023, pp. 79–89. [Online]. Available: https://dl.acm.org/doi/10.1145/3543622.3573180

[35] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "ThunderGP: HLS-based Graph Processing Framework on FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 69–80. [Online]. Available: https://dl.acm.org/doi/10.1145/3431920.3439290

[36] 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*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 2017, pp. 217–226. [Online]. Available: https://dl.acm.org/doi/10.1145/3020078.3021739

[37] N. Engelhardt and H. K.-H. So, "GraVF-M: Graph Processing System Generation for Multi-FPGA Platforms," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, no. 4, pp. 21:1–21:28, Nov. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3357596

[38] Y. Zou and M. Lin, "GridGAS: An I/O-Efficient Heterogeneous FPGA+CPU Computing Platform for Very Large-Scale Graph Analytics," in *2018 International Conference on Field-Programmable Technology (FPT)*, Dec. 2018, pp. 246–249. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8742266

[39] Y.-W. Wu, Q.-G. Wang, L. Zheng, X.-F. Liao, H. Jin, W.-B. Jiang, R. Zheng, and K. Hu, "FDGLib: A Communication Library for Efficient Large-Scale Graph Processing in FPGA-Accelerated Data Centers," *Journal of Computer Science and Technology*, vol. 36, no. 5, pp. 1051–1070, Oct. 2021. [Online]. Available: https://doi.org/10.1007/s11390-021-1242-y

[40] A. Sahebi, M. Barbone, M. Procaccini, W. Luk, G. Gaydadjiev, and R. Giorgi, "Distributed large-scale graph processing on FPGAs," *Journal of Big Data*, vol. 10, no. 1, p. 95, Jun. 2023. [Online]. Available: https://doi.org/10.1186/s40537-023-00756-x

[41] Y. Zhao, K. Yoshigoe, M. Xie, S. Zhou, R. Seker, and J. Bian, "LightGraph: Lighten Communication in Distributed Graph-Parallel Processing," in *2014 IEEE International Congress on Big Data*, Jun. 2014, pp. 717–724, iSSN: 2379-7703. [Online]. Available: https://ieeexplore.ieee.org/document/6906849

[42] C. Bobda, J. M. Mbongue, P. Chow, M. Ewais, N. Tarafdar, J. C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser, M. Herbordt, H. Shahzad, P. Hofste, B. Ringlein, J. Szefer, A. Sanaullah, and R. Tessier, "The future of fpga acceleration in datacenters and the cloud," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 3, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3506713

[43] J. Lin, "Scale up or scale out for graph processing?" *IEEE Internet Computing*, vol. 22, no. 3, pp. 72–78, 2018.

[44] W. Shaddix, M. Samani, M. Fariborz, S. J. B. Yoo, J. Lowe-Power, and V. Akella, "Tegra – scaling up terascale graph processing with disaggregated computing," 2024. [Online]. Available: https://arxiv.org/abs/2404.03155

[45] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," 2012, pp. 17–30. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[46] 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. [Online]. Available: https://www.usenix.org/node/186217

[47] 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*, ser. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, Nov. 2013, pp. 472–488. [Online]. Available: https://doi.org/10.1145/2517349.2522740

[48] A. C, ""giraph: Large-scale graph processing infrastructure on hadoop," 2011.

[49] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," Mar. 2015. [Online]. Available: https://arxiv.org/abs/1503.07241v1

[50] 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*, May 2014, pp. 25–28.

[51] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "HitGraph: High-throughput Graph Processing Framework on FPGA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, Oct. 2019, conference Name: IEEE Transactions on Parallel and Distributed Systems.

[52] 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)*, Oct. 2016, pp. 1–13. [Online]. Available: https://ieeexplore.ieee.org/document/7783759

[53] 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)*, May 2016, pp. 103–110. [Online]. Available: https://ieeexplore.ieee.org/document/7544758

[54] "Vitis Unified Software Development Platform 2023.1 Documentation • Vitis Tutorials: AI Engine (XD100) • Reader • AMD Technical Information Portal." [Online]. Available: https://docs.amd.com/r/2023.1-English/Vitis-Tutorials-AI-Engine-Development/Vitis-Unified-Software-Development-Platform-2023.1-Documentation

[55] 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*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 217–226. [Online]. Available: https://doi.org/10.1145/3020078.3021739

[56] Y.-W. Wu, Q. Wang, L. Zheng, X. Liao, H. Jin, W. Jiang, R. Zheng, and K. Hu, "Fdglib: A communication library for efficient large-scale graph processing in fpga-accelerated data centers," *Journal of Computer Science and Technology*, vol. 36, pp. 1051 – 1070, 2021.

[57] "NVIDIA System Management Interface." [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface

[58] S.-W. J. et al, "RMAT generator library," 06 2018. [Online]. Available: https://github.com/sangwoojun/sortreduce/tree/master/examples/graph/utils

[59] T. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, p. 1, 11 2011.