Accelerating Graph Processing with Near-Memory Accelerator Architectures

Dissertation

presented to the faculty of the

University of Virginia School of Engineering and Applied Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

by

Oluwole Jaiyeoba

Committee:

Chair: Mircea R. Stan, CPE, UVA

Advisor: Kevin Skadron, CS, UVA

Sandhya Dwarkadas, CS, UVA

Felix Xiaozhu Lin, CS, UVA

Adwait Jog, CS, UVA

Tom Fletcher, ECE, UVa

August, 2023

oj2zf@virginia.edu

Abstract

For the past fifty years, Moore's Law and Dennard Scaling have been playing important roles in both performance and energy efficiency of computer systems. Unfortunately, they are not likely to continue, and computers no longer benefit from technology scaling as much as they did in the past. Recently, specialized hardware accelerators have emerged as a promising alternative to general-purpose computing for their potential to achieve orders of magnitude speedup and energy efficiency improvements on data-intensive applications. Graph analytics is an important data-intensive application that has gained prominence over the past years. With the rapid rise in data volumes in important applications such as social media, network analysis, genomics, knowledge graphs, etc, graph analytics allows us extract meaning from vast datasets and understand relationships between entities. However, achieving the full potential of accelerators to run graph analytics remain a challenge since the bottlenecks of such applications do not lie on computation, but data movement. To address this limitation, this thesis presents hardware and software techniques to effectively accelerate both static and dynamic graph processing workloads.

This thesis makes algorithmic (software) and architectural (hardware) innovations to accelerate graph analytics on static and evolving graphs for single- and multi-FPGA settings. First, we propose *GraphTinker* to tackle the long probe distances incurred when following edges to update dynamic graphs (i.e., edge insertions, deletions and modifications). GraphTinker combines a tree-based and open-address hashing scheme to skip entire regions within an edgelist when updating a graph. Second, we propose *ACTS* to tackle the poor scaling challenge associated with FPGA graph accelerators. ACTS embodies a set of architectural innovations that optimize off-chip HBM

bandwidth usage and increase on-chip UltraRAM (URAM) parallelism in HBM-enabled FPGAs. Both contributions allow significant improvement over prior art. Third, we propose *Swift* to tackle the challenges from frequent movement of graph data between FPGAs in a multi-FPGA setup. Swift decouples the fundamental primitives in graph processing into a partly-synchronous-partly-asynchronous model, to keep all available bandwidth (PCIe, HBM, on-chip memory) of the FPGA busy. This also results in superior performance over prior art. Fourth, we address the redundancy associated with current models for updating dynamic graphs. We propose a technique that cultivates abundant parallelism and hides graph updating tasks within graph analytics. With a trend of exponentially increasing demand for data-intensive computing, the techniques presented in this thesis will work as useful tools for the acceleration of such important workloads.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are data structures well-suited to represent the inherent relationships between different entities for a wide variety of applications, e.g., data science, machine learning, social networks, roadmap, and genomics. With the rapid growth of data and the corresponding growing development of graph-oriented tasks, the size and complexity of graphs are still expanding. This poses great challenges for modern graph processing ecosystems in performance and energy efficiency. Graph processing is poorly suited to many-core CPU/GPU architectures, as the irregular structure of graphs requires random memory accesses that prevent them from exploiting the memory- and instruction-level parallelism of such architectures.

Challenges also exist when dealing with dynamic graphs (i.e., graphs that change with time). Such graphs must be updated regularly to maintain their most recent state and also processed regularly to capture newer relationships formed between vertices in time. A basic edge consists of a source ID, a destination ID, and an edge weight. When inserting a new edge into a dynamic graph, conventional graph updating techniques use the source vertex ID of the edge to index the corresponding edgelist where the edge should be inserted. This edgelist can contain several edges depending on the outdegree of the source vertex. The edgelist is read from memory (e.g., DRAM) and traversed to find a matching edge. If a match exists, the matching edge's weight is updated. Otherwise, the new edge is appended to the edgelist. The updated edgelist is then written back to DRAM.

This process wastes DRAM bandwidth as several edges in an edgelist are read from and written to DRAM for every new edge to be inserted with only a fraction of the edgelist updated. On the other hand, running graph analytics uses DRAM bandwidth more efficiently because all edges of an edgelist read from DRAM are processed. This makes graph updating tasks demonstrate significantly lower throughput than graph analytics tasks. In a dynamic graph processing context where both graph updating and analytics tasks are multiplexed in time, Amdahl's law kicks in to degrade performance as overall throughput becomes limited by the slow graph updating process.

The age of big data has caused a consistent rise in the demand for computing power in datacenters. CPUs and GPUs employed to process graphs are known for their relatively high energy consumption. Datacenters use an estimated 200 terawatt-hours (TWh) each year and contribute to 1%

There are three main approaches to developing accelerators for graph processing: ASICs, FPGAs, and GPUs. Of these three accelerators, FPGAs appear to have advantages that make them best suited for graph processing. Hence, we focus on FPGAs as our research platform. FPGAs have advantages over ASICs. First, building a custom ASIC chip, especially a large, high-performance chip, can be expensive. This requires large volumes to amortize the Non-Recurring Engineering (NRE) costs. FPGAs provide a highly customizable fabric that can approach the performance of an ASIC at a much lower cost, especially if the initial deployment volume is not large enough to justify an ASIC. Second, a successful FPGA design can serve as a prototype for a future ASIC to justify the associated costs. Third, because FPGAs are reconfigurable, the accelerator can be continuously optimized until it is ready to be fixed into an ASIC. Fourth, when not used for graph acceleration, the FPGAs can be used to accelerate a wide variety of other applications. FPGAs also have advantages over GPUs for graph processing. The FPGA and the GPU offer fine-grained on-chip parallelism and high off-chip bandwidth (via the High Bandwidth Memory, or HBM), which positively impacts throughput. However, with the FPGA, the architect can design custom datapaths that restructure the locality of graph data and move this data between on-chip processing elements, bypassing DRAM. Data restructuring is important in graph processing to allow efficient bandwidth utilization and on-chip parallelism. This is because graph algorithms are typically memory-bound [4], and the unstructured nature of graphs forces underutilization of memory bandwidth due to

random memory accesses. Data restructuring, too, in principle, can happen with the GPU with its shared memory. However, because restructuring unstructured data requires a non-trivial capacity of on-chip memory to avoid certain DRAM access latency overheads, the smaller capacity of shared memory of the GPU compared to the scratchpad of the FPGA is a limitation.

In this dissertation, I observe that efficient use of available DRAM bandwidth within an accelerator is important in improving the throughput and scalability of graph processing tasks for static and dynamic graphs. Accordingly, **I hypothesize that (1) reducing the probe distance when searching through edgelists to update dynamic graphs is critical to both throughput and load stability, and (2) restructuring the spatial locality of messages passed between vertices (also known as vertex updates), rather than the vertices and edges of the graph, would allow efficient scaling in throughput across increasing graph sizes in FPGA-based environments**. To help investigate this hypothesis, this dissertation first addresses the long probe distance associated with updating dynamic graphs (i.e., edge insertions, deletions, and modifications) and proposes strategies to reduce this distance without sacrificing throughput during graph processing. In this work, I also propose data structures for dynamic graphs that yield high throughput when used for both graph updating and graph analytics tasks. Considering the advantages of the FPGA, as mentioned earlier in this chapter, I shifted my research environment to the FPGA. I observe that graph analytics throughput scales poorly for both static and dynamic graphs when the size of graphs to process on the FPGA increases. I first study this for a single FPGA performing static graph analysis and then extend my solution to dynamic graphs and to multi-FPGA configurations. In the multi-FPGA setting, there is also a need to combat the bottleneck of frequent movement of graph data between FPGAs across a limited-bandwidth communication channel (e.g., PCIe). I propose innovations that hide communication-related activities within the entire graph processing flow. Finally, I addressed the wastage of bandwidth when updating dynamic graphs. This problem makes updating a dynamic graph significantly slower in throughput compared to running graph analytics and impedes overall performance in dynamic graph processing contexts according to Amdahl's law. These contributions are summarized below.

GraphTinker: A High-Performance Data Structure For Dynamic Graph Processing. This

research was motivated by a common tradeoff with state-of-the-art data structures for dynamic graphs. To insert, delete or modify a graph with an edge update, a typical requirement is for an edge update to search through an edgelist to find a match before implementing the insertion, deletion, or modification command. The number of edges traversed when following edges during this search is termed *probe distance*. Several data structures demonstrate high throughput when updating the dynamic graph (i.e., edge insertions, deletions, and modifications) at the cost of low throughput when running graph analytics on the same, or vice-versa. For example, compact data structures (e.g., those based on adjacency lists [5]) support reading edges from DRAM at high throughput and are efficient for graph processing. However, they suffer long probe distances when following edges to perform graph updates because of their tightly packed, unsorted representation. We propose GraphTinker to address this. GraphTinker merges two ideas to achieve the combined benefit of short probe distance (for high-throughput graph updating) and compactness (for high-throughput graph analytics). To achieve high-throughput graph updating, GraphTinker incorporates a novel representation (called Tinker) that combines a tree-like algorithm when following edges and a well-known open hashing algorithm (Robin Hood Hashing). These allow GraphTinker to skip entire regions in an edgelist when searching for a matching edge. To achieve high-throughput graph analytics, GraphTinker maintains a highly compact representation of edges (called Coarse Adjacency Lists) tailored to Tinker that allows sequential streaming of multiple edgelists from DRAM during graph analytics. These contributions allow GraphTinker to demonstrate up to 3.3X superior throughput compared to a prior state-of-the-art data structure for dynamic graphs based on adjacency list [5] and 10X improvement when used to run graph analytics.

ACTS: A Near-Memory FPGA Graph Processing Framework. The advantages the FPGA provides for graph processing earlier discussed motivated me to switch research platforms from the CPU to the FPGA. A notable limitation of prior single-FPGA-based graph accelerators is poor scaling. As the size of the graph to process increases, degradation in throughput becomes more severe. This limits their applicability in the real-world processing of large graph workloads. This problem stems from a widely adopted strategy known as *graph slicing* where a large graph is first sliced during pre-processing into partitions to improve locality, after which all partitions are loaded

to the FPGA, and each is processed one at a time on-chip. Slicing restructures the locality of the graph. Processing of each slice can now benefit from the FPGA's fast but limited-capacity Ultra-RAM (URAM) to perform random accesses. A principal requirement for slicing is that the vertex properties of each slice fit in on-chip URAMs. This work shows that this approach limits scaling, as slicing unstructured workloads such as graphs creates shards that cannot be entirely disjointed. Therefore, processing each shard depends on vertex property data in other shards. This dependency introduces redundancies where several vertex properties are read more than once in each graph iteration, which hurts throughput. This redundancy is exacerbated with larger graph sizes which limit scaling. I tackled this problem by restructuring the locality of vertex updates generated during processing rather than the graph itself. This allows each active vertex to be read only once in each graph iteration, resolving redundancy. Restructuring vertex updates during processing can be a high-overhead task that can limit throughput. I tackled the overheads associated with the online restructuring of vertex updates by proposing a novel recursive partitioning strategy. I also propose an edge-packing strategy to eliminate on-chip data dependencies from multiple edges or vertex updates accessing the same URAM on-chip (due to a graph's unstructured nature). Unlike many other simulation-based works, our design is implemented on real hardware. It delivers a significant speedup of up to 16.5× compared to the state-of-the-art FPGA accelerator based on the HBM. ACTS also delivers speedup over GPU-based solutions showing a geometric mean speedup of up to 1.5x.

Swift: Accelerated Graph Processing with Multiple FPGAs. The success of our single FPGA work motivated us to explore graph processing in multi-FPGA environments equipped with the HBM. The research question was: "How do multi-FPGA accelerators compare with single-FPGA accelerators in throughput? And are multi-FPGA graph accelerators able to scalable efficiently with an increasing number of FPGAs?". Our findings revealed that state-of-the-art multi-FPGA solutions sometimes exhibited the same (and sometimes even worse) throughput compared to their single-FPGA counterparts. This is caused by the low bandwidth PCIe communication channel between FPGAs and the frequent substitution of on-chip data during processing due to graph slicing. Graph processing involves heavy amounts of data movements with very light-weight computation

performed on the data, so excessive communication between FPGAs is a bottleneck. For example, the Xilinx Alveo U280 board equipped with HBM can deliver off-chip DRAM bandwidth of 460GB/s within an FPGA but supports only 16GB/sec from its PCIe Gen4x8 when sending data to another FPGA. To tackle this problem, we propose a decoupled, asynchronous graph processing model based on the Gather-Apply-Scatter (GAS) paradigm. By decoupling the fundamental operations in clusterscale graph processing – processing of edges, importing of graph data from remote FPGAs, and exporting of graph data to remote FPGAs –, we can overlap these processing operations and keep all available bandwidth (PCIe, HBM, on-chip memory) of the FPGA constantly busy across the entire graph processing flow. We also propose a graph placement technique to exchange unstructured active vertex properties between FPGAs at high throughput. Swift exhibit superior throughput over several prior art FPGA solutions and up to 2.6x bandwidth efficiency over a state-of-the-art GPU accelerator.

Dynamic ACTS: A New Approach to Updating Dynamic Graphs. Employing our graph accelerator to process evolving graphs revealed a major challenge — updating a dynamic graph (i.e., edge insertions, deletions, modifications) can be significantly slower (up to 80X) compared to running graph analytics on the same. Unlike graph analytics, graph updating required searching through edgelists for a match and discarding many after this match is found. This bottlenecks the processing flow in a dynamic graph processing context according to Amdahl's law. We revisited the fundamental algorithmic objective of updating dynamic graphs — for edge updates and their matches to meet. We propose a new direction for updating dynamic graphs. Rather than traversing edgelists of a graph to find matching edge slots, we hash a group of edge updates to a large URAM array in the FPGA using a suitable hashing formula. Then, as edges are streamed from HBM during graph analytics, each edge is hashed by the same hashing function to find the appropriate slot in the URAM containing its update (i.e., it has any). The edge then picks and applies its update inflight during graph analytics. This $\mathcal{O}(1)$ runtime complexity boosts the parallelism achieved during graph updating and allows graph updating to be hidden within graph analytics. Our results show superior throughput over prior art dynamic data structures.

The outline of this dissertation is listed as follows. Chapter 2 illustrates preliminary background

information on FPGAs and graph processing. In Chapter 3, I propose GraphTinker, a high-performance data structure for dynamic graph processing. In Chapter 4, I propose ACTS, a scalable FPGA accelerator. In Chapter 5, we propose Swift, a high-performance multi-FPGA accelerator. In Chapter 6, I propose Dynamic-ACTS, a new approach to updating dynamic graphs. We summarize this dissertation and discuss future directions in Chapter 6.

Chapter 2

Background

This chapter presents the background and related work of this thesis. We introduce graph processing and discuss the major abstractions used to process graphs and the particular abstraction chosen in this dissertation. The FPGA allows us to achieve high-performance graph processing because it allows fine-grained parallelism. More importantly, it allows the computer architect to design specific datapaths that restructure and route data on-chip, bypassing the DRAM. We introduce the FPGA architecture, the memory technology (HBM), and the high-level synthesis-based programming methodology.

2.1 Graph Processing Abstractions

In this section, we discuss the most relevant paradigms used to express computation in graph processing systems. Programming models for graph processing have been studied and documented in the literature [6] [7]. The key distinguishing property between the various paradigms is the granularity of the unit of computation.

2.1.1 Vertex-centric ("Think like a vertex")

This is a very popular abstraction for large-scale distributed graph processing. The vertex-centric model places the vertex at the center of the computation and forces the user to express the computation from the point of view of a single vertex by providing a single higher-order function. A vertex-centric program receives a directed graph and a vertex function as input. A vertex serves as the unit of parallelization and has a local state that consists of a unique ID, an optional vertex value, and its outgoing edges, with optional edge values. Vertices communicate with other vertices through messages. The vertex-centric model is general enough to express a broad set of graph algorithms. It is a good fit when the computation can be expressed as a local vertex function that only needs to access data on adjacent vertices and edges.

2.1.2 Edge-centric ("Think like an edge")

A main challenge with the vertex-centric approach is that it first reads vertices before making random accesses to their set of edges. Because most graphs have a much larger number of edges than vertices, access to edges dominates the processing cost. The edge-centric approach was proposed [8] to tackle this problem by iterating over edges and updates on edges rather than over vertices. This allows the edge-centric approach to altogether avoid random access into the set of edges, instead streaming them from memory and making random accesses to their corresponding vertices. This is beneficial because streaming over edges and performing random accesses for vertex updates is better than vice-versa, especially if multiple updates to a vertex can be coalesced. There is performance gain because random access to any storage medium delivers less bandwidth than sequential access. The most popular edge-centric variant is employed by [9]. In this variant, the system that processes edges and generates messages (known as *vertex updates*) is decoupled from the system that performs these updates at their respective destination vertices. This allows opportunities for restructuring the locality of the vertex updates to accelerate performance.

2.1.3 Subgraph-Centric ("Think like a subgraph")

Unlike the vertex-centric and edge-centric abstractions, which are fine-grained, the subgraph-centric abstraction is a coarse-grained abstraction. This abstraction has been employed by Giraph++ [10] and other prior works [11], [12]. In this abstraction, the subgraph is the unit of parallel computation. A subgraph is a collection of associated vertices in a graph, usually one that has a few edges connecting to other subgraphs. By exposing the subgraph to the user function, this abstraction can reduce communication and exploit the subgraph structure to accelerate the convergence of vertex-centric programs. The subgraph-centric model relies on the perception of each subgraph as having defined characteristics rather than just a collection of unassociated vertices. While in the vertex-centric model, a vertex is restricted to accessing information from its immediate neighbors, in the subgraph-centric model, information can be propagated freely inside all the vertices of the same subgraph. This simple property of the subgraph-centric model can lead to significant communication savings and faster convergence in some algorithms.

2.1.4 The Abstraction used in this Dissertation: Edge-centric

ALGORITHM 1: Edge-centric model

foreach *active Streaming Partition* SP *in pivot* **do**
  foreach *outgoing edge* $E(U, V)$ *in* SP **do**
    if *vertex* U *is active* **then**
      $Res =$ **Process_Edge**$(E_{weight}, U_{prop}, V_{prop})$
      $Vtemp =$ **Apply**$(V_{tempprop}, res)$
    end
  end
end

The model employed in this dissertation is the edge-centric model. Several state-of-the-art software-based [8, 10, 13–15] and accelerator-based [9, 16–21] frameworks are based on this model. The

edge-centric model allows the system that generates messages (the Process_Edge stage of algorithm 1) to be decoupled from the system that updates vertex values (the Apply stage of algorithm 1). This decoupling opportunity opens the door to explore research opportunities to design custom datapaths that restructure the spatial locality of messages and maximize off-chip bandwidth efficiency. The edge-centric variant is amenable to a wide memory bandwidth environment like the FPGA-HBM because the wide bandwidth memory facilitates high throughput when streaming edges from memory. The main memory traffic in graph processing workloads is incurred by edge accesses, which can be performed in a streaming manner, as explored in [8]. The input is an unordered set of directed edges of the graph. Undirected edges in a graph can be represented by a pair of directed edges. To process subgraphs within a graph efficiently, the edge-centric model employs Streaming Partitions [8] where the graph is logically split into different intervals by source vertex IDs during preprocessing. In a given graph iteration, only intervals consisting of active vertices are processed. This prevents all edges from being read in every iteration. In the Apply stage, the update tuples generated in the Process_Edge stage are *applied* to the destination vertex to compute the new vertex property. Several iterations of these two functions are repeated until the termination criterion is met.

2.1.5 Graph Algorithms

Throughout the dissertation, we discuss four different fundamental representative graph algorithms. These are the core kernels and building blocks representing the majority of graph processing runtimes in many applications. These algorithms were chosen as benchmarks because they demonstrate memory access behaviors that generalize across various graph processing algorithms.

PageRank (PR): PageRank (PR) is an important algorithm used to rank websites by search engines. It works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites will likely receive more links from other websites. The PageRank algorithm calculates scores of vertices in a graph based on some metric (e.g., popularity). Web pages are represented as vertices,

and hyperlinks are represented as edges. The equation below shows how the PageRank score is calculated for each vertex. α is a constant, and U_{deg} is the out-degree and a constant property of vertex U. In PageRank, all vertices are considered active in all iterations. PageRank has a time complexity of $O(E \cdot K)$ where E is the number of edges and K is the number of iterations.

$$Vscore = \alpha + (1 - \alpha) \cdot \sum_{U|(U,V)\in E} \frac{U_{score}}{U_{deg}}$$

Single Source Shortest-Path (SSSP) This graph traversal algorithm computes the distance between a single source and all other vertices in a weighted graph. Like Breadth First Search (BFS), the algorithm iteratively explores neighboring vertices from starting vertices and assigns the distance to each vertex connected to the active vertices of the iteration. The main difference between BFS and SSSP is that SSSP utilizes edge weights to determine distance, while BFS does not. The equation below shows how the distance is determined for each vertex adjacent to active vertices. SSSP has a time complexity of $O(V^2)$ where V is the number of vertices.

$$V_{dist} = \sum_{U|(U,V)\in E} (V_{dist}, U_{dist} + E_{weight}(U, V)))$$

Hyperlink Induced Topic Search (HITS): Hyperlink Induced Topic Search (HITS) is an algorithm used in link analysis. It is used to discover and rank the web pages relevant to a particular search. This algorithm originated from the fact that an ideal website should link to other relevant sites and be linked by other important sites. HITS uses hubs and authorities to define a recursive relationship between web pages. HITS rates nodes based on two scores, a hub score, and an authority score. The authority score estimates the node's importance within the network, while the hub score estimates the value of its relationships to other nodes. In HITS, all vertices are considered active in all iterations. The Hub score and authority scores of each vertex in HITS are calculated using the formula below:

$$Eachnode'sHubscore = \sum(Authorityscoreofeachnodeitpointsto).$$

$$Eachnode'sAuthorityscore = \sum(Hubscoreofeachnodepointingtoit).$$

Sparse Matric Vector Multiplication (SPMv): Sparse matrix-vector multiplication (SpMV) [22] is a fundamental computational kernel used in scientific and engineering applications. They are widely used for many scientific computations, such as graph algorithms, graphics processing, numerical analysis, and conjugate gradients. This problem is a simple multiplication task where the worst case (dense matrix) has a complexity of $\mathcal{O}(N^3)$. The key feature of the problem is that the majority of the elements of the matrix are zero and do not require explicit computation.

2.2 FPGAs as Accelerators

A basic FPGA architecture (Figure 2.2) consists of thousands of fundamental elements called configurable logic blocks or CLBs (1) surrounded by a system of programmable interconnects (2), called a fabric, that routes signals between CLBs. The routing interconnect of an FPGA is static and consists of wires and programmable switches that form the required connection. Routing signals between CLBs is done during compilation. The FPGA also consists of Input/Output (I/O) blocks (3) that interface between the FPGA and external devices. The CLB is the basic repeating logic resource of an FPGA. When linked together by routing resources, the components in CLBs execute complex logic functions and memory functions. Depending on the manufacturer, the CLB may also be referred to as a logic block (LB), a logic element (LE), or a logic cell (LC). We have witnessed the successes of FPGA-based accelerators in the industry. Both Microsoft and Baidu have adopted FPGA-based accelerators to accelerate production workloads, such as the Bing search engine and machine learning platforms, at a large scale. Amazon, Nimbix, and Alibaba have deployed FPGAs into their cloud computing environments.

An individual CLB (Figure 2.2.2) comprises several logic blocks. The number and arrangement of components in the CLB varies by device; the simplified example in Figure 2.2 contains two three-input LUTs ①, an full adder FA ③ and a D-type flip-flop ⑤, plus a standard mux ② and two muxes, ④ and ⑥, that are configured during FPGA programming. The key building block in the CLB is a lookup table (LUT) that can implement any truth table for an n-bit input, and these

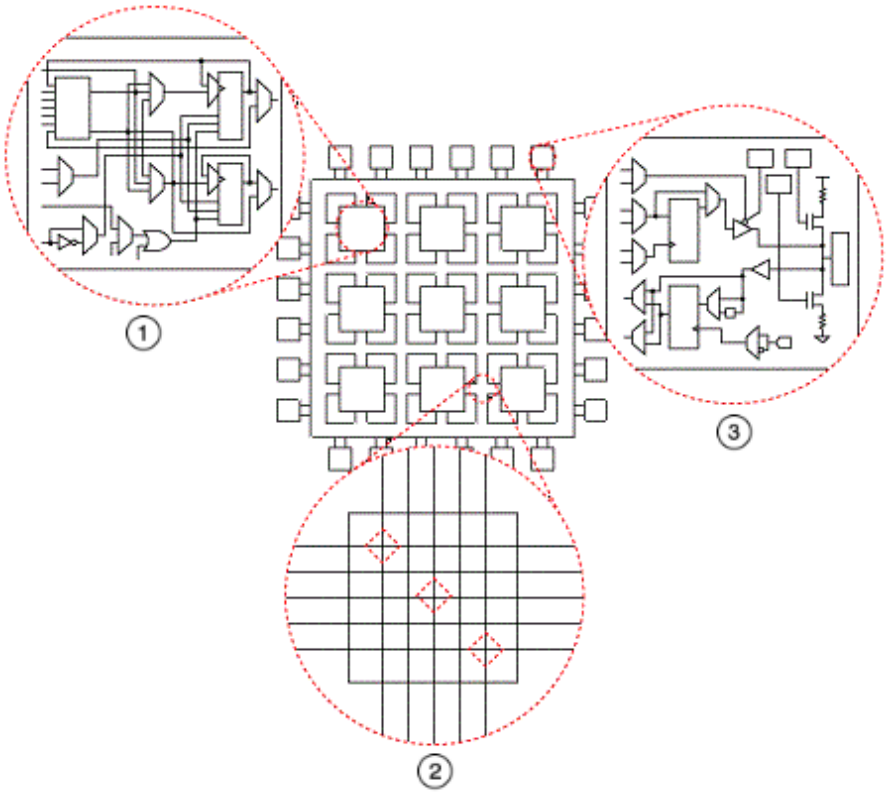


Figure 2.1: The fundamental FPGA architecture [1]

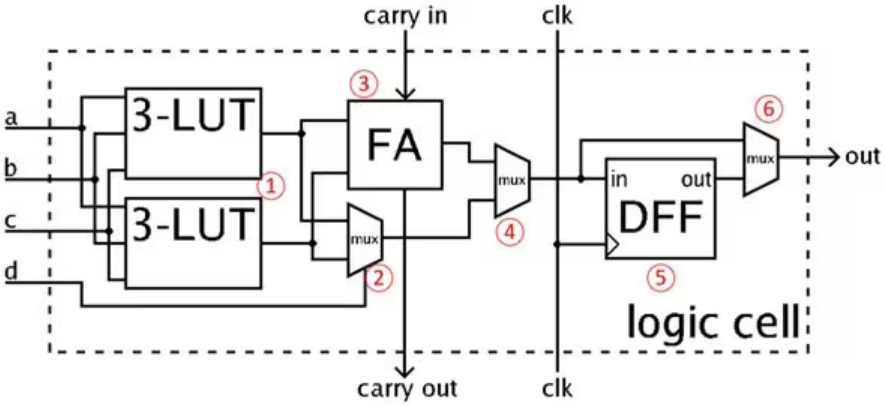


Figure 2.2: A simplified CLB: The four-input LUT is formed from two three-input units [1]

LUTs can then be combined to implement more complex combinational functions. The FPGA also consists of memory blocks called Block RAMs (BRAMs) and Ultra RAMs (URAMs) for storing data on-chip. Both memories are random access memory embedded throughout an FPGA for data storage. Block RAMs come in a finite size, 4, 8, 16, or 32 kb (kilobits) are common. Depending on the application's needs, they have a customizable width and depth, and multiple BRAMs can be linked together to create larger BRAM sizes. The BRAM is analogous to the last-level cache in the CPU or the CPU scratchpad because it is a high-speed memory used to hold small items of data in the FPGA for rapid retrieval. The UltraRAM (or URAM) is a high-density memory building block. Each URAM can store up to 288K bits of data and is configured as a 4K x 72 memory block. UltraRAM has eight times the capacity of a block RAM. Both URAM and BRAM are natively one clock latency reads and writes. The UltraRAM is intended to replace off-board memories enabling better overall performance. The BlockRAM is smaller than URAM but more flexible, allowing multiple blockRAMs to be easily cascaded to make larger memories. The UltraRAM, on the other hand, is larger (having up to eight times the capacity of a block RAM) but less flexible in data width and address space configuration than block memory. Reconfigurable interconnections allow different logic blocks to be wired together to form a large logic design. I/O blocks the interface between the FPGA and external devices. Any design that can fit within the available resources can be implemented on the FPGA through these programmable blocks. FPGAs also incorporate dedicated digital signal processing (DSP) blocks to accelerate commonly used, complex functions such as multiplication, fast Fourier transforms (FFTs), and finite impulse response filtering (FIR), which would otherwise take significant resources if implemented directly using logic (i.e., using LUTs and flip-flops).

CPUs, GPUs, FPGAs, and ASICs are also hardware solutions for graph processing. Graph processing requires abundant fine-grained parallelism in a processing platform. This is because a typical graph has many vertices, and graph processing models require user-defined computations to be performed in parallel on each of these vertices during processing. The CPU's inability to provide this abundant level of fine-grained parallelism makes it less preferred to the FPGA, as long as the parallelism achieved in the FPGA makes up for its lower clock speed (2-3 GHz for CPUs vs. 200-

500 MHz in a typical FPGA implementation). The FPGAs and ASICs have a greater potential for the tasks outlined in this dissertation than the GPU. With FPGAs and ASICs, specialized on-chip datapaths can be designed to restructure graph data's static locality in real time. This is amenable to the unstructured nature of graphs and the memory-centric paradigm of graph analytics because it allows graph data to be moved between arbitrary memory locations in DRAM with few clock cycles. Also, it allows graph data to move between arbitrary processing elements (PEs) within the FPGA, bypassing the DRAM. The GPUs are not a good architectural fit because there are limitations to how data can move between on-chip elements. For example, shared memory in the GPU is only visible to threads in the same block. Hence threads within two different blocks cannot communicate without passing through the DRAM. The FPGAs and ASICs also excel over the GPU regarding energy efficiency. They omit the branch divergence and provide finer-grained parallelism, delivering better performance or energy efficiency, especially when performing concurrent fixed-point operations. FPGAs are preferred to ASICs because of their flexibility. FPGAs allow reconfigurability and reduce design costs by eliminating whole classes of design problems related to ASICs. These problems include transistor-level design, testing, signal integrity, crosstalk, I/O design, and clock distribution.

2.3 High-Level Synthesis

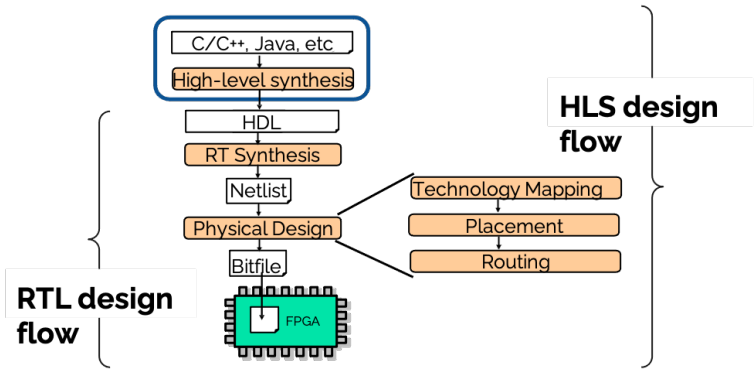


Figure 2.3: HLS synthesis flow

With the FPGA, computer architects can map computationally intensive applications with large amounts of parallelisms onto the FPGAs to execute specific tasks orders of magnitude faster than general-purpose CPUs while consuming a fraction of the power. The main problem with mapping is that it makes the design of these already complex Integrated Circuits even more complex. Thus, new design methodologies are required to facilitate their design. One technology that is being fully embraced, especially for designing accelerators, is High-Level Synthesis (HLS). HLS takes as input an untimed behavioral description in, e.g., C or C++ and generates efficient RTL code that can execute it (Verilog or VHDL). Figure 2.3 shows FPGA synthesis flow with and without HLS. Raising the level of VLSI design abstraction from the RT level to the behavioral level has several advantages. First, it reduces the turn-around time as it requires writing fewer lines of code, making the design and verification much easier. It has been reported that a single line of C code produces, on average, 7× more gates than a single line of Register Transfer Level (RTL) code [9]. Second, it allows us to simulate faster, facilitating the verification process. Last, it facilitates the re-usability of the behavioral description by allowing the generation of micro-architectures with different characteristics by simply using different synthesis options. This implies that a designer only needs to design and verify the behavioral description once and can then generate a micro-architecture with specific power, performance, and area for a particular project by setting the synthesis options to a particular value. This dramatically extends the re-usability of the design. This advantage is also a weakness as it implies that designers must fully understand how the different options interact and how the HLS process works in detail to obtain the desired micro-architecture. This implies that hardware knowledge is still very much required when using HLS.

As shown in Fig. 2.4, high-level synthesis is divided into two steps: i) parse the software program and generate the Control-Data Flow Graph (CDFG), which captures the control/data dependencies of the original program. ii) map the CDFG into a statically scheduled or dataflow circuit.

CDFG. HLS first generates a control flow graph (CFG), the standard data structure for optimizing software programs. Nodes in this graph are basic blocks, which correspond to a collection of consecutive sequential statements with a single entry and exit point. Outgoing edges from a basic block correspond to different potential successors, with the successor chosen based on a specified

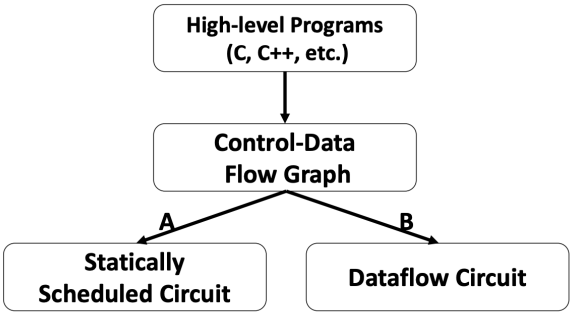


Figure 2.4: HLS overview

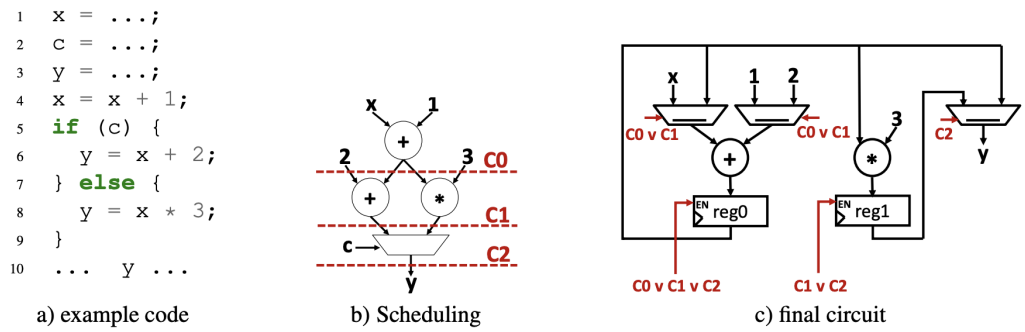
condition. For-loops and while-loops result in cycles in the CFG. Then, a data flow graph (DFG) is generated for each basic block to capture the data dependencies among instructions within it. A CDFG is the combination of CFG and DFGs for all of the basic blocks.

Static or dynamic dataflow circuit. HLS converts the generated CDFG into a statically scheduled circuit, which consists of a datapath that contains all the operations (e.g., addition, multiplication, etc.) from the program and a finite state machine that schedules these operations into clock cycles. The state machine serves as a global scheduler that controls the execution sequence of the whole circuit.

2.3.1 A Simple Example

Fig. 2.5 (a) shows an example code that has two add operations and one multiply operation, and the variable y is redefined in the two branches of the if statement. Fig. 2.5 (b) shows the scheduling result of the static HLS engine. The two branches (Line 6 and Line 8) are scheduled in C1, and the preprocessing (Line 4) and postprocessing (Line 10) codes are scheduled in C0 and C2, respectively. There are two add operations in different clock cycles, and the HLS engine allocates one adder and binds the two add operations to it. Fig. 2.5 (c) shows the final circuit with one adder and one multiplier. It also has two registers to store the intermediate results from the adder and the multiplier, respectively. Furthermore, it has three multiplexers; two are used to share the adder in

C0 and C1, and the last is used to select the right y from the if branches.



a) example code b) Scheduling c) final circuit

Figure 2.5: Simple HLS Example [2]

2.3.2 HLS Optimizations

HLS tools use a C/C++ front end and a set of transformation heuristics to map software constructs onto hardware elements and a back end that generates RTL code [17, 24]. To satisfy resource, layout, and timing requirements, a constraint solver is typically deployed [25]. To guide the transformation, programmers can add #pragma hints. HLS tools make a heuristic effort to translate any valid C/C++ program to RTL. In this section, we present some of the most important pragmas that programmers can use to improve their design performance in HLS. FPGAs enable designers to extract parallelism at a finer granularity to improve performance. Parallelism can be defined at various levels in HLS. It can be the operations within a loop or the parallelism between multiple functions. Programmers can use directives (via pragmas) to express parallelism in their code.

Loop pipelining. Loop pipelining is used to define parallelism at the instruction level. Figure 2.6 shows an example of a simple loop that performs simple arithmetic operations on two vectors (A and C). The loop includes a read operation to read one element of input A, a multiplication operation, and a write operation to write the result to the output array (C). Figure 2.6 shows the loop execution cycles with and without the pipelining. A loop without the pipeline pragma processes the three operations sequentially and waits for all the operations in the previous iteration to finish before starting the new iteration. When a pipeline pragma is used, the next iteration is started

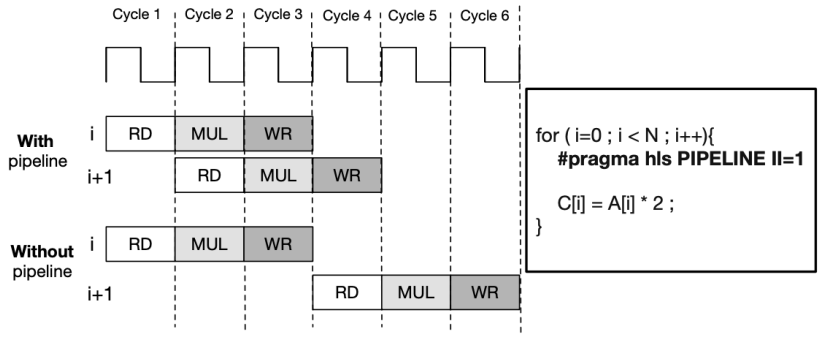


Figure 2.6: Loop pipelining example in HLS [2]

immediately after the module can accept new input. The initiation interval (II) defines how fast the next iteration can begin. II is set at the cycle level. In general, it is desirable to have loops with II=1. In some cases, the tool may not reach the designer's desired initiation interval. For example, loop-carry dependencies can prevent loop pipelining. When this occurs, the tool selects the minimum possible II from that loop. Several other pragmas can be used to provide additional information to the tool to overcome loop-carry dependencies, but we do not explain them here.

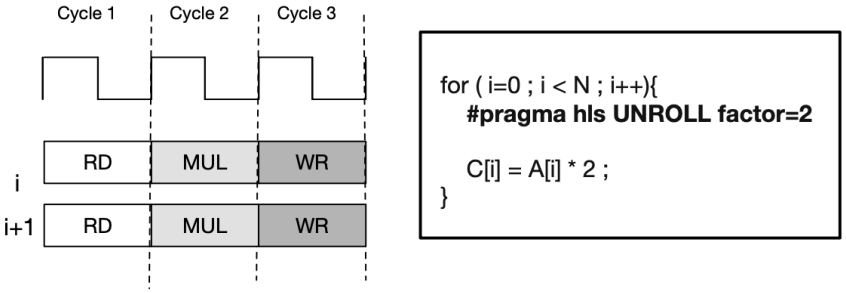


Figure 2.7: Loop unrolling example in HLS [2]

Loop unrolling. Unrolling is another type of parallelism that can be defined within a loop. Unrolling makes multiple copies of the operational module that can then execute in parallel. A user can set the unrolling factor. Figure 2.7 shows the earlier example, this time with an unrolling pragma. In this example, the unrolling directive instructs the HLS tool to create two copies of the multipliers. A loop can be unrolled partially or completely. Loop unrolling represents an area/per-

formance trade-off. Unlike loop pipelines, unrolling can only be applied when loop iterations are known at compile time. Like the loop pipeline, the compiler may fail to achieve the designer-requested unrolling factor. Insufficient resources are one of the most common reasons for this. In our example, if there are not enough memory ports to read input for multiple copies of the loop body, then the compiler cannot unroll the design as requested. We will discuss how the memory units can be partitioned to meet the requirements for the hardware modules later.

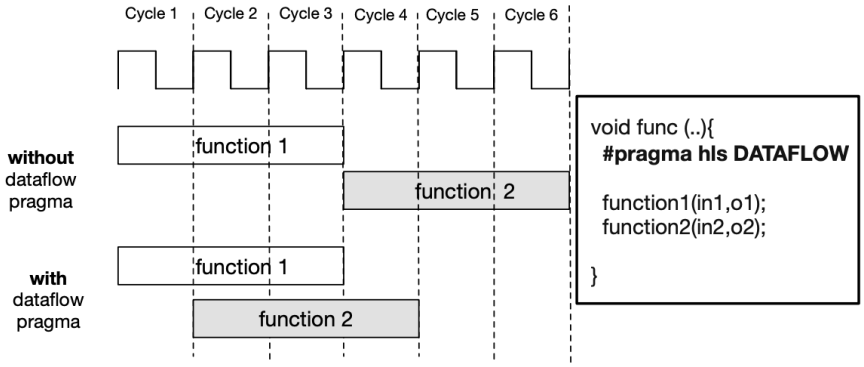


Figure 2.8: Task-level parallelism in HLS [2]

Task-level Parallelism. Parallelism in HLS can be defined at a higher level, such as the entire task. Dataflow pragma is used to define parallelism at the function level. A Dataflow pragma pipelines the functions and schedules them to start their operation as soon as the inputs are ready. Figure 2.8 presents the task-level pipelining. The overall latency to finish two functions is six cycles (three for function one and three for function 2). When they are pipelined, the second function can start its process after the first cycle. Thus, the overall execution time is reduced to 4 cycles with dataflow. In pipelining the tasks, the task with the highest latency determines the initiation interval.

Memory configuration. Memory partitioning is a critical part of HLS to achieve desired performance. Proper memory partitioning can allow a design to achieve the desired parallelism (II=1 for the initiation interval for loop pipelines or the maximum unrolling factor). The FPGA offers various on-chip memory resources such as block RAMs (BRAMs), LUT RAM, and Ultra RAM (URAM), and it is the designer's responsibility to choose optimal memory partitioning of these memory blocks for the design. Memory partitioning divides a single array of data into multiple ar-

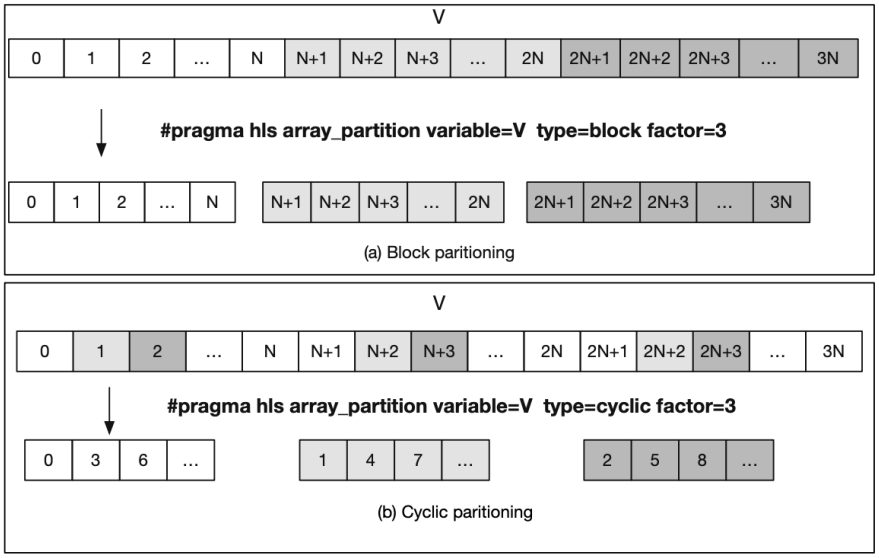


Figure 2.9: Array partitioning in HLS

rays and assigns each array to a different memory resource. Each memory module can be accessed independently. Figure 2.9(a) demonstrates a block partitioning method where each smaller array is created from consecutive blocks of the original array. Another way to partition an array is cyclic partitioning, which creates smaller arrays by interleaving elements from the original array (Figure 2.9(b)).

2.4 High Bandwidth Memory (HBM)

High bandwidth memory (HBM) is a new type of memory that vertically stacks memory chips, like floors in a skyscraper. Various layers of chips are stacked on top of each other using vertical channels called TSVs (through-silicon vias). This allows the HBM to reduce the distance that data needs to travel between the memory and processor and allows greater throughput per area than the conventional DDR memory. By reducing the amount of power needed to transfer data between memory and processor, HBM is also more power efficient than DDR. We use the HBM in this dissertation because of the highly parallelizable and memory-centric nature of graph processing. It

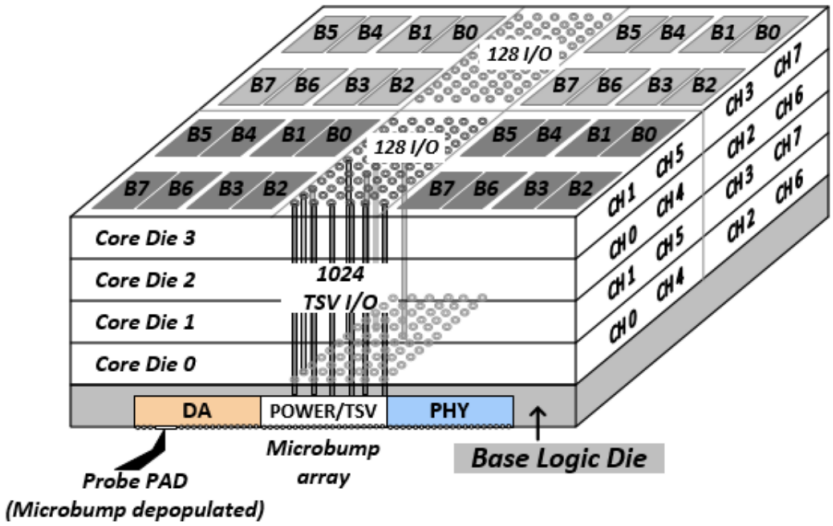


Figure 2.10: Overview of HBM on an FPGA [3]

should be noted that the HBM connected to an FPGA (e.g., the Xilinx Alveo U280 FPGA) would provide two different clock domains. One of these domains is for the FPGA and the other for handling memory reads and writes to the HBM. The highly parallelizable nature allows several concurrent operations to be executed across the many vertices of a graph. The memory-centric nature results in a lot of the processing time being spent on moving data between memory locations. As opposed to the 2D DRAM, HBM uses stacked RAM, which increases the height of the component, but not the width. This leads to significantly more memory capacity without the need for longer connections. While GDDR5 uses the FPGA/GPU chip on a silicon die and is surrounded by off-chip memory, HBM takes a different approach. HBM uses an FPGA placed on an interposer - or an electrical interface that routes connections between sockets - in addition to four pieces of stacked memory, each residing on top of its logic die. The design allows for stacking additional memory on top of each chip, thus reducing the need for longer connections, additional power, performance drops, or heating issues. HBM can allow for a theoretical stack of four times the chips of current RAM. This provides significant power gains in addition to faster and more efficient memory. Because the HBM modules are soldered close to the FPGA die physically, there is a benefit from short paths for data transmission. Combined with the wide memory bus, the FPGA can be fed

with information responsively (lower latency) while consuming considerably less power to achieve similar bandwidth than GDDR5 memory. The capability of stacking memory chips vertically and on the same substrate as the FPGA die allows manufacturers to save space on the board.

The fundamental structure of HBM is composed of a base logic die at the bottom and stacked core DRAM dies, which are interconnected by TSVs as shown in Fig. 2.10 [23]. The power and ground have common planes to support all eight channels. In the heterogeneous HBM structure, the core dies have a conventional DRAM architecture with TSV interfaces. The base die has I/O buffers and inevitable test logic. Using stacked DRAM, TSV, micro-bump, and 2.5D package technologies, HBM offers improved capacity, bandwidth, and power efficiency compared to conventional DRAMs.

Chapter 3

GraphTinker: High Performance Data Structure For Dynamic Graph Processing

In recent years, there has been a growing interest in frameworks for processing streaming graphs because many real-world graphs change in real-time (e.g., [[24], [5], [25], [26], [27], [28]]). These graph-streaming systems receive a stream of queries and a stream of updates (e.g., edge and vertex insertions and deletions, as well as edge weight updates). They must process both updates and queries with low latency in terms of query processing time and the time it takes for updates to be reflected in new queries.

3.1 Challenges

Several prior art frameworks do not satisfy the requirements for both high-throughput graph updating (i.e., edge insertions, deletions and updates) and high-throughput graph analytics (i.e., computing PageRank, BFS, etc.). One is sacrificed at the expense of the other. For example, graph data structures based on the adjacency list model [5] suffer from long probe distances when following edges, causing poor graph update throughputs. The adjacency list-based data structures consist of

vertex tables and edgelists that holds the edges incident to each vertex. During graph updates (i.e., edge insertions and deletions and updates), entire edgelists of a particular vertex sometimes need to be traversed in search for an edge, which causes a worst-case runtime complexity of n (where n is the number of edges inserted at the vertex). On the other hand, the adjacency list-based models enjoy high throughput when used for graph analytics because of their compact edge structure — i.e., all edges incident to a vertex are located in physically contiguous locations in DRAM and can therefore be streamed sequentially. Open address hashing is another classic representation used in modern data structures [29]. A classic example of open address hashing is the Robin Hood hashing (RHH) algorithm [30] [31] [32]. RHH works by moving keys around in a hashtable to reduce probe distance and yields a worst-case runtime complexity of $\mathcal{O}(\ln n)$ (where n is the number of edges inserted at a given source vertex). The RHH algorithm is based on the notion of probe sequence lengths or probe distance. The probe distance of a key is the number of probes required to find a key during a lookup. As new keys (i.e., edges) are inserted into the hashtable, old keys are shifted so that all keys stay reasonably close to the slot they originally hash to. The aim is to minimize the variance of a key's distance from its "home" slots. Open address hashing allows high throughput during graph updating as only a subset of the edgelist of a vertex is usually traversed before an insertion, deletion, or update is made. However, It can impede graph analytics performance due to its non-compact nature because an edge list of a vertex can consist of several unoccupied slots.

To tackle this challenge and achieve both high throughput during graph updating and graph analytics, we propose GraphTinker, a data structure for evolving (i.e., dynamic) graphs that integrates two novel data structures (Tinker and CAL), each with its unique advantage. We propose Tinker in section 3.2. Tinker allows low probe distance when following edges and supports high-throughput graph updating. We introduce Coarse Adjacency List (or CAL) in section 3.3. CAL allows sequential streaming of multiple edgelists together from DRAM during graph analytics. Finally, we tailor these two data structures together to form GraphTinker in section 3.4.

3.2 The Tinker representation

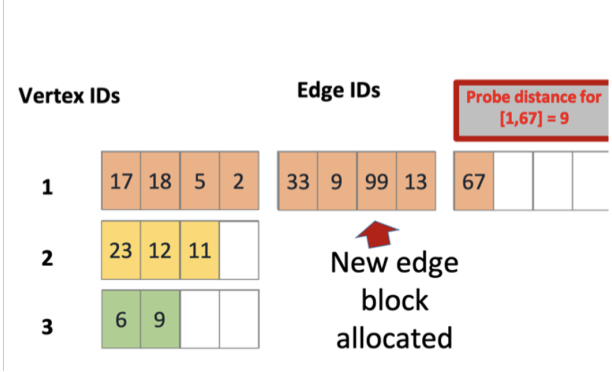


Figure 3.1: Prior art data structures based on adjacency list

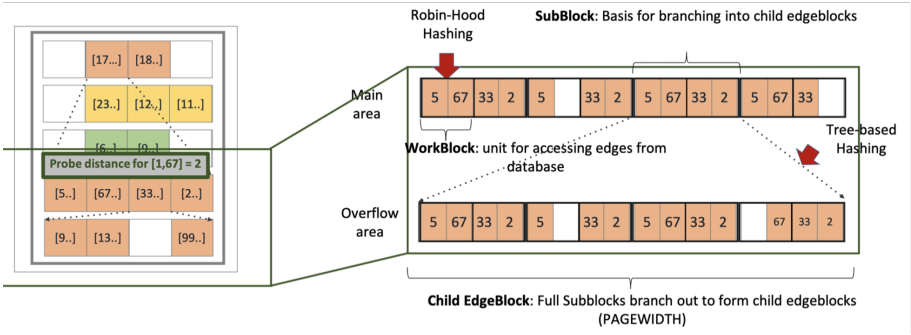


Figure 3.2: Tinker data structure

Our Tinker model also includes open address hashing, particularly the Robin Hood Hashing (RHH). However, the key advantage of Tinker over RHH is shown in Table 3.1. Tinker further reduces the probe distance when following edges (from $\mathcal{O}(\ln n)$ to $\mathcal{O}(\ln(\log_{P*} n))$) compared to RHH. It achieves this via a tree-like representation. By reducing the probe distance, Tinker achieves reduced number of visits to DRAM during edge updating. This in turn reduces the cost of high-latency DRAM access. Tinker combines the Robin Hood hashing (RHH) algorithm with a tree-based hashing technique and maintains a slower degradation in throughput as more edges are added to the datastructure compared to the adjacency list and RHH-based approaches. This allows better load stability. The RHH algorithm prevents Tinker from reading entire edgelists of a source vertex. The tree-based behavior further reduces the probe distance by completely skipping entire chunks of

Approach	Adj List	RHH	Tinker
Time complexity	n	$\mathcal{O}(\ln n)$	$\mathcal{O}(\ln(\log_{P*} n))$

Table 3.1: Comparing time complexities for Adjacency List, RHH and Tinker. P* is the tree fan-out (e.g., 4 from Figure 3.2)

edges when following edges in a logarithmic order. This allows Tinker achieve improved runtime complexity over RHH and the adjacency list model.

The Tinker data structure is divided into two parts, namely the main region and the overflow region. The main region is composed of edgeblocks (called top-parent edgeblocks), which store edges, and is indexed by the vertex source IDs. Therefore, every index of the main region in Tinker consists of edges belonging to a source vertex. The overflow region is also composed of edgeblocks which have the same property as the edgeblocks in the main area, but are descendants of Subblock sections of either the main region (1st generation descendant) or of the overflow region (¿1st generation descendant) itself. When the Subblock regions (of edgeblocks either in the main area or overflow area) become congested and filled with edges, they ”branch out” into child edgeblocks, which reside in the overflow area. This tree-like behavior allows Tinker to grow arbitrarily. Any child edgeblock has the same characteristics as its parent edgeblocks, and also consist Subblocks, which can also “branching out” when congested. Fig. 3.2 illustrates this “branching out” behavior with a simple example. As shown, the third Subblock belonging to vertex with ID one (v1) was at some point congested and branched out to form a new child edgeblock (*ov1*). As the graph grew, the second Subblock of *ov1* also became congested with edges, and in turn, branched out to form its own child edgeblock (*ov3*). With this descendant-level arrangement of edgeblocks, the average probe distance when following edges of a particular vertex vi is of the order $\mathcal{O}(\log_P n)$ (where P is the fan-out). Therefore, as the graph grows, Tinker experiences lesser performance degradation compared to the adjacency list based data structures. The tree-like expansion of Tinker does not only allow it grow arbitrarily, but also plays an important role of reducing the probe distance when following edges. By allowing a given source vertex have multiple branches to store its edges,

only relevant branches can be traversed when following edges. This allows entire branches to be skipped during graph updating. For example, source vertex 2 has twelve edge blocks that store its edges, but the tree-based strategy will ensure only six of its edgeblocks are explored with the RHH algorithm when inserting the edge $\langle 1, 13 \rangle$ (source ID of 1 and destination vertex ID of 13).

3.2.1 The Subblock region

The Subblock regions are components that make up edgeblocks in an EdgeblockArray. The Subblock region is the first layer of granularity of the edgeblock (in the EdgeblockArray). It represents the component of the EdgeblockArray which is capable of 'branching out' (when congested) into child edgeblocks (located in the overflow region) in order to house more out- edges for a particular source vertex. It simply achieves this by pointing to its child edgeblock which is located in the overflow region.

3.2.2 The Workblock region

The Workblock region, on the other hand, forms the second layer of granularity of the EdgeblockArray. Its main purpose is to parameterize the granularity at which edge data are retrieved from the EdgeblockArray for inspection. During the process of updating a new edge, when the Tree-Based Hashing scheme allocates a (or retrieves an existing) Subblock to be retrieved from the EdgeblockArray, the Subblock is retrieved one Workblock at a time for the RHH process. Therefore, having too large Workblock sizes would increase the probability of a successful completion of the RHH process in that retrieval, but at the same time would increase the number of edges retrieved from DRAM. Therefore, the concept of having Workblock size as a parameter during configuration of GraphTinker allows the user to select an optimum performance point.

3.2.3 The VertexPropertyArray

The VertexPropertyArray is the array structure which houses the properties of the vertices of the graph. It stores information pertaining to the vertices such as the degree, value and any flags associated with it and indexed by the vertex IDs of the vertices. This structure works with the EdgeblockArray during the graph computation process because both edge data and vertex properties need to be retrieved for computation.

3.3 The Coarse Adjacency List (CAL)

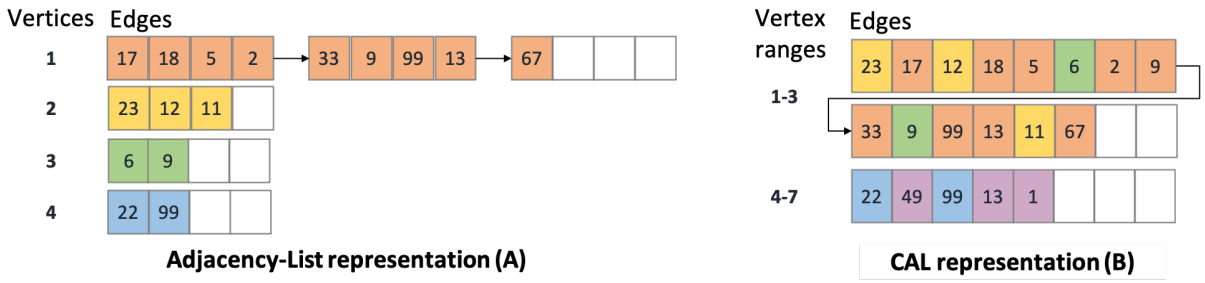


Figure 3.3: Comparing adjacency list (A) and CAL data structures (B)

The Coarse Adjacency List (CAL) allows GraphTinker support high throughput when running graph analytics. CAL and Tinker are connected to form GraphTinker (explained in section 3.4). CAL is linked to, and mirrors the content of Tinker. Therefore inserting, updating or deleting an edge in CAL happens when performing the same operation in Tinker. Every edge in Tinker consist a pointer that points to its corresponding mirror in CAL. Due to its highly compact nature (i.e., consisting no empty slots within edgelists), CAL can maximize DRAM bandwidth by reading groups of edgelists from DRAM. Similar to the adjacency list model, edges in CAL are located in physically contiguous locations in DRAM and can therefore be streamed sequentially. Unlike CAL, however, several source vertices are collected in groups (called vertex groups) and share an entry — i.e., all edges of a group of source vertices are located in physically contiguous locations in DRAM. This makes CAL more efficient than the adjacency list model when reading edgelists

of groups of vertices from DRAM. In CAL, edgelists belonging to a group of source vertices are stored in edgeblocks, and each edgeblock maintains a pointer that connects it to the previous and next block housing edges in the same group. By this, edgeblocks can be arbitrarily allocated in different physical locations in DRAM, and CAL can keep track of which edgeblock belongs to which vertex group. Figure 3.3 illustrates the CAL data structure. To form CAL, source vertices are partitioned into different groups according to their vertex ids, and each group represents a given contiguous range of source vertex IDs. For example, if every group consists of 1024 vertices, then source vertex ids from 0 to 1023 all belong to group 0, etc.

Whenever a new edge is inserted in Tinker, its source vertex id is inspected to find the group that vertex belongs to. The last assigned edgeblock to this group is retrieved and the edge is inserted into the last unoccupied edge slot in this edgeblock. If the edge previously existed in Tinker, its pointer to CAL is retrieved and its corresponding mirror edge in CAL is updated. When an edge is deleted in Tinker, the mirror edge is retrieved in CAL and also deleted (i.e., flagged as invalid). Because this process of updating the CAL does not involve traversing edges, the overhead of its insertion, deletion and updating operation is low. CAL provides a more compact edge data representation in database and therefore reduces the number of non-contiguous edge data accesses from memory during graph analytics computation. How CAL and Tinker are work together to process graphs is discussed in more details in section 3.7.

3.4 GraphTinker

GraphTinker tailors the Tinker and CAL representations together into a data structure as shown in Figure 3.4 to achieve high-throughput graph updating and graph analytics. For every edge inserted into the Tinker representation, its source vertex is inspected and a copy of the edge is also inserted into the next empty slot in CAL. GraphTinker is efficient at processing graph updates because Tinker allows reduced probe distance. To accelerate graph processing, these two data structures are coupled with a hybrid graph processing model. The key insight behind this hybrid model is

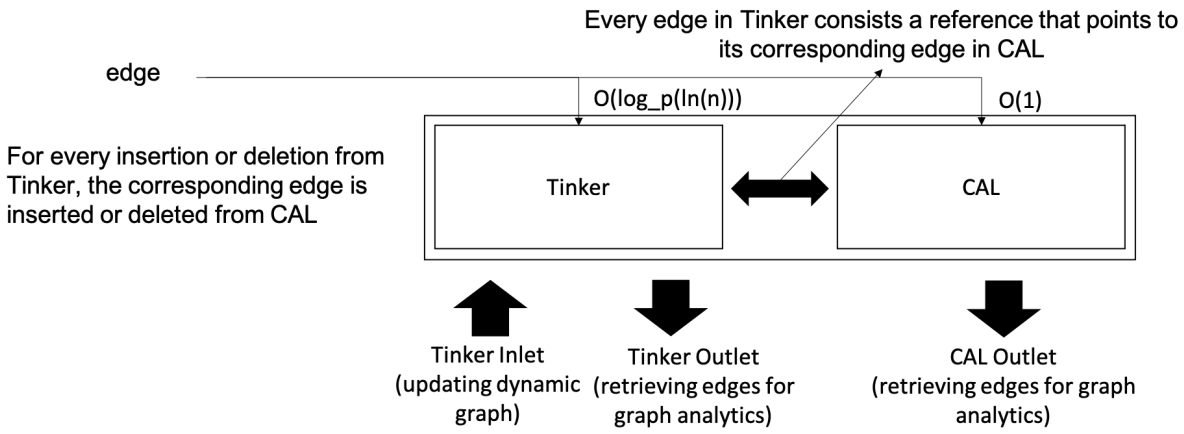


Figure 3.4: Block Diagram of GraphTinker showing Tinker and CAL representations

edges can be read from Tinker or CAL depending on the state of the graph iteration. More details of the hybrid model is discussed in section 3.7.

3.5 GraphTinker Operations

3.5.1 Inserting an edge

Two stages are involved when inserting an edge into GraphTinker: the FIND and the INSERT stage. The FIND stage searches for the edge belonging to the associated vertex in the Tinker, while the INSERT stage attempts to insert a new edge if the FIND stage is unsuccessful. In cases where deletions have previously been made and an empty slot is created, the INSERT stage insert the edge into the empty slots. For each stage, the edge is hashed by its source and destination vertex ids to determine the appropriate Subblock in Tinker to searching. Workblocks are retrieved from the Subblock in sequential order, running the find/RHH algorithm on each Workblock for the FIND or INSERT mode respectively. If the update is still unsuccessful after all the Workblocks of a Subblock are inspected, newer 'branches' (edgeblocks) are created (if not already available) out of the Subblock, rehashing is done again, and the same process continues in the newly-hashed child Subblock region. This process can continue for a few generations for very large-degree vertices.

"Branching out" from a Subblock to form child edgeblocks is simply achieved by inserting a pointer into the Subblock, pointing to the newly created child edgeblock.

3.5.2 Deleting an edge

GraphTinker supports two mechanisms for deleting an edge: delete-only and delete-and- compact. The delete-only mechanism is straightforward; instead of erasing all data (key, value and probe distance) of the element and moving succeeding elements forward, a flag (tombstone) is set to indicate that no edge exists in the bucket anymore. Therefore, any edge that traverses this bucket location the next time sees this bucket as vacant. The delete-and-compact mechanism, on the other hand, attempts to compact the data structure whenever a deletion is made, reducing probe distance and freeing edgeblocks for subsequent insertions and maintaining compactness of the database. All this happens during runtime whenever any edge is deleted. It achieves this by deleting edges (flagging as tombstone) from appropriate child edgeblocks of the data structure. These edges removed are then inserted into the slots where the deletions took place. By doing this, the holes in between edges in an edgeblock that are created by deletion of edges are filled up during the deletion process, thus allowing the data structure to stay compact even as more and more edges are deleted from the database. In order to avoid the significant overhead and complexity of edge tracking associated with the swapping process of the RHH algorithm, only the Tree-Based Hashing algorithm is enabled with this mechanism, with the RHH algorithm turned off.

3.5.3 Retrieving edges

GraphTinker provides two outputs for reading edges. The first connects to Tinker while the second connects to CAL. The choice of which output to use depends on the mode of the graph iteration. More details on these different modes are discussed in section 3.7.

3.6 Scatter-Gather Hashing

GraphTinker maintains a compact graph representation at every stage of a graph's growth life. This prevents the need for pre-processing to compact the graph structure before running graph analytics at any update step. Edge updates streaming into a data structure can be random and non-contiguous (by their source and destination vertex ids). A typical scenario is when the first batch of edge updates to the graph have source vertex IDs in distant physical locations in DRAM. When processing such sparse graph, many non-contiguous read accesses to DRAM will occur. This can result in significant performance degradation at the early stages of a graph's life, where not many edges have been loaded compared to the full capacity of the graph.

To tackle this issue, we introduce scatter-gather hashing when updating GraphTinker. The goal is to ensure the graph remains compact at every stage of its growth. For every edge to be inserted, the source vertex ID of that edge is inspected. If the source vertex ID has not been hashed before (i.e., a new edge), it is hashed by the *Scatter-Gather Hashing* function to obtain a new 'translated' source vertex ID associated with it. This is simply to obtain the the next unused source ID index location in Tinker (starting from zero). On the other hand, if the source vertex ID has been hashed before, the Scatter-Gather Hashing table is checked to obtain the formerly hashed id. In either of both situations, the edge is now associated with a new hashed source vertex ID before the update operation commences. The mapping between the original source vertex ID and the new hashed source vertex ID (and vice versa) is maintained by the Scatter-Gather Hashing table.

3.7 The Hybrid Graph Processing Model

3.7.1 Graph Processing Models for Evolving Graphs

There are two main processing models for running graph analytics on evolving graphs — the store-and-static-compute (or full-compute) model and the incremental-compute model. With the

store-and-static-compute model, graph updates (i.e., edge insertions, deletions and modifications) are made to the graph in discrete time intervals, and classic graph analytics algorithms are re-run on the entire graph after every update interval. Because the entire graph is processed at every update step, edges are sequentially read from DRAM, and the DRAM bandwidth can be utilized efficiently. However, this model suffers from redundant computations, as several vertices that have not changed from the previous iteration are processed. With the incremental-compute model, only regions affected by the batch update at discrete time intervals are processed. *Inconsistent vertices* are vertices whose properties change after a batch of edge updates are processed to the underlying graph. These vertices become the first set of active vertices during graph processing. The incremental-compute model reduces the number of edges and vertices that must be recomputed and can lead to significant performance improvement when this reduction is substantial. However, this model incurs more expensive, non-contiguous data accesses to DRAM because the updates subset of vertices can be located in physically distant locations in DRAM. In instances where many vertices are inconsistent after a batch update, this model can perform even worse that the store-and-static-compute model. The advantages of each graph processing models provide motivates us to propose a hybrid model that leverages the benefits of both.

3.7.2 The Hybrid Model

Our hybrid graph model selects which graph processing model (full-compute or incremental-compute) is best suited for every graph iteration. This model is best suited for algorithms such as BFS, SSSP, and CC, where not all vertices need to be active in every iteration. By deciding between full- and incremental-compute modes at every iteration, our hybrid model select the best execution path for the next iteration, and use the best suited data structure between Tinker and CAL. The Tinker data structure is employed to feed the graph engine with edges when processing in incremental-compute model, while CAL is employed when processing the full-compute mode. This is because CAL allows the edges of multiple contiguous vertices to be read sequentially at high throughput, but is inefficient when used to read edgelists of sparse, non-contiguous subsets

of vertices. In our hybrid model, statistic information is gathered during the apply phase of every graph iteration, and used to predict whether the next iteration should be processed in full- or incremental-compute mode. The heuristic formular is shown in Figure 3.5. The information gathered are the number of active vertices for the next iteration, the total degrees of all active vertices for the next iteration, the current size of the graph, and the maximum size attainable by the evolving graph.

$$mode(i+1) = \begin{cases} FP, & T > threshold \\ IP, & T < threshold \end{cases}$$

Where:

$$T = \frac{A}{E}$$

$$threshold = 0.02$$

FP: Full processing mode
IP: Incremental processing mode
A: Total number of active vertices for iteration $i+1$
E: Total number of edges loaded so far
mode(i): mode registered for iteration i

Figure 3.5: Heuristic Formula for the Hybrid Engine

The threshold value of 0.02 in the heuristic formula is gotten from running several experiments were run for both full and incremental processing modes to investigate the tradeoffs between sequential streaming versus random retrieval of edges from the graph structure, and how these vary with the number of edges retrieved. A typical scenario where our hybrid model is efficient is when a graph iteration (e.g., i) consist very few active vertices but its proceeding iteration (iteration $i+1$) consists a much large number of active vertices. The hybrid model can therefore use the incremental processing (IP) mode for iteration i and the full processing (FP) mode for iteration $i+1$.

3.7.3 Implementation

Fig. 3.6 shows the component-level arrangement of our hybrid engine model, and how the different parts are connected together to achieve its overall functionality. Inconsistency vertices are vertices

in the graph whose properties change because of the update. The *Set Inconsistency Vertices module* sets the initial inconsistency vertices after every batch update step is completed and before the updated graph is processed. The implementation of this module differs slightly depending on the algorithm to be implemented. For example, in the BFS algorithm, the vertices affected by the update batch comprise the source vertices of the edges in the update batch, while the inconsistency vertices when running Weakly Connected Component (CC) comprise both the source and destination vertices of the edges in the update batch. This unit is automatically generated depending on the algorithm to be run.

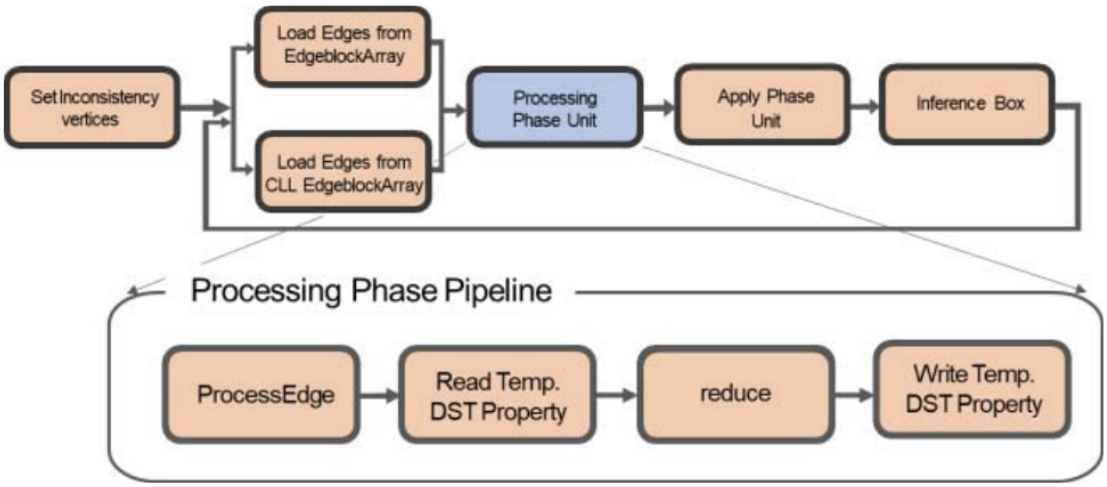


Figure 3.6: Hybrid Engine Implementation

The *LoadEdges* module loads edges from Tinker or CAL depending on the graph processing mode. When processing in full-compute mode, edges are loaded from the CAL because it provides a very compacted data representation of the graph edges. When processing in incremental-compute mode, edges are loaded from Tinker. The *Graph Processing pipeline* implements the modules responsible for the Processing and Apply phase explained in the Graphicionado paper [9]. The *Inference module* decides what the next execution mode should be for the next iteration, based on the collected statistic data and user-defined heuristics.

3.8 Evaluation

3.8.1 Target System

Implementation: We implemented GraphTinker on an Intel CPU (Intel® Xeon® E5-2620 v4) with 8 physical cores, operating at 2.10GHz, with 512GB DRAM. All experiments were run on the CPU. The PAGEWIDTH, Subblock and Workblock sizes of GraphTinker were chosen to be 64, 8 and 4 respectively because our experiments found that they define a good balance between effective data structure performance in updating edges and in graph analytics computation. We compare GraphTinker to the previous state-of-the-art data structure for dynamic graph processing, STINGER [6]. STINGER is a shared memory (in core) parallel dynamic graph processing framework. It performs updates about 3 times faster compared with 12 open-source graph databases and libraries [17], such as Boost Graph Library, DEX, Giraph and SQLite. We used version 15.10, available on GitHub. STINGER's configuration was set to have an average edgeblock size of 16. The batch size of edges used in the experiments to compare GraphTinker and STINGER is 1 million edges per batch. The choice on batch size does not have any impact on results.

Datasets: We evaluate the performance of GraphTinker (insertions and deletions) using a mix of both synthetic and real-world graph datasets. The synthetic datasets are generated from the Graph500 RMAT generator [2], while the real-world datasets are obtained from the University of Florida's Sparse Matrix Collection [8]. The properties of these datasets are shown in Table 1.

3.8.2 Algorithms

We evaluate the performance of GraphTinker as an efficient data structure for dynamic graph processing by evaluating it in conjunction with several algorithms: breadth-first-search (BFS), single-source- shortest-path (SSSP), and connected-components (CC). These algorithms were chosen because they could be modelled using both full and incremental processing modes, and because they are important algorithms in the graph community.

Dataset	# Vertices	# Edges	Type
RMAT_1M_10M	1,000,192	10,000,000	synthetic
RMAT_500K_8M	524,288	8,380,000	synthetic
RMAT_1M_16M	1,048,576	15,700,000	synthetic
RMAT_2M_32M	2,097,152	31,770,000	synthetic
Hollywood-2009	1,139,906	113,891,327	Real world
Kron_g500-logn21	2,097,153	182,082,942	Real world

Table 3.2: GRAPH DATASETS UNDER EVALUATION

3.8.3 Performance

GraphTinker vs. STINGER (Insertion throughput performance)

Fig. 3.7 shows the insertion throughput of GraphTinker and STINGER's data structures (i.e., without any graph analytics computation taking place) when used to insert edges into the graph using the Hollywood2009 dataset. Two different setups for GraphTinker were used: First, when the GraphTinker is used with the CAL feature and second, when used without it. A single thread was used to run the experiment. The y-axis in the figure represents the insertion throughput (in million edges per second) while the x-axis represents the input sizes of edges loaded (in million edges inserted in batches of 1 million edges per batch).

As shown in Fig. 3.7, GraphTinker's insertion throughput outperforms STINGER by up to 2.7X when GraphTinker is used with the CAL module and up to 3.3X when GraphTinker is used without the CAL module. One important observation from the plot is that GraphTinker shows less performance degradation than STINGER as the load (input size) increases. As shown, GraphTinker decreased from 1.6 million edges/sec in the fifth input batch to 1 million edges/sec in the last batch, giving about 34% throughput degradation, while STINGER decreased from 1.3 million edges/sec in the fifth input batch to 0.4 million edges/sec in the last input batch, giving about

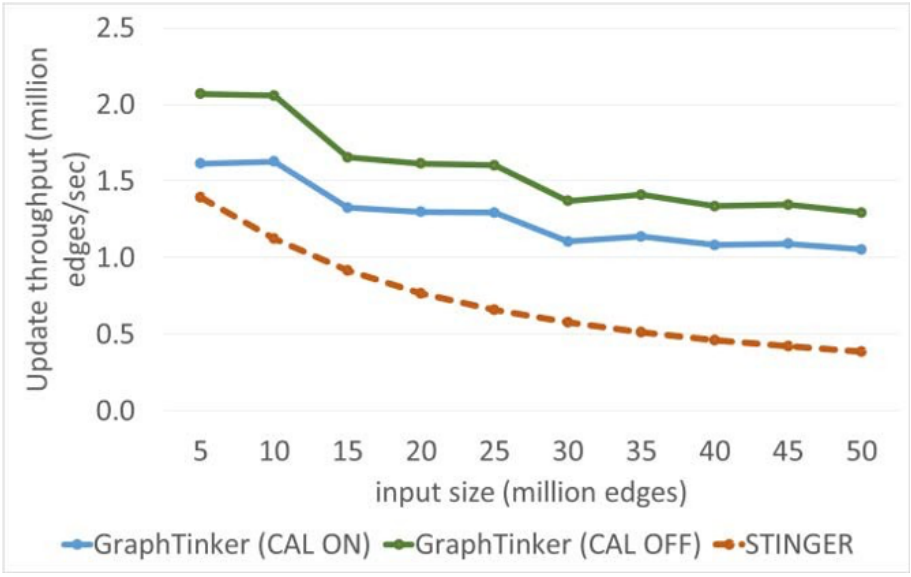


Figure 3.7: Insertion throughput for GraphTinker vs. STINGER with different input sizes and using the hollywood-2009 dataset

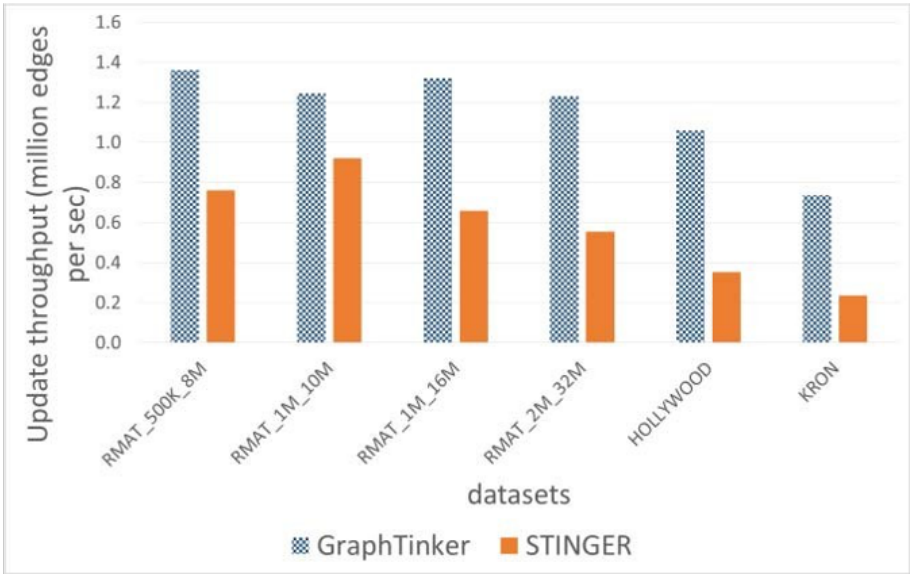


Figure 3.8: Insertion throughput for GraphTinker vs. STINGER on different datasets and with batch size of 1 million edges

72% throughput degradation. This implies that GraphTinker has better load stability compared to STINGER. The improvement in throughput and the higher load stability of GraphTinker over the STINGER structure is because GraphTinker makes fewer edge traversals during updating. This makes GraphTinker more efficient in DRAM accesses than STINGER.

Fig. 3.8, on the other hand shows the throughput performance of GraphTinker and STINGER for insertions using different datasets. As shown, GraphTinker outperforms STINGER across all the datasets. One interesting thing to note from the figure is that as the size of the datasets increases, GraphTinker's performance advantage also increases. This is again because STINGER has to do more edge-traversals than GraphTinker.

GraphTinker vs. STINGER (multicore performance)

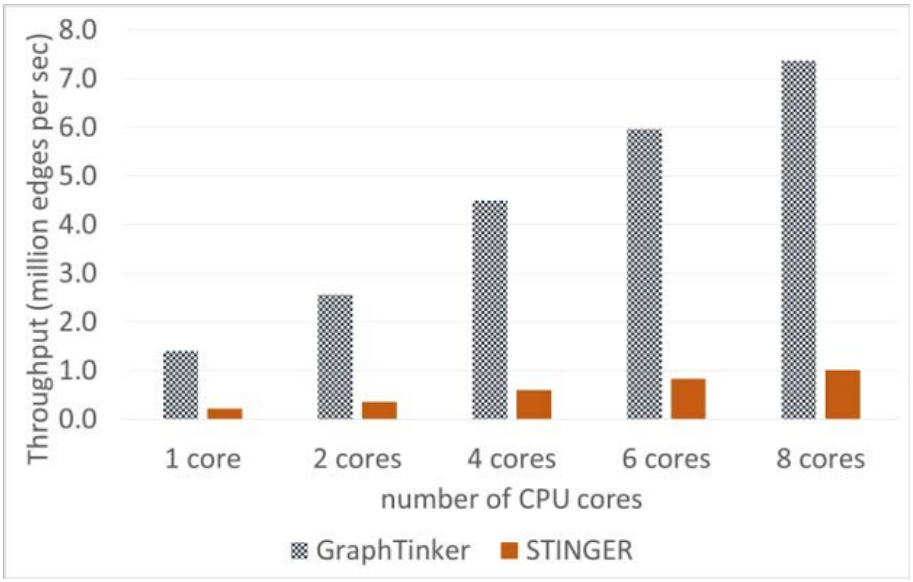


Figure 3.9: Update throughput for GraphTinker vs. STINGER using different number of CPU cores

Fig. 3.9 shows the performance of the GraphTinker data structure as the number of cores of the multicore CPU system increases. The y-axis in the figure represents the insertion throughput (in million edges/sec) of the data structure, while the x-axis shows the number of cores used. The

dataset used in this experiment is the hollywood2009 dataset. GraphTinker maintains its performance advantage as core count increases. As shown, GraphTinker outperforms STINGER in each case. In this experiment, we observe that, for each of the different number of cores used, STINGER starts off with fairly good insertion throughputs during the first set of batch insertions, but then experiences rapid deterioration as more batches are inserted. For example with 8 cores, STINGER experienced a decrease from about 3.4 million edges/sec in the first iteration to about 1 million edges/sec in the last iteration. In contrast, with GraphTinker, we observe far less degradation in throughput as subsequent batches are inserted.

GraphTinker vs. STINGER for Different Graph benchmarks

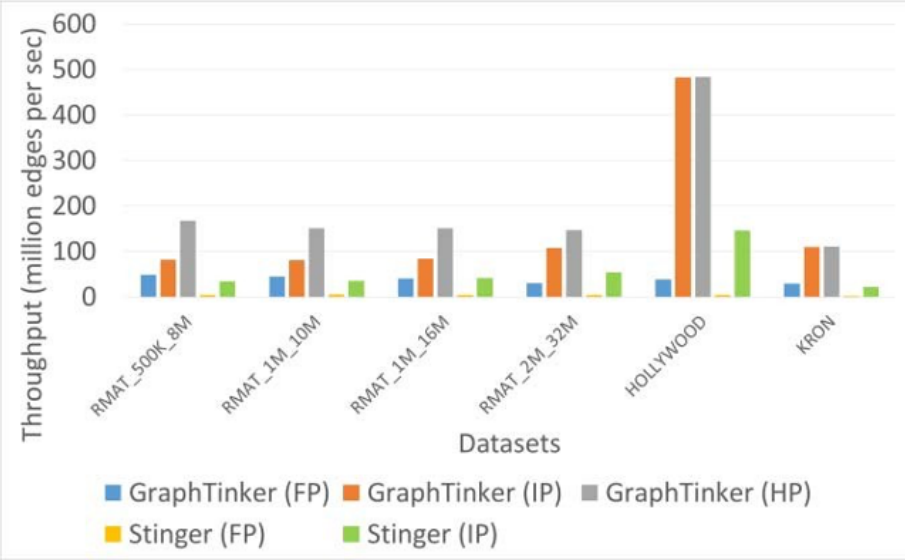


Figure 3.10: Processing throughput for GraphTinker vs. STINGER when running BFS on different datasets

In order to evaluate the impact on performance of GraphTinker on important graph algorithms, we ran graph analytics on BFS, SSSP, and CC using GraphTinker as the data structure and our hybrid engine as the graph engine. In each experiment, edges of the given dataset are loaded into the data structure in batches (of 1 million edges) to update the graph. After each batch insertion, the graph engine runs the given graph analytics algorithm on the current state of the graph. This

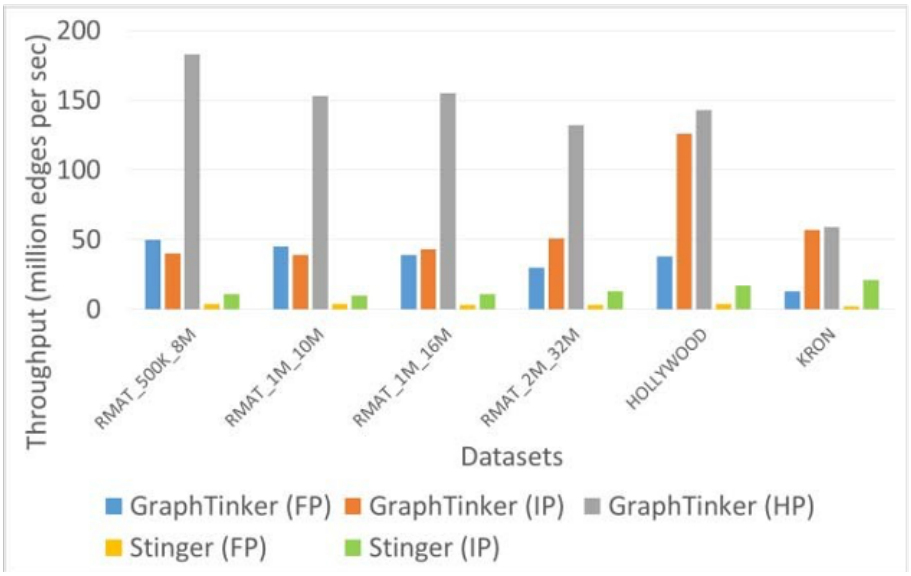


Figure 3.11: Processing throughput for GraphTinker vs. STINGER when running SSSP on different datasets

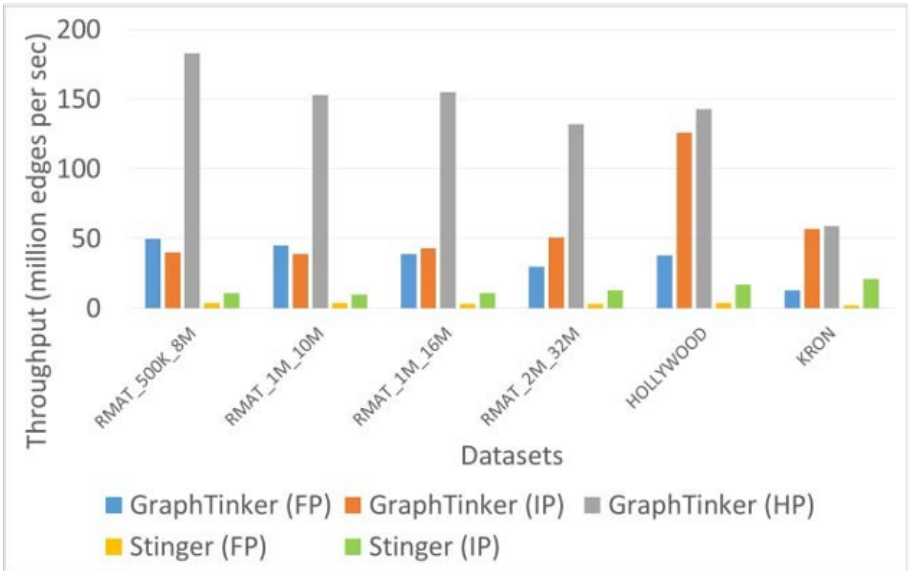


Figure 3.12: Processing throughput for GraphTinker vs. STINGER when running CC on different datasets

two-step process continues in turn until the final batch of edges is loaded and there are no more edges remaining to load. For comparison, we also ran graph analytics using STINGER.

Figs. 3.10, 3.11 and 3.12 show the performances of BFS, SSSP and CC algorithms when using GraphTinker and STINGER. The y-axes in the figure represent the throughput (in million edges/sec) when running each algorithm across the different datasets, while the x-axes represent the different datasets used in the experiments. As shown, when the hybrid engine is configured to run in full processing mode and using GraphTinker as the data structure, it demonstrates significantly better performance than when using STINGER (up to 10X performance improvement).

There are two major reasons for this. First, the Coarse Adjacency List (CAL) EdgeblockArray representation of edges maintained by GraphTinker allows a highly- compacted representation of edges, which reduces non- contiguous memory accesses compared to STINGER. Second, the Scatter-Gather Hashing (SGH) scheme implemented in GraphTinker allows it to reduce DRAM memory accesses during edge retrievals compared to STINGER. We conducted experiments where we disabled the CAL and the SGH features of GraphTinker and observed that GraphTinker then results in only about 1.5 times better than STINGER when running in full processing mode. Additional experiments show that the SGH and CAL feature account for a combined improvement of over 91% in GraphTinker's performance when enabled. This shows how significant these two features are to GraphTinker as an effective data structure for analytics of dynamic graphs.

These figures show that the hybrid mode always demonstrates better performance than the full and incremental processing modes for all the algorithms (BFS, SSSP and CC), and using all the datasets. This is because the inference box (the decision-maker) of the hybrid engine makes excellent predictions on the best execution path to take for every iteration. The reason for the significant gap in performance with the hybrid engine mode over both full and incremental processing modes in some cases (such as in CC) is that, in these instances, the hybrid engine makes especially successful predictions (we observed up to 97% correctness).

Figs. 3.10, 3.11 and 3.12 also show that there are instances where the incremental processing model can perform worse than the full processing model. An example is the CC experiment on

dataset RMAT_500K_8M. In this experiment, about 30% of the iterations involved a very large number of active vertices (¿ 100,000 active vertices). This caused IP to sometimes perform worse than FP, with ranges of 3X to 9X performance degradation in some iterations. This performance degradation is caused by the significant amount of non-contiguous memory accesses incurred by IP in these scenarios. However, in many applications and data sets, IP is superior. Hybrid execution is thus an important ingredient of an efficient graph engine.

Comparison of GraphTinker and STINGER edge deletion mechanisms

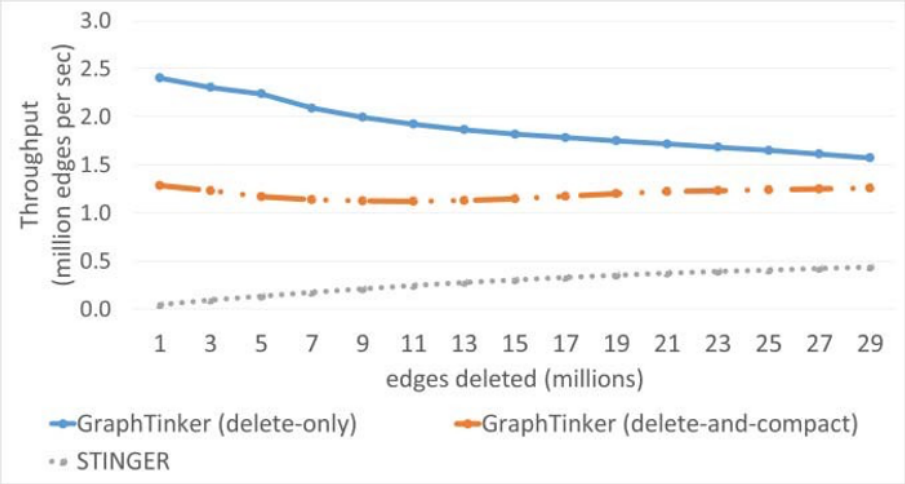


Figure 3.13: Edge deletions throughput for GraphTinker vs. STINGER data structure with different input sizes and using the RMAT_2M_32M dataset

We evaluate the impact of GraphTinker's deletion mechanisms on data structure and graph analytics performance. We run the BFS algorithm (in FP mode) on the RMAT_2M_32M dataset using a single core. The graph is initially fully loaded, after which deletions are then made (at 1 million edges per batch) in batches until the database is empty. This is to evaluate GraphTinker's performance as deletions are made to the data structure. Also, graph analytics is performed after every batch is deleted in order to evaluate GraphTinker's performance in analytics (i.e., after edge deletions are carried out). The performance of GraphTinker for edge deletions and also for analytics after edge deletions have been made is observed. For comparison, we compare to STINGER.

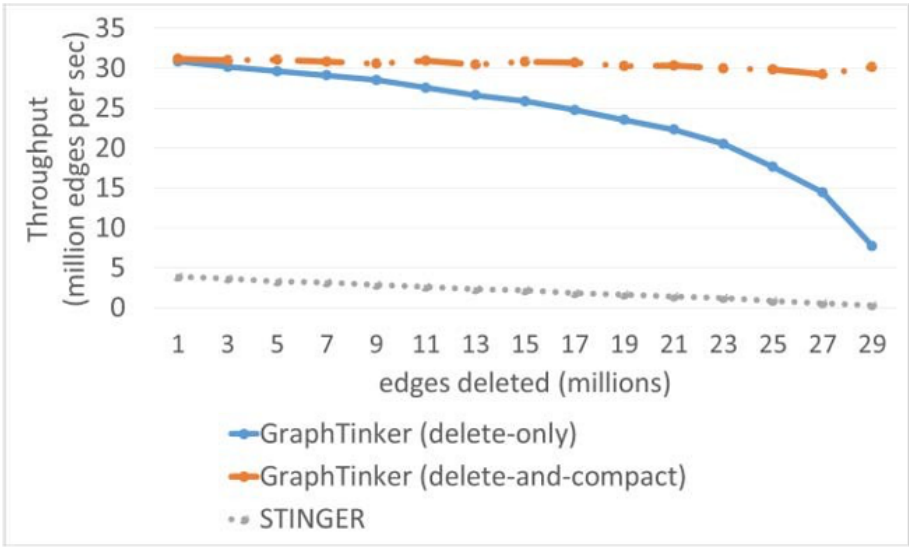


Figure 3.14: Throughput for GraphTinker vs. STINGER when running BFS on the RMAT_2M_32M dataset and with different number of edges deleted.

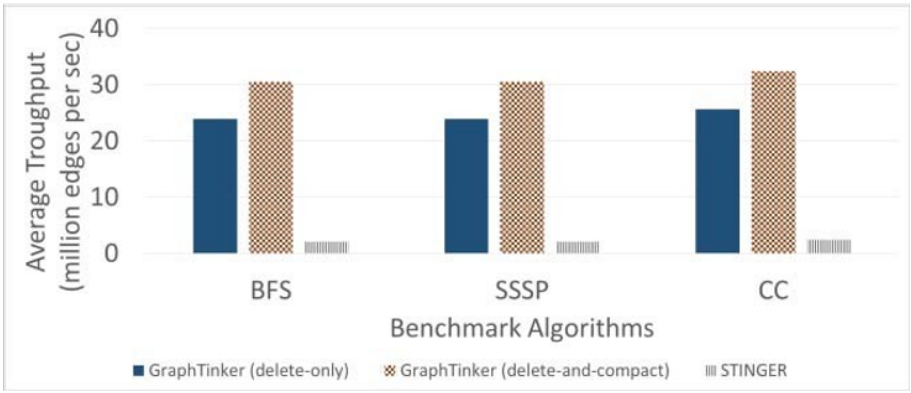


Figure 3.15: Average processing throughput for GraphTinker vs. STINGER when running BFS, SSSP and CC algorithm on the RMAT_2M_32M dataset and performing edge deletions

Fig. 3.13 shows the throughput of both data structures (GraphTinker and STINGER) from the experiment when used for edge updates (deletions) only and without any graph analytics process taking place. The y-axis in the figure represents the deletion throughput (in million edges per second) while the x-axis represents the amount of edges deleted (in million edges). As shown, the delete-only mechanism for GraphTinker outperforms the delete-and- compact mechanism by up to 2X when the first batch is deleted and only about 1.2X when the last batch is deleted from the database. Both deletion mechanisms, however, outperform STINGER's deletion mechanism. Another important observation is that GraphTinker's delete-only mechanism causes a throughput degradation as more edges are deleted from the database, whereas the delete-and- compact-in mechanism shows stable performance. This is because, with the delete-only mechanism, there is no shrinking of the data structure and so the same time has to be spent following edges, even though the number of edges in the database decreases. Whereas with the delete-and- compact mechanism, the data structure shrinks as more edges are deleted, allowing less time spent in following edges and more stable throughput.

Fig. 3.14 show the effect of the deletion mechanisms on analytics when both data structures (GraphTinker and STINGER) are used for graph analytics. The y-axis represents throughput (in millions edges/sec) of graph analytics, while the x-axis represents the amount of edges deleted (in millions of edges). As shown, the delete-and-compact mechanism outperforms the delete-only mechanism about 1.2X when half of the edges are deleted, to as much as 4X when the last batch is deleted. Both mechanisms also surpass STINGER's deletion mechanism. Another important observation is that the delete-only mechanism experiences degradation in throughput (from 30 million edges per sec in first batch to 7 million edges per sec on last batch) while the delete-and-compact mechanism experiences stable performance. The reason is similar to what was described earlier: with the delete-only mechanism, the data structure does not experience any shrinking and so the time spent retrieving edges from the database remains the same even as more edges are deleted.

Fig 3.15 shows the performances of BFS, SSSP, and CC algorithm running across the RMAT_2M_32M dataset. The y-axes in the figure represent the average throughput (in million edges/sec). The fig-

ure shows that the delete-and- compact mechanism demonstrates better performance, compared with the delete-only mechanism for all three algorithms.

Effect of different PAGEWIDTH sizes on GraphTinker's performance.

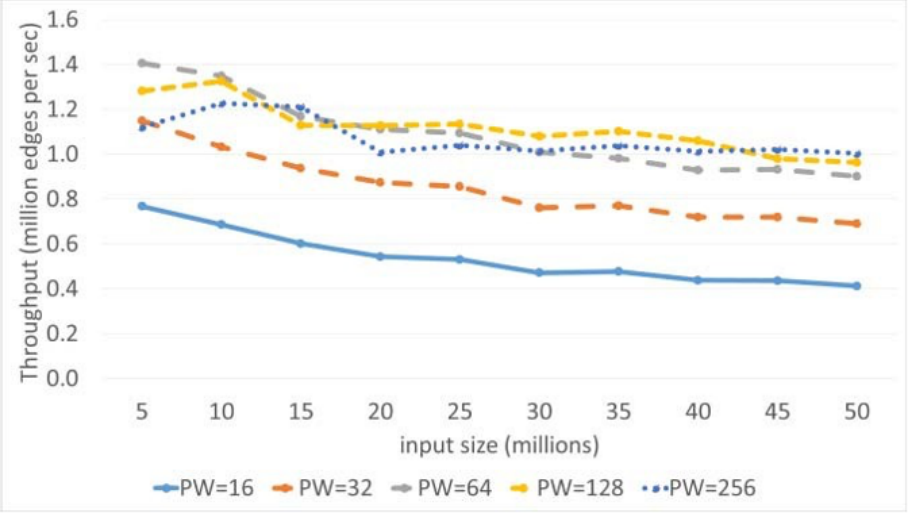


Figure 3.16: Effect of different PAGEWIDTH sizes on insertion throughput to GraphTinker data structure when loading the Hollywood2009 dataset.

In order to evaluate the effect of different PAGEWIDTH sizes on GraphTinker's data structure and analytics performance, we configure GraphTinker on different PAGEWIDTH sizes (16, 32, 64, 128 and 256) and run the BFS algorithm on the Hollywood2009 dataset. This dataset was chosen arbitrarily.

Fig. 3.16 shows the performance of GraphTinker's data structure with the five different PAGEWIDTH sizes. The y- axis represents the insertion throughput while the x-axis represents the input sizes of edges loaded. Fig. 3.17, on the other hand, shows the corresponding BFS performance with the five different PAGEWDTH sizes when the graph engine was configured to run on incremental processing (IP) mode. This mode is selected because it utilizes the EdgeblockArray.

Fig 3.16 highlights two interesting behaviors. First, an increase in the PAGEWIDTH size from 16 to 256 demonstrates an increase in the insertion throughput experienced by the data structure.

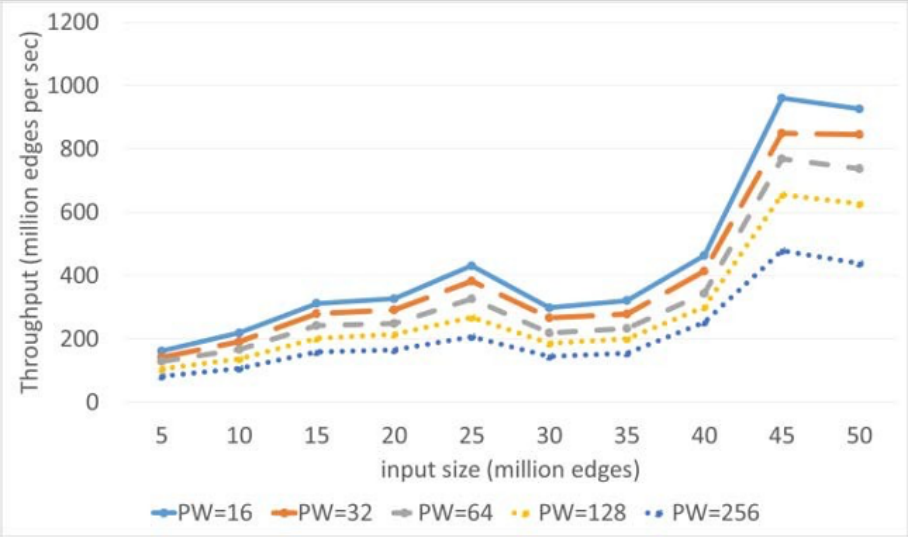


Figure 3.17: Effect of different PAGEWIDTHs on graph analytics throughput when running the BFS algorithm on the Hollywood2009 dataset.

This is because increasing PAGEWIDTH size increases the hash range of edgeblocks in the data structure, leading to reduced frequency of collision experienced by the RHH algorithm. Second, increased PAGEWIDTH size allows the data structure to experience lesser degradation of throughput as more edges are inserted (i.e., better throughput stability), with PAGEWIDTH size of 256 experiencing the highest throughput stability.

On the other hand, Fig 3.17 shows that increasing the PAGEWIDTH decreases the throughput of graph analytics and vice versa. This is because smaller PAGEWIDTH sizes give a more compacted data structure compared to larger PAGEWIDTH sizes, and this allows more edges to be retrieved per unit time during graph analytics.

Choice of optimal PAGEWIDTH.

In order to find the most optimal PAGEWIDTH size for GraphTinker, we designed an experiment which investigates the performance with varying ratios of updates to analytics. This experiment projects two use cases of a dynamic graph: (1) where edge updates are made frequently and analyt-

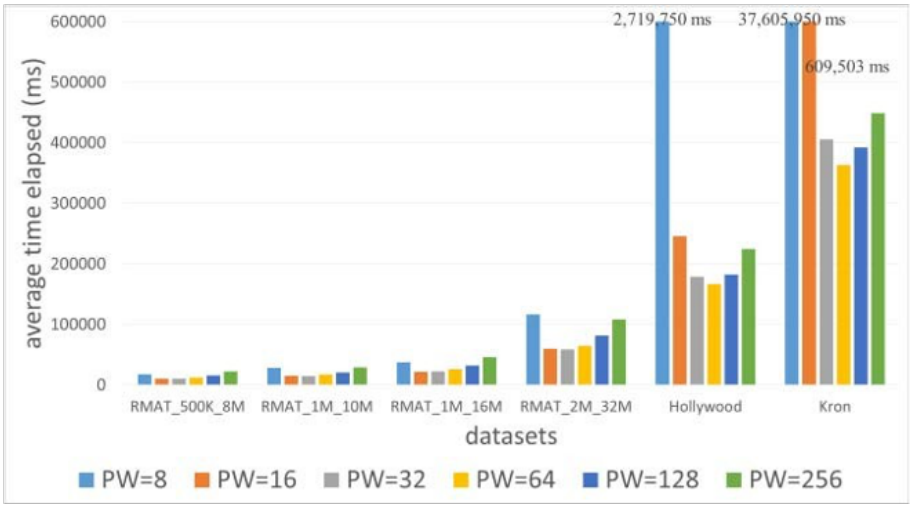


Figure 3.18: Behaviors of different PAGEWIDTHs in a combination of updates and analytics using different datasets when running the BFS algorithm. Bars are averaged across updates/analytics ratios.

ics rarely, and (2) where edge updates are made rarely and analytics frequently. The algorithm for analytics used for this experiment is the BFS algorithm. The ratios of updates to analytics in each of these experiments range from 1:10 to 10:1 and the PAGEWIDTHs used range from 8 to 256. For each dataset, 20 vertices among those with the highest degrees are pre-collected so that each analytic in each experiment uses a different root vertex. In each experiment comprising a dataset, PAGEWIDTH and updates/analytics ratio, edges are inserted into GraphTinker at batch sizes of 1 million edges per batch, and analytics are done in different intervals according to the update/-analytics ratio of that experiment. The left part of the ratio (updates) determines how frequently the insertion process is intercepted in order to run graph analytics, while the right part of the ratio (analytics) determines how many analytics should be run on each interception. For example, with a ratio of 4:7 on the RMAT_2M_32M dataset which contains 32 million edges, the edge insertion process to the graph is intercepted 4 times (i.e., after every 6 batches are inserted) in order to run 7 different analytics in each interception, each on a different choice of root vertex. After each experiment is conducted (360 experiments in total), we then calculate the average time lapses of each experiment using a given dataset and a given PAGEWIDTH (36 different data points). The

result is then plotted as shown in figure 3.18.

The y-axis of Fig 3.18 represents the time elapsed (in milliseconds) averaged across the updates/analytics ratios of each dataset and each PAGEWIDTH combination, while the x-axis represents the different datasets used in the experiments. As shown, the PAGEWIDTH size of 64 demonstrates the best overall performance, especially when dealing with larger datasets. Lower PAGEWIDTH sizes such as 8 exhibit poor performance due to very low edge- update performance, which becomes more pronounced with larger datasets. Although higher PAGEWIDTH sizes exhibit good edge update performance, they ultimately experience poor analytics performance due to their less compacted arrangement of graph edges. The PAGEWIDTH size of 64 appears to represent a good balance of update/analytics ratio. These experiments were conducted with the BFS algorithm. Nevertheless, we expect this behavior to generalize to other GAS based algorithms using the edge centric model because the experiment illustrates the memory access behavior of the BFS algorithm, which is the same model as any other GAS based analytics algorithm.

3.9 Related Work

Over the years, a number of data structures have been proposed for graph processing. There has been a tradeoff between the effectiveness of these data structures while updating edges versus supporting efficient real-time graph analytics. This is because more compacted graph data representations, which result in higher throughput performance, typically require large data movements when performing updates. We explore prior data structures for dynamic graphs in this section.

3.9.1 Adjacency matrix

A classic data structure is an adjacency matrix, which holds a 2D matrix representation of the edges of the graph, so that an edge with endpoints u_i and v_i is located in the matrix position a_{ij} of the adjacency matrix structure. Even though this model provides O(1) edge insertion time, it is

unsuitable for dynamic graph processing, because the overall sizes of today's graphs would warrant huge memory consumption $\mathcal{O}(n^2)$ as well as very sparse representation of graph data.

3.9.2 Adjacency list

STINGER [5] is a state-of-the-art data structure for dynamic graph processing. It is a shared-memory data structure based on adjacency lists. STINGER's model consists of a vertex table and an edge list. Each element of the vertex table (called Logical Vertex Array) points to a given location in the edge list (Edge Block Array). The vertex table holds the vertices in the graph while the edge list consists of edgeblocks, which hold the edges associated with each vertex. Edgeblocks can point to other edgeblocks to accommodate more space for edges belonging to a vertex. While this model allows some level of compaction of edges - meaning the edges in an edge block are packed close together - it still suffers from long probe distance during graph updates because entire chains of edgeblocks (belonging to a particular vertex) have to be traversed during an edge insertion or deletion process. This is because the edges belonging to the vertices are not sorted or hashed in any way and could be located in any of the edgeblocks belonging to a particular vertex. Additionally, this representation does not yield a highly compacted representation of edges in data structure because of the many non-contiguous edgeblocks that could be present in the database. These drawbacks direct our approach.

3.9.3 Robin Hood Hashing

The Robin Hood Hashing (RHH) algorithm [3-5] provides a solution for achieving low probe distance. A brief description of how the algorithm works is discussed below. Fig 3.19 shows the insertion process of an edge into a simple hash table using the Robin Hood Hashing algorithm. Assume i, j is an edge between a vertex i and j. If this edge is to be inserted into the hash table, a suitable hash function (e.g., h = Vid mod C, where Vid is ID of the edge's source vertex and C is the capacity of the hash table) is used to compute the initial position (known as initial bucket) of the

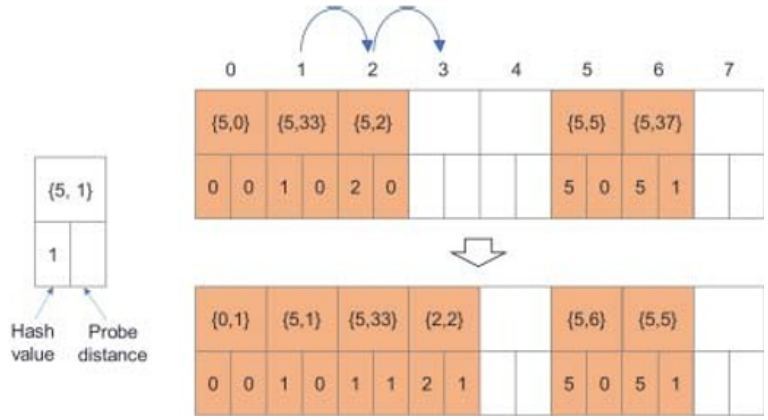


Figure 3.19: Robin Hood Hashing

hash table from where inspection begins. For the example in Fig. 1, the initial bucket for the edge x is 1. Next, because the edge entry in bucket 1 of the hash table is already occupied, the probe distance of both edges are compared to decide who now stays in the bucket. If the probe distance of the edge currently present in the bucket is lower than that of the edge to be inserted, then the edge to be inserted now swaps the edge currently present in the bucket, so that the edge formerly in the bucket now becomes the new floating edge looking for another location to be inserted. Otherwise, the edge to be inserted skips over that bucket to inspect the bucket. Probe distance, in the context of Robin Hood Hashing, refers to the distance between the original hashed positions of the edge to its current displaced position. For the example, the new edge (1, 5) wins because its probe distance is equal to the probe distance of the edge already present in bucket 1 (1, 6). Since edge (1, 6) is evicted, it must find a new bucket to occupy. When comparing with edge (2, 2), it wins because it is "poorer" than the one in bucket 2 (2, 2) – meaning its probe distance is greater than the one in bucket 2 (2, 2). Edge 2, now evicted, finds bucket 3 empty and moves there.

3.9.4 Graph Processing Models for Dynamic Graphs

Two primary execution models exist for updating analyses of a graph as the contents change.

Store-and-static-compute model: Traditional works [13- 15] proposed graph processing on dy-

namic graphs by employing the store-and-static-compute model, in which updates are made to the graph in discrete time intervals and classic, static graph analytics algorithms are re-run on the entire graph after every update interval. If preprocessing is carried out before using this model to provide a compact representation of the graph data (e.g., from adjacency list to CSR representation), it can offer advantages because retrieving of edges can now be achieved in a highly contiguous manner. However, this model suffers from having to perform redundant computations, which can be significant when dealing with large graphs.

Incremental-compute model: Later works [20-24] explore an incremental approach to avoid this redundant computation. In this model, only regions affected by the batch update at discrete time intervals are processed. The vertices affected by the batch updates are referred to as inconsistent vertices, and they are identified as vertices in the graph whose properties change because of the update. These vertices become the first set of active vertices during graph processing. The advantage of this model is that it reduces the number of edges and vertices that must be recomputed and can lead to significant performance improvement when the reduction is substantial. However, this model incurs more expensive, non-contiguous data accesses to memory because the set of active vertices for a given iteration can be non-sequential and unsorted. In instances where many vertices are to be processed in a given iteration, the outcome using this model can be worse than the store-and-static-compute model.

The advantages each of these two graph computation models provide motivated us to create a hybrid model that leverages the benefits of both.

3.10 Conclusion

This chapter presents GraphTinker, a high-performance data structure for dynamic graph processing, as well as a new hybrid graph engine to improve efficiency when processing dynamic graphs. Technical advances offered by GraphTinker include: (1) a novel data structure that combines the benefits of two well-known hashing schemes to reduce the probe distance while following edges,

and hence provides better solutions to the state-of-the-art data structure models based on adjacency lists, and (2) ScatterGather and Coarse Adjacency List (CAL) features that allow efficient compaction to our data structure to minimize DRAM accesses. The hybrid graph engine offers an improvement over the store-and-static-compute model and the incremental-compute model proposed in previous works. This is achieved by automatically selecting the best execution path between these two modes for every iteration in order to combine the advantages offered by both models. Evaluation of GraphTinker on a variety of datasets and graph algorithms demonstrates that GraphTinker is capable of providing up to 3.3X performance improvement in update throughput on the CPU over the previous state of the art, STINGER. When used to run algorithms such as BFS, SSSP and CC on dynamic graphs, GraphTinker's more efficient data structure enables up to 10X improvement in performance compared to STINGER when running analytics in full processing (FP) mode. Finally, experimental tests using our hybrid graph engine demonstrate its capability to provide up to 2X performance improvement over the incremental processing model and up to 3X performance improvement over the full processing model.

Chapter 4

ACTS: Scalable Graph Processing on HBM-enabled FPGAs

4.1 Introduction

Despite the high off-chip bandwidth and on-chip parallelism offered by today's near-memory accelerators, software-based (CPU and GPU) graph processing frameworks still suffer performance degradation from under-utilization of available memory bandwidth because graph traversal often exhibits poor locality. Emerging FPGA-based graph accelerators tackle this challenge by designing specialized graph processing pipelines and application-specific memory subsystems to maximize bandwidth utilization and efficiently utilize high-speed on-chip memory. To use the limited on-chip (BRAM) memory effectively while handling larger graph sizes, several FPGA-based solutions resort to some form of graph slicing or partitioning during pre-processing to stage vertex property data into the UltraRAM. While this has demonstrated performance superiority for small graphs, this approach breaks down with larger graph sizes. For example, GraphLily [33], a recent high-performance FPGA-based graph accelerator, experiences up to 11X performance degradation between graphs having 3M vertices and 28M vertices. This makes prior FPGA approaches impractical for large graphs. We propose ACTS, an HBM-enabled FPGA graph accelerator, to address

this problem. Rather than partitioning the graph offline to improve spatial locality, we partition vertex-update messages (based on destination vertex IDs) generated online after active edges have been processed. This optimizes read bandwidth even as the graph size scales. The detailed model describing the various features in ACTS and its implementation on the HBM-enabled is discussed in this chapter.

4.2 The Challenge

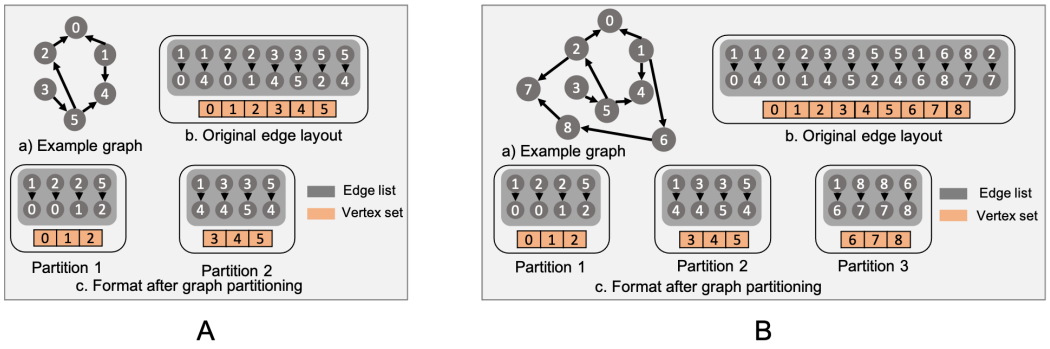


Figure 4.1: The challenge associated with graph slicing during pre-processing

A notable limitation with several single-FPGA graph accelerators is poor scaling with respect to increasing graph sizes. They experience degradation in throughput that worsens as the size (number of vertices) of the graph workload increases. Hence, the throughput when processing large graphs (tens of millions to hundreds of millions of vertices) can be far lower than the same when processing small graphs (thousands to millions of vertices). This problem stems from their strategy of slicing the graph into chunks to leverage fast on-chip URAM memory for random accesses to vertex properties. Because the FPGA's on-chip memory (BRAMs, URAMs, scratchpads etc.) are much smaller compared to their off-chip memory (DDR, HBM, SSD etc.), many slices can be created which leads to redundancies when reading source vertex properties. For example, the HBM of Xiliinx Alveo U280 is 128x larger than its URAM. The URAM, however, can be several times (10x) faster than HBM memory. The slower speed of HBM relative to URAM comes from

the need to be dynamically refreshed all of the time so it does not lose its data. This continuous refreshing can be time consuming and slow down the memory, especially when random accesses is involved. To leverage this high-speed potential of the URAM, several accelerators [33] [17] [34] [35] [18] [36] [37] [38] [39] [40] [41] [42] [43] resort to using the URAMs of the FPGA to house vertex properties of the graph, and the HBM/DDR to house edge properties. This is because several graphs have much smaller number of vertices that edges. Housing vertex properties in URAM allows random accesses to be made to fast URAMs, which avoids expensive random accesses to the HBM. URAMs are preferred to BRAMs to house vertex properties because of their larger capacity. Several accelerators employ this strategy. To handle large graphs with vertices more than the URAMs can accommodate, they resort to slicing such that the vertex properties of each slice can fit in URAM. This is a pre-processing step. During processing, the graph is then processed slice by slice. When processing each slice, destination vertex properties are buffered in URAMs first. Their corresponding edges and source vertex properties are then streamed from HBM, vertex updates are generated, and used to reduce their corresponding destination vertex properties in URAM. Because locality of each slice has been restructured around destination vertex properties, random accesses to destination vertex properties can be made to the URAM, which is much faster than if they were made to the HBM.

The fundamental challenge with slicing a graph to leverage high-speed URAMs as explained is that, because graphs are unstructured data, partitioning a graph's edges and vertices across a given dimension (e.g., destination ID) improves the URAM locality across that dimension, but degrades the same across the other dimension (i.e., source ID). No slice will therefore be completely separate from another as slices would be linked by edges that span across slices. Processing a slice will therefore depend on source vertex properties assigned to other slices. Buffering vertex properties across the sliced dimension (destination vertex ID) during processing will enjoy optimal DRAM read bandwidth usage due to its high locality. Buffering vertex properties of the other dimension (i.e., source vertex ID) however will suffer from excessive DRAM read bandwidth usage from reading unused vertex properties. For example, GraphLily [33], a recent state-of-the-art FPGA accelerator experiences up to 2X throughput advantage over Gunrock, a well known state-of-the-art

GPU accelerator for graph sizes under eight million vertices, but up to 4X throughput degradation for graphs between sixteen and sixty four million vertices.

Figure 4.1 illustrates this problem using a simple example. Figure 4.1A and B shows two sample graphs, with graph B larger than graph A. Assuming the available BRAM can only house three vertex properties. Graph A and B will therefore be considered too large and will need to be sliced to restructure locality. Graph A consists 6 vertices and will need to be sliced in two parts, while Graph B consisting 12 vertices will need to be sliced in three parts. When processing Partition 1 (of graph A), destination vertex properties (0, 1 and 2) will be read from DRAM to on-chip BRAM. The source properties and edges will then be streamed from DRAM in chunks to BRAMs when generating vertex updates. To maximize DRAM bandwidth utilization when reading source vertex properties from DRAM, the source vertices in each chunk is read from contiguous DRAM memory locations. For example, vertices 1, 2, 3, 4 and 5 are all read when processing Partition 1, even though only vertices 1, 2 and 5 will be used. These unused source vertex properties (3 and 4) account for DRAM bandwidth over-utilization. Additionally, vertex 1's source property is required when processing both Partition 1 and 2, which means that it will be read more than once. Therefore, reading destination vertex properties to URAM will enjoy good use of DRAM bandwidth, because every destination vertex property will be read and written to DRAM only once in each GAS iteration, but reading the source vertex properties will suffer DRAM bandwidth over-utilization from its degraded locality. This problem amplifies with the size of the graph because more degradation of source vertex locality will be experienced. With graph B for example, vertices 1, 2, 3, 4, 5, 6, 7 and 8 will all have to be read when processing Partition 1 (an additional three vertices read), even though only vertices 1, 2 and 8 will be used. Also, vertex 1's source property is required when processing Partitions 1, 2 and 3 (one time more than in 4.1A).

To combat this poor scaling issue we propose ACTS, a graph processing framework for static graphs on the HBM-enabled single-FPGA environment. ACTS employs the push-based GAS (Gather Apply Scatter), edge-centric style of processing. ACTS is designed to run on a single FPGA.

4.3 Why FPGAs?

We analyze the fundamental primitives in graph processing to justify our decision for using the FPGA for our architecture, as opposed to the CPU and the GPU. The first primitive is the processing of edges to generate vertex updates. This involves reading the edge and its corresponding source vertex property, and using the edge_function of the algorithm to generate a vertex update message. This simple operation needs to be repeated across all active vertices of the graph, and is therefore amenable to parallelization across the multiple hardware threads of the FPGA, the multiple CUDA threads of the GPU, and the multiple threads of the CPU. This operation is therefore amenable to all three architectures — the FPGA, the GPU and the CPU.

Another fundamental operation in graph analytics is updating corresponding destination vertices with the vertex updates. This operation is a central overhead in graph processing, and a motivation for a lot of research in graph processing. It is characterized by random memory accesses because the destination vertex properties can be located in arbitrary and sometimes very distant locations in DRAM. It therefore exhibits poor memory locality. This operation would benefit from a hardware architecture having both disaggregated on-chip memory blocks (scratchpad, URAM etc.), each of considerably large size, that can be connected together. The on-chip memory requirement is so that the random access behavior can be mapped on-chip, and random accesses can be made across fast SRAM rather than slower DRAM. The large size requirement is because of the unstructured nature of graphs forcing random accesses. The disaggregated nature is to allow parallelism, where several sections of the graph can be updating simultaneously in parallel. The requirement for connectedness is to that data can routed to destinations that lie outside the range of the on-chip memory. This is because the capacity of on-chip memory is certainly limited. Of the three architetcures, only the FPGA can satisfy all three architectural requirements. Though the CPU contain large SRAM memory (L2, L3 caches in CPU and shared memory in GPU), their non-disaggregated nature prevents exploiting parallelism.

4.4 Implementation Details

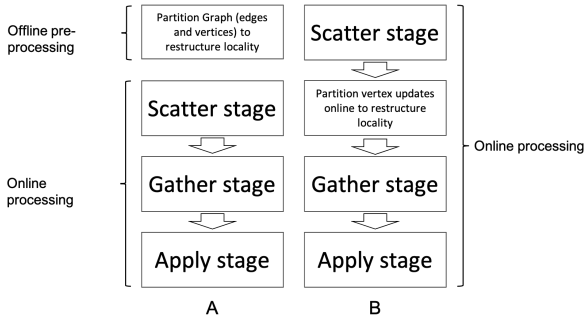


Figure 4.2: Prior art (A) vs. ACTS (B) graph processing workflow

4.4.1 Partition Vertex Updates not Edges

To tackle this problem, we propose the restructuring the URAM locality of vertex updates rather than of edges and vertices during pre-processing. This change moves restructuring from an offline to an online perspective. If the graph is restructured by partitioning into slices, the generated graph partitions are not entirely separate because several edges can span across partitions, especially when the size of the graph (and hence the number of slices) increases. This introduces the redundancy explained in section 6.1. On the other hand, if vertex updates (generated by processing source vertices and their incident edge properties) are restructured, there is clean separation between the generated vertex update partitions, removing the redundancy issue. Hence, every source vertex property is read only once from DRAM in every graph iteration before being discarded, and optimal DRAM read bandwidth usage of vertex property data can be maintained even when processing larger graphs. Figure 4.2A shows the processing flow employed by prior art where the graph's edges and vertices are first restructured before processing starts, while Figure 4.2B shows our proposed processing flow where the generated vertex updates are restructured instead. In our proposed processing flow, the source vertices and their outgoing edges are first processed according to the *edge_function* of the algorithm to generate vertex updates. These vertex updates are then partitioned according to their destination vertex IDs in the *Partition stage* to generate vertex-

update partitions, such that the destination vertex properties associated with each partition can fit in URAM. In the final *Apply stage*, the destination vertex properties for each partition are read from HBM into URAM, their corresponding vertex-update partitions are streamed from HBM to perform the updates.

4.4.2 Online Recursive Partitioning

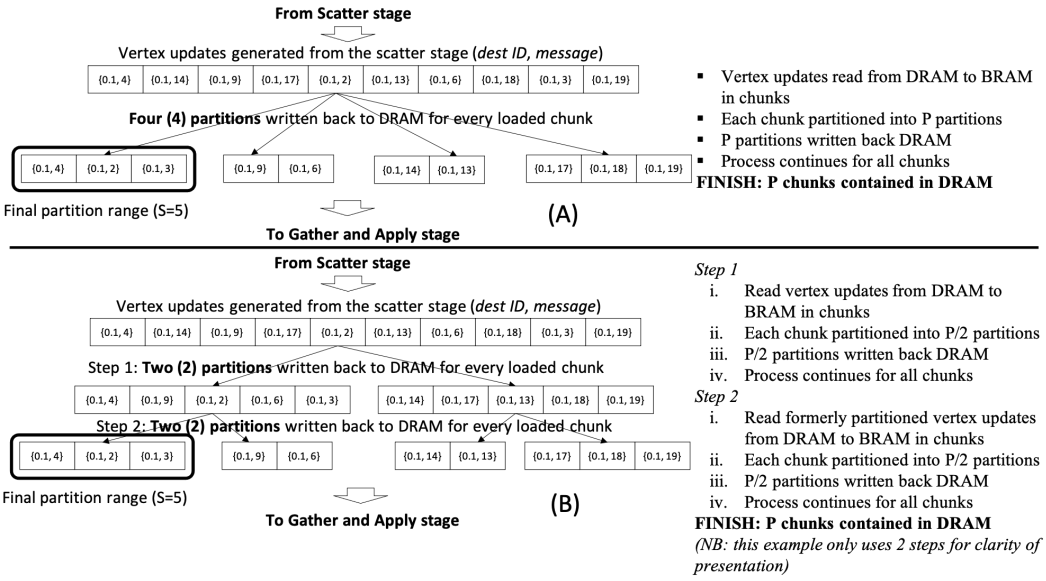


Figure 4.3: (A) Conventional Bucket-based partitioning vs. (B) Recursive Bucket-based partitioning

The main challenge associated with our proposed processing flow is the overheads that can be associated with online partitioning. With prior approach of partitioning the graph, the preprocessing cost of one-time graph partitioning is amortized over multiple iterations, runs, and algorithms. With our approach however, online partitioning needs to be done in every graph iteration. A high overhead partitioning strategy can therefore degrade throughput and obscure the no-redundancy advantages online partitioning intended to provide in the first place. A principal requirement of an effective online partitioning strategy is that it sustains minimal throughput degradation when handling large graphs. As the size of graph (i.e., number of vertices) increases, the range of desti-

nation vertex IDs in the vertex updates to partition increases. However, because the URAM within the FPGA has a limited fixed capacity, more partitions need to be generated to satisfy the URAM-locality requirement (where the destination vertex properties of each partition can fit in URAM).

Employing conventional online partitioning approaches such as bucket- and radix-based partitioning [44] [45] [46] [47] can suffer high overhead as the graph to process grows. This is because they can suffer overheads from DRAM access latency due to movements of small chunks of vertex update between fast BRAM and slower HBM memory of the FPGA board. In the conventional bucket-partitioning strategy, unstructured vertex updates are loaded from HBM to BRAM in BRAM-sized chunks in a *read stage*, partitioned on-chip using FPGA logic and BRAM into several high-locality vertex-update partitions (P_0 to P_p) each having range R. This happens in a *partition stage*. Each vertex-update partition is then written back into its distinct HBM-bucket locations in a *write-back stage*. These vertex updates are represented by key-value pairs (keys representing destination vertex IDs and values representing messages). This read, partition and writeback process occur repeatedly until the entire vertex updates generated from the scatter stage is processed. With larger graphs however, more partitions need to be generated to attain the same range (R) because of the fixed size of the FPGA's URAM. This can result in very small-sized chunks of vertex updates moving between BRAM and DRAM during the writeback process that consequently exacerbates the impact of DRAM access latency, and degrade the overall partitioning throughput. To reduce this overhead, we propose a recursive partitioning strategy. This fundamental idea behind recursive partitioning is to split the bucket-partitioning task into multiple steps to manage the granularity of data movements between BRAM and DRAM during partitioning, and therefore reduce the impact of DRAM Access Latency.

Figure 4.3B shows the recursive partitioning strategy, and how it differs from the conventional partitioning in Figure 4.3A. Both figures show vertex updates (U) with range (R=20) generated from the scatter stage. The range (R) refers to the range of destination vertex IDs of the vertex updates. Rather than completely partitioning each chunk read from DRAM into P(=4) partitions (like Figure 4.3A), our recursive partitioning strategy partially partitions them into P/K partitions (K=2 in this example) and write the P/K partial partitions back to DRAM (i.e., writeback process). After

the partial partitioning process is finished, each partial partition is then read back from DRAM using the same process to generate further P/K partial partitions. This recursive process continues until the the final locality of partitions is attained (S=5 in this simple example). The benefit of recursively partitioning over the conventional bucket-partitioning approach is that irrespective of the range (R) to be partitioned, we can maintain a fixed-sized granularity at which vertex updates are transferred between BRAM and DRAM during the writeback process, and alleviate the impact of DRAM access latency. For example, with the conventional bucket-partitioning approach in Figure 4.3A, four update partitions (buffered in BRAM) are written back to DRAM for every chunk (which will increase with R), while with recursive partitioning in Figure 4.3B, only two partitions are written back to DRAM for every chunk. After the recursive partitioning process finishes, the apply stage then *applies* these high-locality partitions at their respective destination vertices. An important research finding in this project is that the multiple passes through the vertex updates (U) required during recursive partitioning still presents a lower overhead compared to the conventional partitioning approach for several large graphs.

4.4.3 Efficient Edge Packing

The FPGA's AXI memory interface allows a wide bitwidth read and write accesses at rates reaching 512 bits per clock cycle from each HBM channel. This means 16384 bits of graph data can be read every clock cycle across all 32 HBM channels. Today's FPGAs can also deliver an impressive Internal SRAM Total Bandwidth of up to 30TB/s (Alveo U280 FPGA). These two provisions allow a rich environment for an embarrassingly parallel computational paradigm where many edges are streamed from the HBM in parallel, processed in parallel using independent BRAM/URAM and logic resources and written back to HBM, all in a pipelined fashion. This processing behavior can be hard to achieve with graph analytics because the unstructured nature of graphs can cause scenarios where two or more edges need to access the same URAM simultaneously to read source vertex properties, or two or more vertex update may need to access destination vertices in the same URAM simultaneously. Several strategies have therefore been proposed in attempt to com-

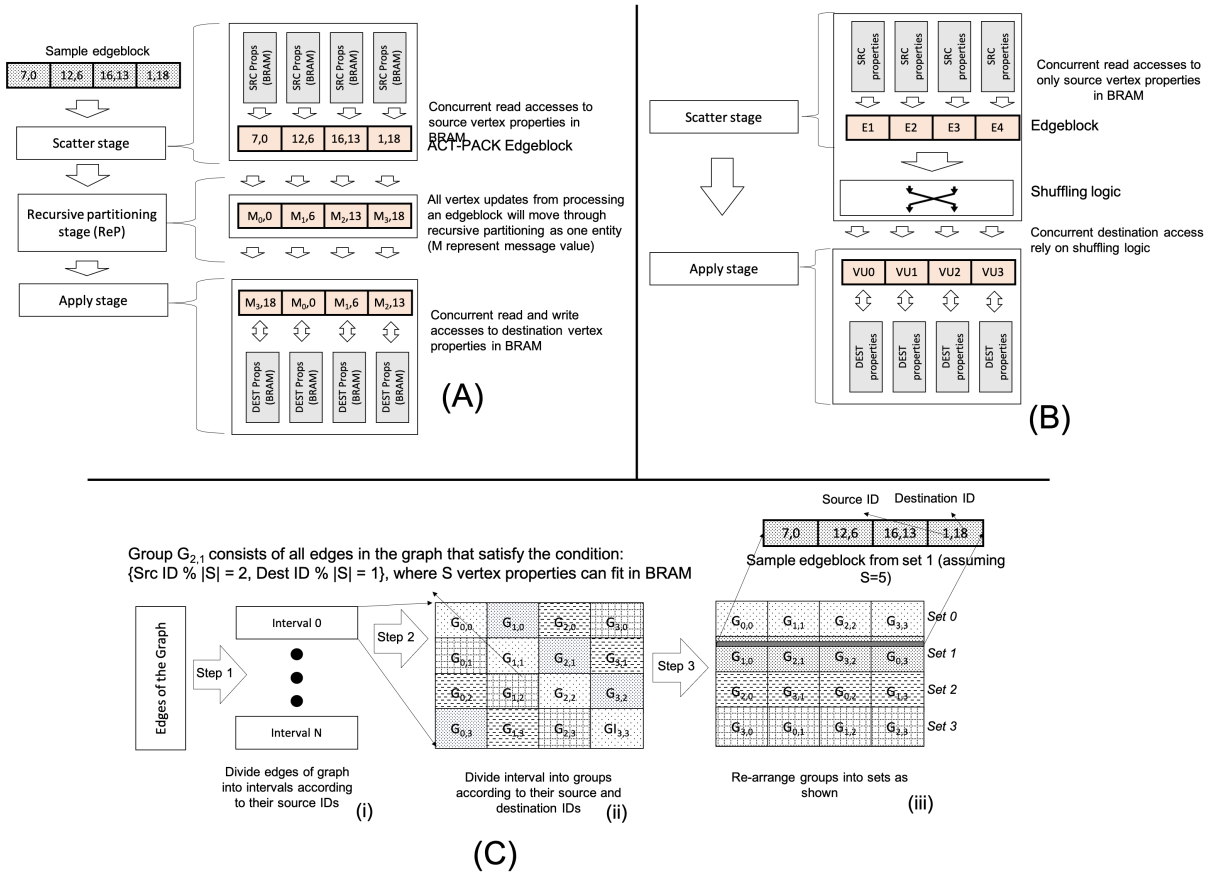


Figure 4.4: (A) ACTPACK representation allows concurrent BRAM accesses across both source and destination vertex ID dimension; (B) Prior-art representation allows concurrent BRAM accesses across a single vertex ID dimension; (C) How edges of a graph are represented in ACT-PACK

bat this problem. Prior art allow concurrent URAM accesses to vertex property data in only one dimension (i.e., source vertex ID or destination vertex ID), and would required shuffling [17] or arbitration logic [33] to optimize access in the other dimension (see Figure 4.4B). For example, CSPR [33] packs edges in HBM such that independent and parallel accesses can only be made to destination vertex properties in URAM. An arbitration infrastructure is however required to improve parallelism when reading source vertex properties. ThunderGP [17] employs a shuffling logic to improve parallelism when updating destination vertex properties in URAM. The limitations of arbitration or shuffling is both exploit suffer from workload imbalance among PEs which can serve as a bottleneck.

We propose a novel edge packing format called ACTPACK that allows this embarrasingly parallel streaming behavior when processing edges, partitioning vertex updates and applying vertex updates in the scatter-partition-apply paradigm. The graph to be processed is converted from whatever format it is represented (e.g., CSR format) into our ACTPACK format. This happens during pre-processing. With our edge packing, edges in HBM can be streamed in parallel, processed in parallel, and applied to their respective destinations in parallel, all while accessing independent URAMs and with no URAM conflicts. Also, because there is no need for shuffling and arbitration logic, the limitation of high logic utilization is eliminated. Figure 4.4B illustrates how edges represented in ACT-PACT support a high throughput graph processing pipeline. Given an edgeblock consisting four edges represented in ACTPACK as shown, concurrent read accesses can be made to source properties stored in BRAM during the scatter stage (at four edges per clock cycle), and concurrent read and write accesses to destination properties stored in BRAM during the apply stage. More importantly, all edges within an edgeblock will belong to the same partition after the recursive partitioning process, and can therefore all move as one unit through the entire recursive partitioning process. This allows the partitioning process to happen at wide-word rates of four (4) edges per clock cycle.

Figure 4.4C shows how edges are packed in HBM with the ACTPACK representation. First, the edges of a graph are divided into intervals according to their source IDs. Each interval has a source ID range of S, where S source vertex properties can fit in URAM. Second, edges within

each interval is further divided into groups by hashing both source and destination IDs. E.g. $G_{2,1}$ consist edges that meet the condition ($SrcID \mod |S| = 2$, $DestID \mod |S| = 1$). For clarity, we refer to the subscripts attached to each group as the dimensions of the group (e.g., group $G_{2,3}$ has x dimension equals 2 and y dimension equals 3). Third, groups are arranged into sets such that each set consists groups with contiguous y dimensions and cyclic contiguous x dimensions as shown in step 3 of Figure 4.4A. Fourth, edgeblocks are retrieved by selecting edges from distinct groups within the same set as shown in step 4.

4.4.4 Hybrid Processing of Sparse Frontiers

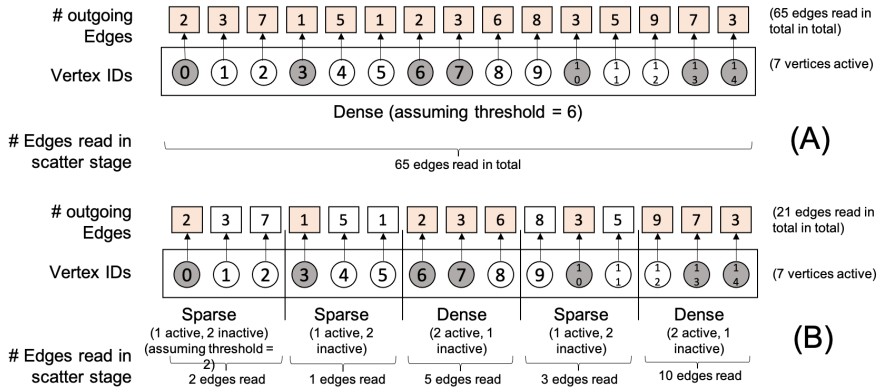


Figure 4.5: (A) Prior art's heuristic model labels the entire graph as sparse or dense in every GAS iteration; (B) Our heuristic model is more tightly coupled, labelling some parts of the subgraph as sparse and others as dense

Some graph algorithms such as Breadth First Search (BFS), Single Source Shortest Path (SSSP) and Connected Components (CC) are associated with sparse active subgraphs in certain iterations. These are a subset of the vertices of the graph (called active vertices) that are active in a given GAS iteration. They can be a small subset scattered in non-contoguous locations in HBM (termed sparse) or a large subset collocated within a given physical location in HBM (termed dense). Several FPGA graph accelerators employ the edge-centric computing style [8] because it allows sequential streaming of edges which can benefit from the high HBM bandwidth of the FPGA. This

processing style can however suffer from poor utilization of HBM bandwidth when processing sparse active subgraphs, as edgelists are scattered in non-contiguous locations in HBM. Therefore, and many edges can read from HBM with only few actually needed, as entire 64-byte cacheline data can be load from HBM data while operating on only a portion of the data. To tackle this challenge, prior art [33] resort to dual modes of execution, where the active subgraph can either be processed as a sparse or dense subgraph. They employ heuristic models that label the active subgraph in a given GAS iteration as either sparse or dense by recording the number of active vertices in that iteration. They also employ data structures suitable for each mode of processing. For example, [33] reads edges from CSR format when processing sparse subgraphs and their CSPR format when processing dense subgraphs.

Figure 4.5A and B show the heuristic model used by prior art and ACTS respectively. The shaded circles and boxes represent vertices and edges of the active subgraph respectively. Prior art (Figure 4.5A) labels the entire graph as dense since the number of active vertices (7) is greater than the threshold (assuming a threshold of 6 vertices). This means all (65) edges of the graph are read during the scatter stage. Rather than labelling the entire graph as sparse or dense, ACTS label intervals (i.e., regions of the graph) as dense or sparse depending on whether the number of active vertices they contain exceeds or lies below the threshold respectively. Assuming a threshold of 2 in this example, 21 edges are read in total compared to the 65 edges read in Figure 4.5A when processing this graph iteration. This extra level of coupling where sparsity is determined on an interval-by-interval basis rather than across the entire graph allows ACTS more accurately capture the different levels of sparsity across the graph, and prevent reading too many useless edges during processing. This reduces the bandwidth wastage when reading edges from HBM during processing. Like prior art, we also employ a dual data structures for each mode of processing. We employ the CSR (compressed sparse row) format to access sparse intervals and the ACTPACK format to read edges in dense intervals. This is because CSR represents outgoing edges in compact for (i.e., outgoing edges of a vertex are located in contiguous locations in DRAM) unlike ACTPACK that fragments the edges.

4.5 Evaluation

This section describes the experiments and analyses to evaluate ACTS. We chose the HLS C language because it generates low-level RTL from high level C/C++ language, and hides away all details of AXI control, BRAM/URAM memory and IP setup transparent from the programmer, allowing him concentrate on the application.

4.5.1 Target Hardware System

The hardware system for our hardware accelerator is the Xilinx Alveo U280 FPGA. The FPGA has 1.7 million LUTs, 960 288 Kb (270 Mb) URAM blocks and 41 MB BRAM blocks, making a total of for a total of 311 MB SRAM memory. The final clock frequency after hardware synthesis was 150 MHz. The FPGA is connected to 8 GB of HBM via 32 512-bit channels with a max frequency of 350 MHz. This means the peak line bandwidth is 460 GB/s. The FPGA is connected via PCIe Gen4x8, or 8 lanes of Gen4 to the host processor system. The host side of the graph applications were implemented in OpenCL C and executed on a 16-core Intel Xeon Silver 4216 CPU ruining at 2.10GHz. Our target hardware system features up to 32 HBM channel. Each channel is privately connected to its processing element (PE), which is responsible for processing the vertices and edges housed by that channel.

4.5.2 Applications

Four different analytics applications were explored in ACTS:

- PageRank [48]: Computes the PageRank score of each vertex in the graph
- Hyperlink Induced Topic Search (HITS) [49]: A link analysis algorithm that rates Web pages
- SpMV: Sparse Matrix-Vector Multiplication (SpMV) [22]

- Single-Source Shortest Path (SSSP) [50]: Computes the shortest distance to every vertex in the graph from a given root vertex

4.5.3 Datasets

We evaluate the performance of our framework over GraphLily [33] and Gunrock [51] using a mix of both synthetic and real-world static graph datasets. We chose these datasets because they expressed diverse cache behaviors. The synthetic datasets were generated from the RMAT graph generator [52] located at [53], while the real-world datasets were obtained from the University of Florida's Sparse Matrix Collection [54]. The probabilities $\{0.57,0.19,0.19,0.05\}$ were used as input parameters in generating the RMAT datasets. The synthetic datasets were included to investigate how ACTS will perform on a set of graphs having a fixed number of edges but a varying number of vertices. Due to limited HBM memory capacity (of 8GB), we could not run graphs larger than the RMAT26 dataset (67 million nodes and 268 million edges) on the FPGA for GraphLily or our work. For ACTS, this is because a portion of each HBM channel is also used to house vertex-update partitions and sorted edgelist representation required for processing. The next generation of HBM is expected to be 2GB per channel, allowing for handling larger graphs. Table 5.1 shows the properties of all the datasets evaluated.

4.5.4 Experimental Setup

Baselines We set up three different experiments to evaluate the impact of ACTS. Table 4.1 shows the system parameters of the evaluated system.

We implemented ACTS end-to-end, including I/O and FPGA kernel invocation costs in the first setup on a Xilinx Alveo U280 Ultrascale+ FPGA Accelerator Card with HBM memory bandwidth capable of delivering up to 460GB/s. The Xilinx HLS tool was used to generate RTL code from C++ HLS source, while the Xilinx Vitis tool was used to synthesize this design and run it on the Xilinx Alveo U280 FPGA board. All performance results are measured on the FPGA board. Vitis

	Xilinx Alveo U280	**NVidia Titan X**
Memory technology	HBM	GDDR5X
Max Bandwidth	460 GB/s	480 GB/s
Memory capacity	8 GB	12 GB
Max clock	300 MHz	1500 MHz
Max Power	225 W	250 W

Table 4.1: EVALUATION SYSTEMS

could only synthesize up to 24 PEs and yielded a clock frequency of 170 MHz (out of 300 MHz) and a HBM memory clock of about 350 MHz (out of 450 MHz).

In the second setup, we ran GraphLily [33] on the same Xilinx Alveo U280 Ultrascale+ FPGA Accelerator Card. GraphLily's source code is publicly available on GitHub, so we could run experiments across a variety of datasets and also gain insight into its memory traffic behavior using the Xilinx Vitis software emulation tool. We chose GraphLily because it is also an FPGA-based graph framework that utilizes the HBM memory. Because GraphLily did not provide an implementation for HITS, we compared it against PageRank, SSSP, and SpMv.

In the third setup, we ran Gunrock on the NVIDIA Titan X GPU (with DDR5 off-chip memory bandwidth of 480GB/s and base clock speed of 1417MHz). We chose Gunrock [51] for comparison because 1) It is one of the fastest graph processing engines available, and it uses the wide off-chip memory bandwidth of the GPU. 2) Today's GPUs are available with HBM, and 3) Gunrock adopts a similar programming model as ACTS for running graph analytics (i.e., the Gather-Scatter-Apply model). We chose the NVIDIA Titan X GPU because its bandwidth closely matched that of the Xilinx Alveo U280 Ultrascale+ FPGA Accelerator Card. Ideally, we would incorporate the GPU memory coalescing framework into a graph engine running on an FPGA, but NVIDIA's memory coalescing algorithm is highly tuned and proprietary.

In the fourth setup, we ran the GAP benchmark suite [55] as a baseline. We used GAP because it

includes high-performance reference implementations and is a shared standard widely adopted by the graph processing community.

It should be noted that no tuning is required per benchmark or dataset in ACTS, and the threshold parameter to switch from ACTPACK to the sorted edgelist is a one-time setting passed as a parameter to ACTS. To derive the optimal threshold, we ran preliminary tests across random graphs on the Xilinx Alveo U280 FPGA to obtain an optimal Streaming Partitions size (i.e., 131072 edges) and the threshold value for switching between ACTPACK and the sorted edgelist (i.e., 8192 active edges). Also, ACTS is synthesized once to run all algorithms (PageRank, SSSP, HITS, and SpMv) used in our evaluation. The Processing and Apply functions corresponding to the four algorithms evaluated are synthesized on the FPGA, and an input parameter specifying the appropriate function is passed during processing.

Metrics (1) We measured performance in execution time (ms) and energy consumption in Watt. In the GPU and FPGA experiments, the execution time does not include the data transfer overhead from the host CPU to the GPU/FPGA accelerator over PCIe. We query GPU power using Nvidia-smi and FPGA using Xilinx's xbutil.

4.5.5 Resource Utilization

LUT	FF	DSP	BRAM	URAM	Clock frequency
870K (65.4%)	720K (25.4%)	339 (2.7%)	2001 (49.3%)	768 (80.0%)	150 MHz

Table 4.2: Resource utilization of ACTS on the Xilinx Alveo U280 FPGA

Table 4.2 breaks down the on-chip resource usage of each accelerator. The on-chip clock synthesis frequency is degraded due to congestion from multiple HBM AXI interfaces in the same SLR region. On Alveo U280, all the HBM channels are located in SLR0, and this results in severe congestion on SLR0 even with the coarse-grained floorplanning and pipelining.

4.5.6 Accelerator Performance

	KR21	ST-OV	OR	SOC-LJ	LJ	R23	R24	UK	R25	R26
GAP (ms)	46	16	43	29	32	214	354	66	532	699
GraphLily (ms)	27	7	36	14	17	192	276	364	-	-
Gunrock (ms)	44	10	79	19	21	70	75	82	88	103
ACTS (ms)	15	6	17	11	12	35	41	55	64	110
GraphLily vs. GAP	**1.7**	**2.3**	**1.2**	**2.1**	**1.9**	**1.1**	**1.3**	**0.2**	**-**	**-**
Gunrock vs. GAP	**1.0**	**1.6**	**0.5**	**1.5**	**1.5**	**3.1**	**4.7**	**0.8**	**6.0**	**6.8**
ACTS vs. GAP	**3.1**	**2.7**	**2.5**	**2.6**	**2.7**	**6.1**	**8.6**	**1.2**	**8.3**	**6.4**

Table 4.3: Execution time (in ms) for PageRank; Bottom section is Speedup (based on execution time)

From a performance standpoint, ACTS outperforms GraphLily in PageRank, SSSP, and SpMv algorithms across various datasets. An important observation is its efficient scaling behavior, where ACTS expresses greater speedup over GraphLily with the larger graphs (i.e., the last five datasets). Because ACTS' does not slice the graph during pre-processing, all source vertex properties are read only once during each GAS iteration. Therefore, ACTS is able to avoid reading redundant source vertex properties from HBM during processing. Upon further investigation using the Vitis software emulation tool, we realize that GraphLily experienced a surge in memory traffic when running the last five (larger) datasets from reading many unused vertex properties. This sharp degradation in performance is caused by slicing during pre-processing. Further profiling into GraphLily's source code confirms this observation, as a graph such as the RMAT24 dataset (consisting of 16 million vertices) needs to be split into about sixteen slices to make each slice BRAM-friendly. This forces many unused vertex properties to be read on-chip. ACTS overcomes this hurdle by incorporating efficient on-chip partitioning, lowering the overall read bandwidth usage. GraphLily could not run DL, R25, and R26 datasets (marked as *'*OOM'*) because their slicing technique created large replications of vertex properties that could not all fit in HBM memory. With the

	KR21	ST-OV	OR	SOC-LJ	LJ	R23	R24	UK	R25	R26
GAP (ms)	136	56	154	141	157	533	1023	550	1312	2062
GraphLily (ms)	194	75	345	201	244	2718	3045	3367	-	-
Gunrock (ms)	104	34	121	48	49	320	370	35	260	600
ACTS (ms)	47	24	41	53	62	220	184	147	404	502
GraphLily vs. GAP	**0.7**	**0.7**	**0.4**	**0.7**	**0.6**	**0.2**	**0.3**	**0.2**	**-**	**-**
Gunrock vs. GAP	**1.3**	**1.6**	**1.3**	**2.9**	**3.2**	**1.7**	**2.8**	**15.7**	**5.0**	**3.4**
ACTS vs. GAP	**2.9**	**2.3**	**3.8**	**2.7**	**2.5**	**2.4**	**5.6**	**3.7**	**3.2**	**4.1**

Table 4.4: Execution time (in ms) for Single Source Shortest Path (SSSP); Bottom section is speedup (based on execution time)

	KR21	ST-OV	OR	SOC-LJ	LJ	R23	R24	UK	R25	R26
GraphLily (ms)	22	6	29	11	14	155	223	294	-	-
ACTS (ms)	12	5	14	9	10	28	33	44	52	89
ACTS vs. GraphLily	**1.8**	**1.2**	**2.1**	**1.3**	**1.4**	**5.5**	**6.7**	**6.6**	**-**	**-**

Table 4.5: Execution time (in ms) for Sparse Matrix Dense Vector Multiplication (SPMV); Bottom section is speedup (based on execution time)

SSP algorithm, ACTS generally performs better than GraphLily on the this algorithm compared to the other algorithms. Unlike PageRank, SSSP is a traversal algorithm that consists of iterations with varying levels of frontier sparsities. GraphLily's processing engines for sparse (SpMSpV) and dense (SpMv) frontier subgraph sizes are highly optimized for only extreme sparsity levels (i.e., either very sparse or very dense), making it challenging to capture moderate frontier levels of sparsity. For example, running SSSP on LJ using GraphLily in Vitis software emulation mode revealed excessive read bandwidth usage (when using the SpMv accelerator) and excessive random accesses (when using SpMSpV accelerator) in several iterations of the SOC-LJ dataset.

Additionally, ACTS outperforms Gunrock in PageRank, SSSP, and HITS across various datasets

	KR21	ST-OV	OR	SOC-LJ	LJ	R23	R24	UK	R25	R26
Gunrock (ms)	155	34	195	42	30	375	403	32	313	443
ACTS (ms)	85	18	81	33	36	150	180	165	192	330
ACTS vs. Gunrock	**1.8**	**1.9**	**2.4**	**1.3**	**0.8**	**2.5**	**2.2**	**0.2**	**1.6**	**1.3**

Table 4.6: Execution time (in ms) for Hyperlink-Induced Topic Search (HITS); Bottom section is speedup (based on execution time)

with some exceptions, demonstrating competitive performance across increasing graph sizes. An outlier is the UK dataset, where Gunrock outperforms both ACTS and GraphLily by a considerable margin with SSSP. This dataset is highly regular and benefits greatly from high cache hits when running graph traversals. Further investigation using NVidia's Nsight tool revealed that this dataset experiences up to 82% cache hit rate compared to others like the Orkut dataset (OR) which experienced only 34%.

4.5.7 Energy Usage

We also measure the energy consumption from running PageRank and SSSP across all datasets and show the results in Tables 4.7 and 4.8. ACTS reduces average power by an average of about 50% compared to Gunrock. The mean energy-delay product (EDP) of ACTS is 88% lower. Further power profiling of ACTS revealed that up to 80% of ACTS' overall power consumption is used by the HBM memory channels, while only about 20% is spent by on-chip FPGA activity.

	ST-OV	LJ	POK	KR20	KR21	UK	DL	R24	R25	R26
Gunrock (mJ)	5.0	1.1	8.9	2.1	2.4	7.9	8.5	9.3	9.9	11.6
ACTS (mJ)	0.7	0.3	0.7	0.5	0.5	1.5	1.8	2.4	2.8	4.8
Energy Improvement	**7.5**	**4.3**	**11.9**	**4.4**	**4.5**	**5.1**	**4.7**	**3.8**	**3.5**	**2.4**

Table 4.7: Energy consumption for PageRank in milli joules; Bottom section is energy improvement (ACTS vs. Gunrock)

	KR21	ST-OV	OR	SOC-LJ	LJ	R23	R24	UK	R25	R26
Gunrock (mJ)	11.8	3.8	13.7	5.4	5.5	36.2	41.8	4.0	29.4	67.8
ACTS (mJ)	2.1	1.1	1.8	2.3	2.7	9.7	8.1	2.1	17.8	22.0
Energy Improvement	**5.7**	**3.6**	**7.6**	**2.3**	**2.0**	**3.7**	**5.2**	**1.9**	**1.7**	**3.1**

Table 4.8: Energy consumption for SSSP in milli joules; Bottom section is energy improvement (ACTS vs. Gunrock)

4.6 Related Work

4.6.1 FPGA-based Graph Processing Frameworks

There has recently been significant interest in the area of accelerating graph analytics using FPGA and ASIC accelerators. The advantages of FPGA accelerators over ASIC is the re-configurability it provides. Several FPGA accelerators [17–19, 33–37] and ASIC accelerators [9] rely on available on-chip URAMs to deliver superior performance. Graphicionado demonstrates up to 6.5X speedup compared to state-of-the-art software solutions, owing a majority of its performance improvement to the efficient use of large on-chip scratchpad memory. For larger graphs whose vertex properties can not fit in fast, on-chip memory, such as BRAM or URAM, these accelerators resort to *graph slicing*, where a graph is first sliced during offline pre-processing to improve the BRAM (or URAM) memory-access locality, after which all slices are then loaded into the accelerator,

which processes them one slice at a time. GraphGen [18] focuses on generating an application-specific accelerator for a given vertex program specification rather than providing a single re-usable domain-specific accelerator. On the other hand, FPGP [37] targets a different problem where edges are stored in a device with extremely limited bandwidth (e.g., disk). GraphOps [56] is a concurrent work that provides a set of modular hardware units implemented in FPGA for graph analytics. GraphOps optimizes for graph storage and layout to provide efficient use of the off-chip memory while ACTS optimizes for graph access patterns utilizing an on-chip scratchpad and eliminating unnecessary off-chip memory accesses for efficiency. Lastly, Tesseract [18] targets the same domain as our work, but explores different technology by implementing specific hardware extensions using the logic layer of a 3D-stacked DRAM.

4.6.2 GPU- and Software-based Graph Processing Frameworks

There are a few graph analytics software frameworks and libraries specifically optimized for GPUs; Gunrock [51], MapGraph [57], nvGraph [58], and Enterprise [59] are representative examples. Fair comparisons against GPU-based frameworks are difficult since GPUs often run with much larger clock frequencies bandwidth than what FPGAs can provide. Several software graph processing frameworks also exist. GraphChi [60], TurboGraph [61], and XStream [8] are also similar frameworks utilizing disk-based systems for graph processing. Since these frameworks often focus on optimizing for efficient data locality and access patterns, they are closely related to ACTS; however, ACTS is a hardware implementation that optimizes for off-chip memory bandwidth consumptions rather than memory-to-disk bandwidth consumptions.

4.7 Conclusion

We presented ACTS, an accelerator for HBM-equipped FPGAs based on the push-based edge-centric computation style. ACTS addresses the excessive bandwidth usage overhead experienced by prior art FPGA-based frameworks in two important ways. 1) It removes the requirement of

offline slicing and embedding it within the Gather-Apply-Scatter abstraction. This allows ACTS to maintain an optimal read bandwidth usage of vertex property data, making it scale more efficiently to larger graph sizes. 2) ACTS optimizes the read bandwidth usage of edge data by integrating a more tightly coupled heuristic model that efficiently captures the various levels of sparsity when processing sparse active frontiers. This allows it to optimize data transfer efficiency and maintain the high on-chip memory parallelism benefit of the edge-centric style. ACTS shows a geometric mean speedup of 3.6X over GraphLily, a state-of-the-art HBM-enabled FPGA graph accelerator, and 1.5X over Gunrock, a state-of-the-art GPU graph accelerator. ACTS also shows a geometric mean power reduction of 50% and a mean reduction of energy-delay product of 88% relative to Gunrock. Future work with ACTS will be to deploy ACTS in a cluster-scale setting.

Chapter 5

Swift: Accelerated Graph Processing with Multiple FPGAs

5.1 Challenges

Graph processing with multiple FPGAs face major challenges that are not captured in their single-FPGA-based counterparts. The objective to use multiple devices (whether CPUs, GPUs or FPGAs) for processing graphs is to handle graphs too large to fit in a single device, or achieve an increased throughput from the combined bandwidth provided by each individual device. However, our comparisons of prior multi- and single-FPGA graph accelerators reveal that the multi-FPGA solutions sometimes perform even worse than their single-FPGA counterparts. The multi-FPGA accelerators attribute this performance degradation to limited inter-FPGA communication channel and frequent substitution of on-chip vertex properties across processing multiple slices of the graph. The underlying issue with frequent substitution of on-chip data is redundancy which was tackled in chapter 4 of this dissertation. The challenge of limited inter-FPGA communication bandwidth stems from the memory-centric characteristic of graph processing which forces vertices to have frequent communication between each other both within and between FPGAs. This in turn creates a non-trivial amount of data movements across the limited PCIe communication channel as shown

in Figure 5.1. This communication channel, which can have much lower bandwidth than off-chip memory (HBM or DRAM), naturally becomes a bottleneck. We tackle this issue by decoupling the fundamental parts of the graph processing paradigm, so that heterogeneous processing activities can occur on the graph in each FPGA in the cluster simultaneously. Therefore, edges can be processed in a region of the graph for a given iteration, with active frontiers in another region exported to remote FPGAs, and active frontiers imported into another region from remote FPGAs, all happening simultaneously. This saturates all channels (PCIe/QSFP, HBM and on-chip BRAMs/URAMs) in the cluster, and hides expensive FPGA-to-FPGA communication latency overheads within the entire graph processing flow.

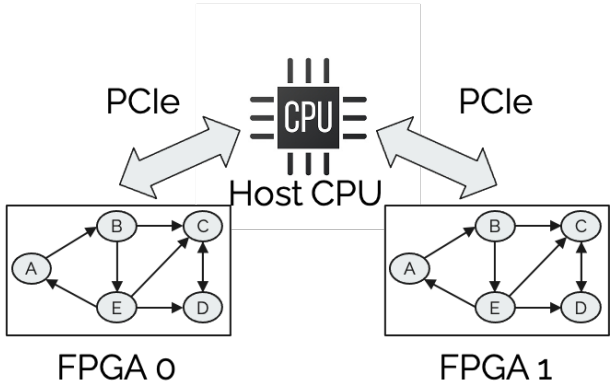


Figure 5.1: PCIe is a Bottleneck in multi-FPGA Graph Processing

In this work we also identify a bottleneck during the exchange of graph data between FPGAs at the end of every graph iteration. The goal of workload placement strategies in graph processing is to place the graph workload across the FPGAs such that workload balance is achieved. Workload placement can distort the original vertex ID ordering of the graph, and vertices in contiguous physical locations in the HBM can end up having non-consecutive vertex IDs. Transferring the vertex properties of these unstructured vertices from one FPGA (sender) to another FPGA (receiver) in between graph iterations will incur random access to HBM when re-arranging them at the receiver FPGA. This random access behavior limits throughput. We tackle this issue by proposing a workload placement strategy restricts the workload placement process to allow both workload balance and avoid this bottleneck. The fundamental idea is to divide the graph into vertex intervals and

place the graph workload across the cluster one vertex interval at a time, maintaining workload balance across each vertex interval. This allows graph data received into a local FPGA from a remote FPGA to be re-ordered using high speed URAMs

5.2 Implementation Details

5.2.1 Decoupled Asynchronous Execution Flow

Our decoupled asynchronous execution model is inspired by the fact that when a graph within an FPGA is processed to generate active vertices, a time window exists between when these updated vertex properties are generated and when they will be used by another FPGA in the next iteration. This time window can be overlapped with other high-latency processing operations such as the exchange of graph data between FPGAs. The fundamental difference between our decoupled asynchronous execution model and the conventional bulk synchronous execution model is that our model relaxes the bulk synchronization demands of the conventional edge-centric model. Bulk synchronization forces an operation to be executed across all active edges (which constitutes a *phase*) in the graph before the next operation can commence. The edge-centric execution model separates processing into *phases* (Process_Edges, Partition, Apply, Exchange). The first phase (*Process_Edges*) involves processing active edges to generate vertex updates, the second phase (*Partition*) involves restructuring the URAM locality of these updates via partitioning, the third (*Apply*) involved applying these updates at their respective destination vertices, and the fourth (*Exchange*) exports these updated vertices to remote FPGAs in the cluster. Therefore, only one phase can be executed on the graph at a time. Our single-FPGA work discussed in section 4 is based on this bulk synchronous model. Because each of this phase keeps only one channel busy (either the HBM, PCIe (write) and PCIe (read) channel), the the other two remain idle. This is a limitation to throughput, and poses limitations when mapping our single-FPGA work to the clusterscale setting. Our decoupled asynchronous execution model, on the other hand, relaxes this bulk-synchronization demands and allows multiple operations to occur on the same graph

simultaneously. It decouples these processing operations from the underlying graph structure, such that different operations can be executed in different sections of the graph within an FPGA at the same time. This is shown in Figure 5.3. In other words, an operation (e.g., processing of edges to generate vertex updates) can be happening in a given region of the graph, with another operation (e.g., importing of vertex properties) happening in another region of the same graph, and another operation (e.g., exporting of vertex properties) happening on yet another region of the same graph, all happening simultaneously. It allows opportunity for cross-iteration-activity, where separate regions of the graph can be consecutive iterations at a given time.

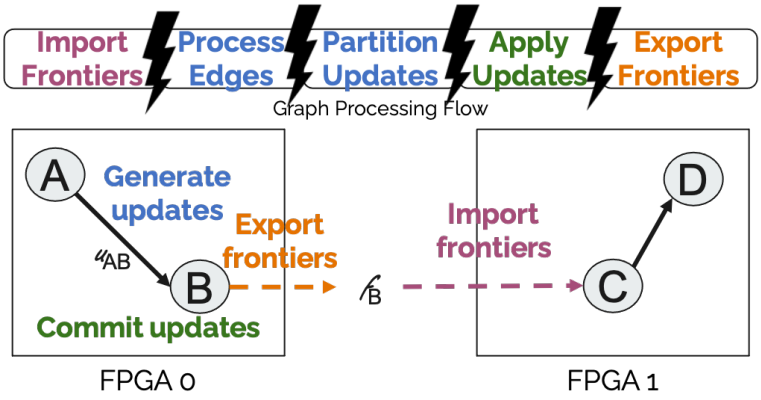


Figure 5.2: Decoupled Graph Execution Model

This interleaving between computation and communication keeps all three channels of each FPGA (i.e., BRAM, HBM and FPGA-to-FPGA communication channels) busy and hides the expensive PCIe communication latency. We illustrate this insight with a simple schematic shown in Figure 5.3. The graph workload assigned to each FPGA is first divided by vertex IDs into *vertex intervals*. This is a pre-processing step. A vertex interval consist a set of vertices with the edges that point to them (i.e., incoming edges). Within each FPGA, three managers exist that runs process_edges or apply_updates, export vertex updates to remote FPGAs and import vertex updates from remote FPGAs on the underlying graph. They are the process manager (PM), export manager (EM) or import manager (IM) respectively. During graph processing, a vertex interval can be in any one of three states at a time (*ready-for-process*, *ready-for-export* and *ready-for-import*). In ready-for-process state, the active edges belonging to the interval are ready to be processed to

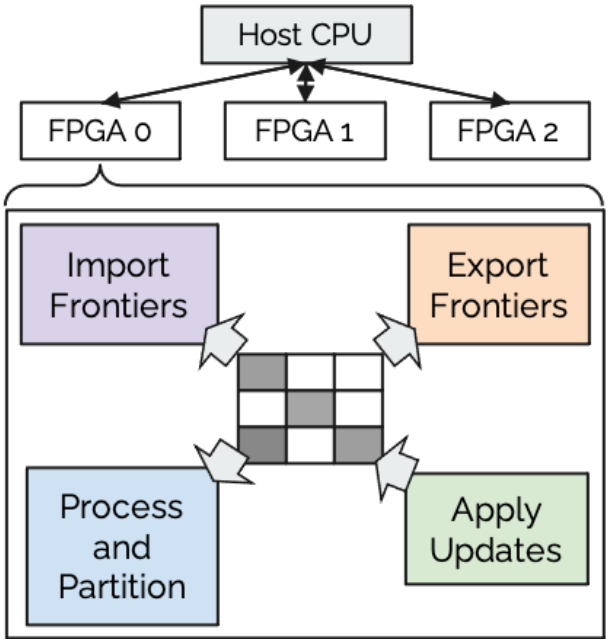


Figure 5.3: Decoupled Operations Executing Asynchronously on Graph

generate vertex updates. In the ready-for-export state, the vertices of a vertex interval has been updated and its contents (i.e., active frontiers) can be now be exported to remote FPGAs. This happens when the apply_update operation is completed on that vertex interval. In the ready-for-import state, the vertex interval is ready to receive updated vertices (i.e., active frontiers) from remote FPGAs for processing. The process manager (PM), export manager (EM) and import manager (IM) continuously poll the state of the vertex intervals to perform computation, exportation and importation operations respectively. When running computation over a given interval, the PM process outgoing edges in that interval, generate vertex updates and applies these updates using the process_edge, partition_update and apply_updates operations respectively. When performing export operation over a vertex interval, the EM sends the active frontiers to a URAM buffer which is then sent across the PCIe to remote FPGAs. When performing import operation over a vertex interval, the IM collects active frontiers from a URAM buffer and stores them in HBM. The execution of PM over a vertex interval changes its state to ready-for-export, the execution of EM over a vertex interval changes its state to ready-for-import, and also changes the state of the target remote interval to ready-for-import. The execution of IM over a vertex interval changes the state of the

interval to ready-for-process. This allows the managers can pass commands between themselves and maintains consistency.

5.2.2 Example Flow

We explain the working of our execution model in a simple example listed below. Assume a cluster of four (4) FPGAs, with FPGA 0 being the reference FPGA. Note that $FPGA_X$ means FPGA X and I_{YX} means vertex interval Y in FPGA X.

- At the start of processing, all vertex intervals containing active vertices in $FPGA_0$ to $FPGA_3$ set to the ready-to-process state.
- The process-edge module (PM) in $FPGA_0$ is invoked when it sees the ready-to-process state (because there is at least one interval containing active vertices). It runs the process-edge and partition-update operations over vertex interval $I_{0,0}$. The result is a set of vertex updates stored in different partitions.
- The process-edge module (PM) continues this operation until all vertex intervals containing active vertices in $FPGA_0$ are processed.
- After all vertex updates are generated and partitioned, the Apply-updates module (AM) then executes the apply function on each vertex-update partition, producing a new set of active vertices (also known as *active frontiers*) for each vertex interval.
- The Apply-updates module (AM) marks as ready-for-export each vertex interval whose new active frontiers have been generated.
- The export-frontier module (EM) is invoked upon the ready-for-export flag in a vertex interval and exports the active frontiers generated to the host CPU.
- The process-edge module (PM) and export-frontier module (EM) continue their apply and export operation concurrently, generating and exporting frontiers from FPGA 0.

- When active frontiers belonging to a vertex interval is exported, the export-frontier module (EM) marks all intervals in remote FPGAs that mirror that interval as ready-for-import.
- This invokes the import-frontier module (IM) in the remote FPGAs to import these frontiers from the host memory to their FPGA off-chip memory. At this point during processing, three operations (apply update, export frontiers and import frontiers) are happening simultaneously in the same FPGA, keeping the PCIe and HBM busy.
- Because $I_{0,0}$ frontiers are also used by $FPGA_0$ no importing of $I_{0,0}$ is required for $FPGA_0$.

5.2.3 Graph Layout in Memory

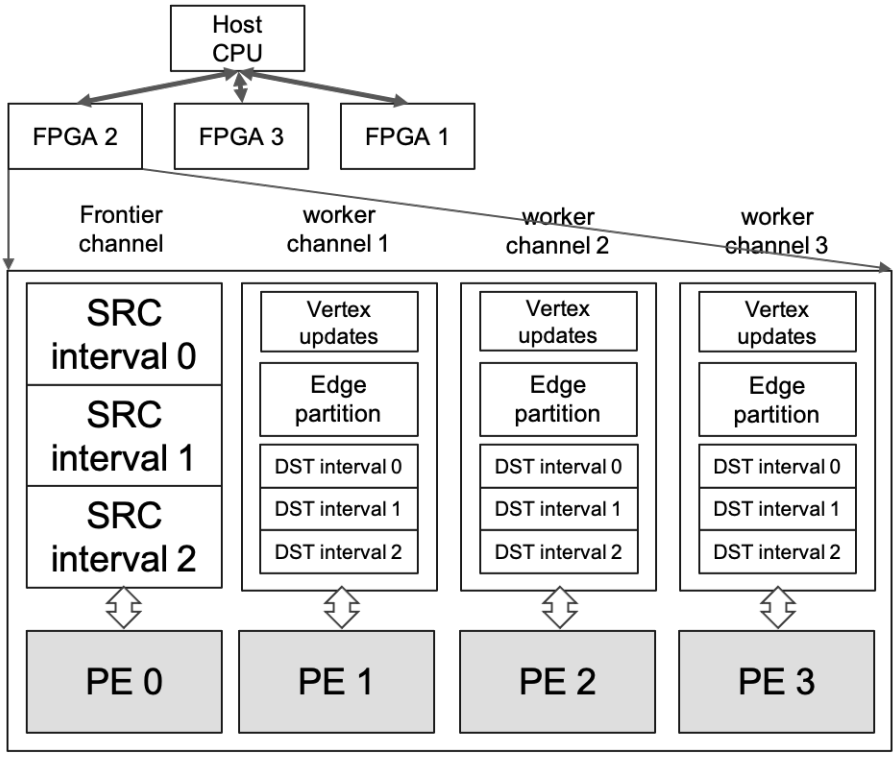


Figure 5.4: Graph Layout in HBM

In each FPGA, a single HBM channel (called *frontier HBM*) is dedicated to house active frontiers imported from the communication channel. The other HBM channels in the FPGA (called worker

HBMs) each house a portion of the vertices of the graph and their incoming edges. Each processing element (PE) is connected to a worker HBM and processes the edges within that channel. The layout is shown in Figure 5.5. Maintaining unique edges and vertices across the HBM channels avoids duplication of graph data and maintain storage efficiency. The vertices in each HBM is divided into vertex intervals with range V/NUM_PEs, where V vertex properties can fit in URAM. NUM_PEs is the number of processing elements in the cluster, with each PE connected to a worker HBM. Therefore, the combined range of vertex intervals 0 across all PEs in the cluster is V. The placement model (discussed in more details in section 5.2.4) ensures workload balance such that each worker HBM across the cluster consists roughly the same number of vertices and edges.

5.2.4 Workload Balance Across The FPGA Cluster

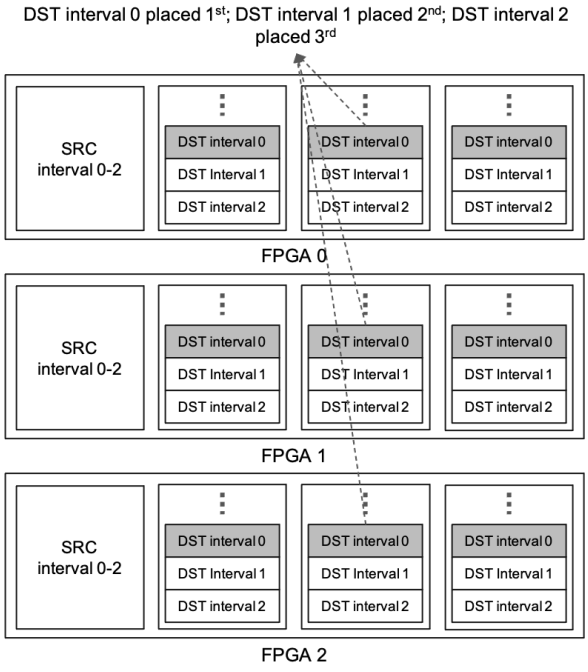


Figure 5.5: Workload Balancing Strategy

A clusterscale, HBM-enabled FPGA environment provides abundant parallelism at different levels. An efficient workload placement strategy is therefore critical to improving performance by

exploiting all these parallelism and preventing any stragglers in the system. The independent FPGAs in the cluster provide the first level of parallelism while the independent HBM channels (i.e., 32) within each FPGA provides the second level of parallelism. This means up to 128 PEs can be running independently in an 4-FPGA setup. Any straggler PE within the cluster can easily become a bottleneck and impact throughput, so an efficient graph placement strategy that ensures uniform workload balance is important.

The limitation of prior workload placement schemes is that while workload balance is achieved, bottlenecks can be created within the system that sacrifice throughput during graph processing. Our placement strategy advances over prior art by providing workload balance without sacrificing throughput. To balance workloads among different machines, prior art [62] distributes the graph across the machines (e.g., CPUs) by a distributed graph placement algorithm that maintains workload balance and allows minimum edge cuts across machines. All these happen during preprocessing. Their placement model has a high degree of freedom as any vertex can be placed in any machine in the cluster, as long as each machine has roughly the same number of vertices and edges at the end of the placement process. This freedom sacrifices throughput of an important datapath in the system — when frontiers are generated in a given machine to when they are processed in another machine in the next iteration. Because a graph is unstructured, some edges are cut and span across machines. Distributing the graph therefore distorts the sequential ordering of vertex IDs of the graph as two originally contiguous vertex IDs can be assigned to different machines. To achieve storage efficiency, vertex translation is employed so that each machine maintains a *global-to-local* hashtable that translates global vertex IDs to local vertex IDs and vice versa. The global vertex ID of a vertex is its original vertex ID in the original graph, while the local vertex ID is an assigned ID local to the machine where the vertex resides. Translation is therefore required when exporting generated frontiers from one FPGA to another across the communication medium. The random accesses involved with this translation at both the sender and receiver machine constitutes the first bottleneck to throughput. Also, because the active frontiers coming into an FPGA from other remote FPGAs are unstructured, processing these frontiers will involve reading edgelists from random locations in DRAM. This constitutes the second bottleneck. While these bottlenecks

are not obvious with clusterscale CPU-based graph processing environments [62], where more dominant bottlenecks exists, their impact is more pronounced in FPGA-based architectures where these other competing bottlenecks have been tackled.

To achieve workload balance and still maintain high throughput, we make certain adjustments in our graph workload placement strategy. The objective is to 1) avoid translations at the sender FPGAs, 2) avoid translations at the receiver FPGAs, and 3) leverage fast URAMs when processing chunks of active frontiers received into an FPGA. To achieve 1), we include the global vertexID information when representing destination vertex properties in each FPGA. This extra information is read from HBM and written to HBM during the apply operation. With this, there is no need to translate active frontiers at the sender end. This increases the storage space occupied by destination vertex properties in HBM, and introduces overheads when reading and writing destination vertex properties to/from HBM. However, these are trivial overheads compared to the parallelism benefit it offers. Because the referencing of vertices in all FPGAs in the cluster is the same (i.e., via global IDs, as discussed in section 5.2.3), 2) is naturally achieved. To achieve 3), we enforce a restriction while placing graph workload with any of the novel placement schemes discussed in [62]. Rather than a free placement strategy where vertices can be placed in any machine in the cluster as long as workload balance is maintained at the machine level, we place the graph one vertex interval at a time, achieving workload balance on vertex-interval basis. Assume a cluster consisting 3 FPGAs as shown in 5.5, all vertices and edges in vertex interval 0 is placed across the entire cluster to maintain workload balance in this interval before moving to the next. At the end of placement, the graph workload is not only balanced across the different machines in the cluster but also across the different vertex intervals in the cluster. An important implication of this design decision is that when the import manager (IM) of a given FPGA imports the active frontiers of a given vertex interval from all remote FPGAs across the cluster, these imported active frontiers can all fit in, and be re-arranged in, low-latency URAM. Our re-ordering strategy is explained in more details in section 5.2.5

Balancing algorithm To balance workload within a given vertex interval, we first partition the

destination vertices in that interval into several groups based on their in-degree percentiles (i.e., the number of edges pointing to them). For example, a given distribution can be 0-24th, 25th-49th, 50th-75th and 75th-100th percentiles, where 0-24th percentile represent vertices whose in-degree lie within 0-24% the mean in-degree in the vertex interval. The vertices in each group is then interleaved across the worker HBM channels of the cluster according to the formular i % NUM_PEs (i is the ith vertex in a given group, while NUM_PEs is the total number of PEs in the cluster). This distributes the vertices and edges data of a vertex interval evenly across the cluster.

5.2.5 High Throughput Exchange Datapath

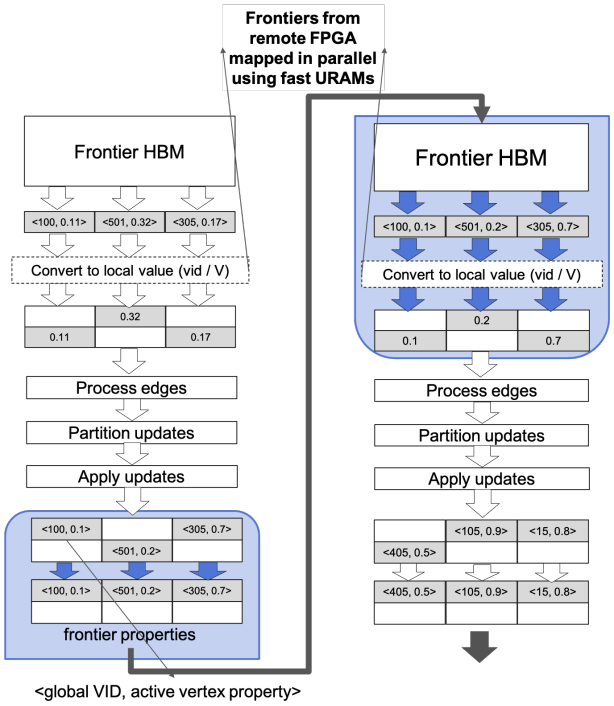


Figure 5.6: High Throughput Exchange Datapath

In figure 5.6, we illustrate this datapath with a simple example, and how our workload placement strategy (discussed in section 5.2.5) maintains high throughput in this datapath.

At some time during processing the process manager (PM) of FPGA 0 will perform the apply operation on interval 0 (i.e., $I_{0,0}$) to generate active frontiers. it will then label this interval as

ready-to-export. These active frontiers are unstructured because of their random, non-contiguous global vertex IDs. However, our workload placement strategy discussed in section 5.2.4 ensures that they will all be within a range that can fit in URAM. At another time afterwards, the export manager (EM) of FPGA 0 will export these active frontiers to the host CPU. Subsequently, the import manager (IM) of FPGA 1 will import these unstructured active frontiers and re-order them using the URAM as shown in figure 5.6. In this re-ordering process, the global ID of each frontier is first converted into a local ID by dividing it by V (i.e., size of a vertex interval), to get a unique index local to URAM. Its property value is then inserted into this local ID slot in URAM. Even though this hashing involves random accesses, the random accesses are made to fast on-chip URAM arrays. Also, because the URAM arrays can house vertices of an entire vertex interval, each active frontier of interval 0 is hashed into a unique slot in URAM. After this re-ordering is completed, the processing of these newly imported active frontiers can now commence. The concurrent generation of these active frontiers of $I_{0,0}$ in FPGA 0 using fast URAMs, and the concurrent reordering of the same in FPGA 1 using fast URAMs accounts for the high throughput of this exchange datapath.

5.3 Evaluation

5.3.1 Target Hardware System

We compare ACTS against a number of 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 also made first order comparisons against Gunrock because it was open sourced. We implemented ACTS end-to-end, including I/O and FPGA kernel invocation costs on four (4) Xilinx Alveo U280 Ultrascale+ FPGA Accelerator Cards with HBM memory bandwidth capable of delivering up to 460GB/s per FPGA. We tested Gunrock on a NVidia A40 GPU with HBM2 memory supporting 696GB/s per GPU. The Xilinx HLS tool was used to generate RTL code from C++ HLS source, while the Xilinx Vitis tool was used to synthe-

size this design and run it on the Xilinx Alveo U280 FPGA board. Vitis could only synthesize up to 24 PEs and yielded a clock frequency of 150 MHz (out of 300 MHz). In the second setup, we ran Gunrock on a cluster of GPUs (with DDR5 off-chip memory bandwidth of 480GB/s and base clock speed of 1417MHz).

5.3.2 Applications and Datasets

We evaluated three common graph algorithms, Pagerank (PR), Sparse Matrix Vector Multiplication (SPMv) and Hyperlink Induced Topic Search (HITS) to explore the novel contributions in ACTS. We chose these algorithms because they generalize the memory access behaviors shared by several other graph algorithms.

Dataset	**Abbr**	**# Vertices**	**# Edges**	**Type**
Indochina	IND	7.4M	194M	Real
Twitter	TW	41.6M	1.4B	Real
Sk-2005	UK	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

Table 5.1: GRAPH DATASETS UNDER EVALUATION
(M: millions; B: billions; Abbr: Abbreviation)

We evaluate the performance of these architectures using a mix of both synthetic and real-world static graph datasets. These datasets represent diverse cache behaviors — some exhibiting high inherent spatial locality while the others exhibiting lower locality. The synthetic datasets were

generated from the RMAT graph generator [52] located at [53], while the real-world datasets were obtained from the University of Florida's Sparse Matrix Collection [54]. The probabilities $\{0.57,0.19,0.19,0.05\}$ were used as input parameters in generating the RMAT datasets. Due to limited HBM memory capacity (of 8GB) per FPGA, we could not run some very large graphs in our 4-FPGA cluster. The next generation of HBM is expected to be 2GB per channel, allowing for handling larger graphs. Table 5.1 shows the properties of all the datasets evaluated.

5.3.3 Accelerator Performance

Algorithm	Graph	Metric	# FPGAs	Peak BW*	Performance	System	# FPGAs	Peak BW	Performance
PR	Twitter	execution time	4 FPGAs	1840 GB/s	84ms	ForeGraph [16]	4 FPGAs	304 GB/s	15s
PR	Twitter	execution time	4 FPGAs	1840 GB/s	84ms	PowerGraph [62]	512 CPUs	-	36s
BFS	Twitter	execution time	4 FPGAs	1840 GB/s	576ms	ForeGraph [16]	4 FPGAs	304 GB/s	7.9s
BFS	Twitter	execution time	4 FPGAs	1840 GB/s	576ms	TurboGraph [61]	CPU	41.6 GB/s	76s
BFS	Twitter	execution time	4 FPGAs	1840 GB/s	576ms	FPGP [37]	1 FPGA	-	121s
PR	RMAT_16	execution time	4 FPGAs	1840 GB/s	43ms	FDGLib [63]	16 FPGAs	1232 GB/s	7.35s
PR	RMAT_32	execution time	4 FPGAs	1840 GB/s	86ms	FDGLib [63]	16 FPGAs	1232 GB/s	7.84s

Table 5.2: Comparing ACTS with prior FPGA-based accelerators; Peak BW* refers to the total off-chip bandwidth available in evaluation platform; Performance in execution time (in ms)

Graph accelerator	Memory BW*	Effective BW*	Communication BW*	Clock Freq*
Gunrock [51]	696 GB/s	696 GB/s	112 GB/s (NVLink)	1305 MHz
ACTS	460 GB/s	345 GB/s	17 GB/s (PCIe)	150 MHz

Table 5.3: FPGA and GPU Platform specifications. Memory BW* refers to off-chip DDR4/HBM memory bandwidth; Communication BW* refers to PCIe/NVLink bandwidth between the FPGA/GPU respectively; Clock Freq* refers to clock frequency of the FPGA/GPU; Effective BW* refers to maximum bandwidth the algorithm can use upon deployment

MTEPS										
	IND	TW	SK	UK	SN	WB	R8	R16	R32	Geo-mean
Gunrock	8726	11241	22634	17950	8627	11742	15426	13243	10323	11800
ACTS	12179	16685	18740	13412	4012	6465	25795	20900	15936	13771
MTEPS/BW*										
	IND	TW	SK	UK	SN	WB	R8	R16	R32	Geo-mean
Gunrock	3.1	4.0	8.1	6.4	3.1	4.2	5.5	4.8	3.7	4.2
ACTS	8.8	12.1	13.6	9.7	2.9	4.7	18.7	15.1	11.5	10.0
Improvement	2.8x	3.0x	1.7x	1.5x	0.9	1.1x	3.4x	3.2x	3.1	**2.4x**
MTEPS/Watt										
	IND	TW	SK	UK	SN	WB	R8	R16	R32	Geo-mean
Gunrock	16.7	21.5	43.2	34.3	16.5	22.4	29.4	25.3	19.7	22.5
ACTS	69.2	94.8	106.5	76.2	22.8	36.7	146.6	118.8	90.5	78.2
Improvement	4.2x	4.4x	2.5x	2.2x	1.4	1.6x	5.0x	4.7x	4.6x	**3.5x**

Table 5.4: Comparing ACTS with Gunrock on 16 iterations of PageRank using 4 FPGAs/GPUs; Comm BW* refers to communication bandwidth between devices; Perf* refers to performance in million edges traversed per second or MTEPS (top); Perf*/Watt refers to energy efficiency in MTEPS / Watt (middle); Perf* / Band* refers to bandwidth efficiency in MTEPS / (GB/s) (bottom)

MTEPS										
	IND	TW	SK	UK	SN	WB	R8	R16	R32	Geo mean
Gunrock [51]	10907	11686	50799	32962	9649	17241	24700	20036	14937	16245
ACTS	12179	16685	18740	13412	4012	6465	25795	20900	15936	13771
MTEPS/BW*										
	IND	TW	SK	UK	SN	WB	R8	R16	R32	Geo mean
Gunrock [51]	3.9	4.2	18.2	11.8	3.5	6.2	8.9	7.2	5.4	5.8
ACTS	8.8	12.1	13.6	9.7	2.9	4.7	18.7	15.1	11.5	10.0
Improvement	2.3x	2.9x	0.7x	0.8x	0.8x	0.8x	2.1x	2.1x	2.2x	**1.7x**
MTEPS/Watt										
	IND	TW	SK	UK	SN	WB	R8	R16	R32	Geo mean
Gunrock [51]	20.8	22.3	96.9	62.9	18.4	32.9	47.1	38.2	28.5	31.0
ACTS	69.2	94.8	106.5	76.2	22.8	36.7	146.6	118.8	90.5	78.2
Improvement	3.3x	4.3x	1.1x	1.2x	1.2x	1.1x	3.1x	3.1x	3.2x	**2.5x**

Table 5.5: Comparing ACTS with Gunrock on 16 iterations of SPMV using 4 FPGAs/GPUs; Comm BW* refers to communication bandwidth between devices; Perf* refers to performance in million edges traversed per second or MTEPS (top); Perf*/Watt refers to energy efficiency in MTEPS / Watt (middle); Perf* / Band* refers to bandwidth efficiency in MTEPS / (GB/s) (bottom)

MTEPS										
	IND	TW	SK	UK	SN	WB	R8	R16	R32	Geo mean
Gunrock [51]	6886	-	12663	11152	7638	8300	12015	11675	10632	9779
ACTS	6594	-	9370	6643	1832	3232	13205	10942	7891	6713
MTEPS/BW*										
	IND	TW	SK	UK	SN	WB	R8	R16	R32	Geo mean
Gunrock [51]	2.5	-	4.5	4.0	2.7	3.0	4.3	4.2	3.8	3.5
ACTS	4.8	-	6.8	4.8	1.3	2.3	9.6	7.9	5.7	4.9
Improvement	1.9x	-	1.5x	1.2x	0.5x	0.8x	2.2x	1.9x	1.5x	**1.4x**
MTEPS/Watt										
	IND	TW	SK	UK	SN	WB	R8	R16	R32	Geo mean
Gunrock [51]	13.1	-	24.2	21.3	14.6	15.8	22.9	22.3	20.3	18.7
ACTS	37.5	-	53.2	37.7	10.4	18.4	75.0	62.2	48.8	38.1
Improvement	2.9x	-	2.2x	1.8x	0.7x	1.2x	3.3x	2.8x	2.2x	**2.0x**

Table 5.6: Comparing ACTS with Gunrock on 16 iterations of Hyperlink Induced Topic Search (HITS) using 4 FPGAs/GPUs; Comm BW* refers to communication bandwidth between devices; Perf* refers to performance in million edges traversed per second or MTEPS (top); Perf*/Watt refers to energy efficiency in MTEPS / Watt (middle); Perf* / Band* refers to bandwidth efficiency in MTEPS / (GB/s) (bottom)

5.3.4 Overall Performance

Tables 5.2, 5.4, 5.5 and 5.6 show the comparisons of Swift with these frameworks and accelerators and yield several important observations.

1. As shown in Table 5.2, Swift demonstrates superior performance over prior multi-FPGA frameworks. This is due to three main reason:

   (a) Within each FPGA, Swift handles the random accesses associated with vertex-to-vertex communication at higher throughput, allowing it to enjoy more of the HBM bandwidth and on-chip URAM parallelism within each FPGA. This is because of Swift employs online partitioning proposed in ACTS [64] to sequentialize access to HBM and use fast on-chip URAMs.

   (b) Swift's decoupled, asynchronous execution model (discussed in section 5.2.1) allows overlapping between computation (within the FPGAs) and communication (between the FPGAs) during graph processing. Prior art do not offer this advantage, causing the FPGAs to experience idle times during communication.

   (c) Swift's load balancing strategy allows for uniformly balanced workload across the cluster. It also supports high throughput when active frontiers migrate from one FPGA to another. (discussed in section 5.2.5).

2. Swift demonstrate mixed performance compared with Gunrock. It should however be noted that the GPU cluster used to evaluate Gunrock (A40 GPUs) has greater bandwidth and inter-FPGA communication link than the FPGA setup. The UK-2005 and IT datasets are highly regular datasets that exhibit high cache hit rates, making them benefit significantly from caching in the GPU. With the other relatively unstructured datasets, ACTS demonstrate superior throughput over Gunrock.

5.4 Conclusion

The FPGA has great potentials for accelerating graph processing on the clusterscale arena. Its flexibility in designing custom datapaths and memory subsystem to parallelize stubborn bottlenecks in standard graph algorithms is perhaps its greatest advantage. In this work we present Swift, a clusterscale graph accelerator for HBM-equipped FPGAs. Swift extends our ideas in our previous work [64] and addresses critical concerns that are not manifested in single-FPGA accelerators: These are 1) limited bandwidth of FPGA-to-FPGA communication infrastructure, and 2) throughput degradation that arises from prior workload balancing strategies. To tackle the first we propose a decoupled, asynchronous GAS-based execution model that allows the overlapping of three important graph processing primitives — computation, importing and exporting. This allows the saturation of communication (PCIe/QSFP), offchip (HBM/DDR) and on-chip (SRAM) bandwidth across the entire graph processing flow, effectively hiding inter-FPGA communication with intra-FPGA computation. With our model, computation can be run on a portion of the graph within an FPGA (called *vertex interval*) at the same time the importation of active frontiers is happening over another vertex interval, and at the same time exportation is happening over another interval, all within the same FPGA. To tackle the second we impose some constraints on prior workload placement strategies to maintain a high throughput *exchange datapath*, an important datapath in the cluster. Swift was built on top of our prior single-FPGA-based accelerator [64], and demonstrates superior performance over several prior FPGA-based frameworks and a well-known GPU-based accelerator.

Chapter 6

Dynamic ACTS: A Dynamic Graph Accelerator For HBM-Enabled FPGAs

6.1 Challenges

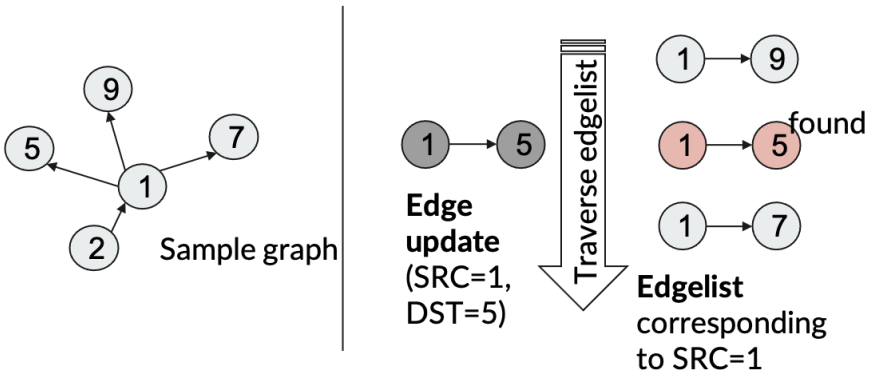


Figure 6.1: Dynamic graph updating model employed by prior art

In this chapter we revisit the processing of evolving graphs, this time using HBM-enabled FPGAs. We highlight a major limitation to the widely-adopted traversal-based approach employed by the state-of-the-art to updating evolving graphs (i.e., edge insertions, deletions and updating). We then propose a novel approach of tackling this limitation.

The technique employed by prior art to perform graph updating is *traversal based*. With this approach, edgelists of the graph are traversed to search for a matching edge before the insertion, deletion or update operation is executed. As discussed in section 3.4, the extent to which this search space is reduced depends on the traversal algorithm, and directly correlates to the performance of the data structure. RHH reduces it to $\mathcal{O}(\ln n)$, while GraphTinker reduces it further to $\mathcal{O}(\ln(\log_{P*} n))$ (n is the number of edges in the edgelist)

There are two main issues with this model. First, even with the best approaches to reduce this search space, this approach can still suffer throughput degradation from reading too many unused edges from DRAM, and only using a few of these edges. This is because these still depend on n. This places a limit on the throughput of graph updating models and prevent them from coming close to matching the throughput of graph processing models. Second, this approach cannot be interleaved (i.e., hidden within) with graph analytics tasks. Therefore, graph updating and graph analytics happen in separate time slots, where the graph is first updated before graph analytics is run on it and so-on and so-forth.

6.2 A More Promising Pathway

There is a wide gap in the graph community between the throughput when performing graph updates (i.e., edge insertions, deletions or modifications) to a dynamic graph and when running graph analytics on the same. A dynamic graph is one whose topology evolves with time, and graph analytics on such graphs have to keep up with the evolution of the graph at regular time intervals. In a typical processing context, updates are performed at regular intervals followed by graph analytics. This makes the throughput when performing updates as important as that in analyzing the graph, and causes a bottleneck (from amdahl's law) when one is far slower than the other. Recent state-of-the-art frameworks that support dynamic graphs demonstrate orders of hundreds to thousands of millions of edges per second throughput when running common graph algorithms, but only tens of millions of edges per seconds when performing edge-insertion. This is fundamentally because

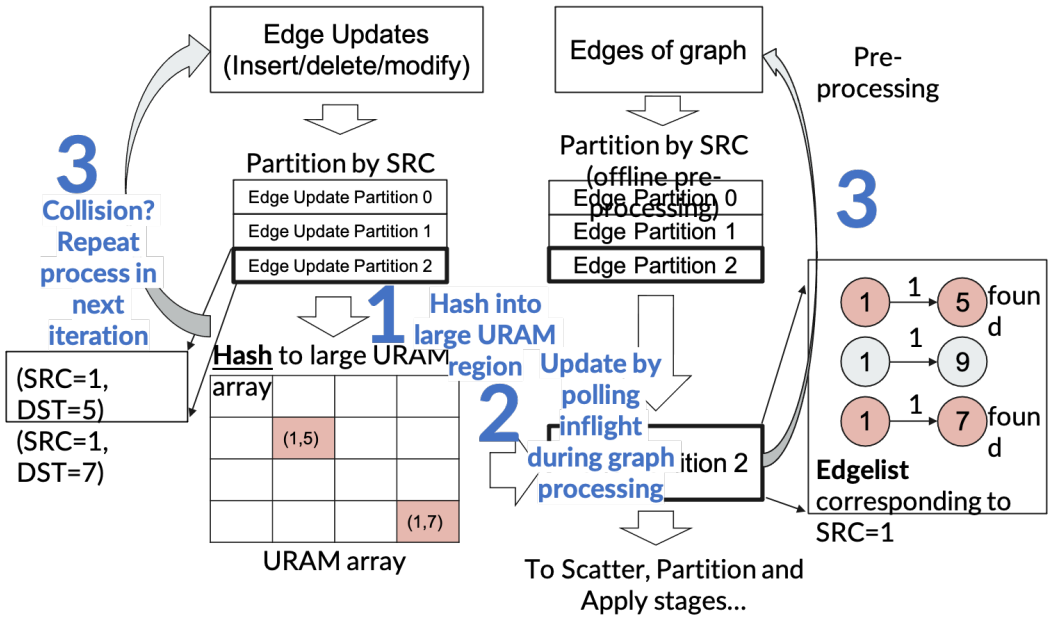


Figure 6.2: Interleaved Dynamic Graph Updating Flow

graph updating (i.e., edge insertions, deletions and edits) generally requires reading more unused edges than running graph analytics. The achievable throughput when performing graph updates is bounded by lookup operations where several unused edges will have to be read from DRAM and analyzed before the required edge is found. With graph analytics on the other hand, a majority of the edges read from DRAM are actually useful during processing. Due to amdahl's law, combining graph updating solutions with graph analytics solutions in a dynamic graph processing environment is bottlenecked by the poorer graph-updating throughput, and this makes even the fastest graph engines perform poorly in such scenarios. This project aims to tackle this issue.

The main insight to achieving our goal in this project is to identify high-latency operations common to both graph updating and graph analytics and interleave them together (hiding one in the other) such that such operations are performed only once when processing evolving graphs. This will allow graph updating to be performed at graph-analytics speeds. We currently highlight two expensive processes common to both:

- ***Traversing edgelists***: Both graph updating and graph analytics involve traversing edgelists. During edge insertions, edgelists are traversed when following edges to insert, delete or modify an edge. During graph analytics, edgelists are also traversed when traversing neighboring vertices to send messages from source to destination. Both operations are expensive as they incur costly accesses to high-latency DRAM.

- ***Loading of non-contiguous edgelists in DRAM***: During edge insertions consecutive edges to be inserted can belong to different edgelists. Therefore edgelists are loaded on-chip from non-contiguous vertices to perform insertions. Similarly, edgelists are also loaded on-chip from non-contiguous vertices when traversing a graph during graph analytics. Both these operations are also expensive as they also involve high-latency DRAM accesses.

Motivated by these common expensive operations, we propose to improve GraphTinker for FPGA-HBM environment and tailor it into ACTS' framework to support high performance and scalable dynamic graph processing. In this work, we will first expose the different layers of parallelism in GraphTinker which are (1) parallelism across source vertex IDs and (2) parallelism across destination IDs belonging to a given source ID. We will then propose methods to map efficiently to the FPGA leveraging the inter- and intra-HBM parallelism of the FPGA, as well as its high on-chip parallelism. Next, we will combine these high-latency tasks common to both so that such tasks are performed only once during the workflow.

In a typical processing flow, active vertex properties for the current graph iteration are read from DRAM with their corresponding outgoing edgelists. Additionally, graph updates (i.e., edges to be inserted, deleted or modified) corresponding to the edgelists will also be read from DRAM. These edgelists are then traversed on-chip to achieve two goals: (1) to generate messages and commit to neighboring destination vertex properties (graph analytics), and (2) to update the graph structure. At the end of the traversal process two outputs will be produced and committed back to the graph: (1) modified destination vertex properties, and (2) modified edgelists. This project will allow graph updating occur at throughputs competitive to running graph analytics on static graphs and bridge the gap between the two.

6.3 Conclusion

We presented Dynamic-ACTS, an accelerator for graph processing, and a graph-updating engine designed for HBM-equipped FPGAs. Dynamic-ACTS is based on the push-based edge-centric computation style. To accelerate graph processing, Dynamic-ACTS removes the requirement of offline slicing and embedding it within the Gather-Apply-Scatter abstraction. This allows Dynamic-ACTS to maintain an optimal read bandwidth usage of vertex property data, making it scale more efficiently to larger graph sizes. To accelerate graph updating (i.e., edge insertions and modifications to the dynamic graph), Dynamic-ACTS employ a hashing strategy where a group of edge updates is hashed to a large URAM array, and edges are streamed across the array to pick and apply their edge updates inflight. This approach shows significant performance benefits compared to the widely adopted technique where edgelists are first traversed to find a match before performing an update. This allows increased throughput to graph updating and the opportunity to hide graph updating within graph analytics. Dynamic-ACTS demonstrates superior performance over GraSU, a state-of-the-art graph updating engine for the FPGA. Future work with Dynamic-ACTS will be to deploy Dynamic-ACTS in a cluster-scale FPGA setting.

Chapter 7

Conclusion And Future Work

The use of graphs and graph algorithms to model and reason about data has seen a huge investment of effort from both theoretical and practical communities over the past decade. This thesis contributes to this research effort in Chapters 3, 4, 5, and 6 by developing graph processing methods and strategies that allow analytics of both static and dynamic graphs to be accelerated on the FPGA.

Chapter 3 (GraphTinker) proposes a solution for the tradeoff associated with data structures for dynamic (or evolving) graphs. Several data structures for dynamic graphs demonstrate high throughput when inserting, deleting, or updating edges to the graphs but at the cost of low throughput when running graph analytics, or vice versa. In this chapter, I propose a data structure for the CPU that allows a compact data representation, where edges within the data structure are packed together with little to no empty slots in-between, and an optimal data structure that requires only a small fraction of the edgelist to be traversed for every edge to be inserted, deleted or updated. The techniques incorporated into GraphTinker allow it to demonstrate superior performance over prior art.

My dissertation shifted gears to the FPGA domain considering the benefits of the FPGA as potential graph accelerators. Chapter 4 (ACTS) investigates the challenge of poor scaling (w.r.t. graph

size) with prior FPGA-based graph accelerators. While current FPGA accelerators demonstrate excellent performance when running analytics on small graphs, their performance can degrade significantly with larger graphs. This is due to a popular adopted strategy called *graph slicing*, which aims to use fast URAMs for random accesses to vertices during processing. With graph slicing, the graph is partitioned (to restructure locality) into several *slices* to stage the vertex properties of each slice in chunks that fit on-chip FPGA memory (i.e., URAM). The consequence of this is a wastage of HBM bandwidth, as several vertex properties can be read more than once from HBM when processing each slice. This problem exacerbates with larger graphs resulting in degradation in throughput. My solution proposes to restructure the locality of messages generated during processing rather than the graph itself. This eliminates this redundancy issue and allows scaling, as no vertex property is read more than once in each graph iteration. The main drawbacks associated with this approach are discussed and tackled in this chapter.

The work in Chapter 5 (Swift) was motivated by the success of ACTS discussed in Chapter 4. Swift attempts to map ACTS in a multi-FPGA environment and address the issue of low communication channel bandwidth between FPGAs. Swift proposes a decoupled asynchronous approach where computation (within FPGAs) and communication (between FPGAs) tacks are asynchronously executed on separate regions of a graph concurrently. The objective is to keep the PCIe communication channel, the HBM, and the on-chip URAM/BRAM of the FPGA busy. While an asynchronous approach to graph processing on FPGAs allows this advantage, it introduces a new set of problems that can cause the (1) degradation of HBM bandwidth efficiency within each FPGA, (2) distortion of the correct flow of execution, causing incorrect results, and (3) displacing the online partitioning strategy of vertex updates, which is a principal contributor to performance in ACTS. Chapter 5 proposes techniques to address each of these concerns to deliver superior performance over prior art.

Chapter 6 revisits dynamic graph processing and proposes a new approach to updating dynamic graphs. The goal was to reduce the gap in throughput between graph updating and graph analytics tasks. Rather than employing the widely known approach of traversing edgelists of a graph to search for and update a matching edge, I propose a new strategy where a group of edge updates

is hashed into a large URAM array, and edges from HBM are streamed across that array to pick their updates at their hashed indexes. This approach serves two important benefits (1) it allows a runtime complexity of $\mathcal{O}(1)$ because there is no need to search through any edgelists. (2) It allows graph updating tasks to be hidden within graph analytics tasks as inflight edges which have been updated can move through the graph processing pipeline. The presence of conflicts is a challenge with this approach, where two or more edge updates can be hashed to the same URAM slot. This challenge is tackled in this chapter.

There are three prominent future directions for my dissertation. The first would be to develop a single synthesized implementation to support multiple algorithms. This would prevent the need to re-synthesize all over again when making changes to the model, or when implementing a new algorithm. The second would be to distill lessons in ACTS to develop near-memory or in-memory accelerator. The third would be to implement caching functionality that exploit locality within graphs.

Above all, this dissertation achieves efficient graph analytics and efficient updating of dynamic graphs by optimizing bandwidth efficiency.

Appendix A

List of Publications

A.1 Publications

1. **W. Jaiyeoba** and K. Skadron, ”GraphTinker: A High Performance Data Structure for Dynamic Graph Processing,” 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, 2019, pp. 1030-1041, doi: 10.1109/IPDPS.2019.00110.

2. **W. Jaiyeoba**, N. Elyasi, C. Choi, and K. Skadron. 2023. ACTS: A Near-Memory FPGA Graph Processing Framework. In Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA ’23). Association for Computing Machinery, New York, NY, USA, 79–89. https://doi.org/10.1145/3543622.3573180

A.2 Planned Publications & Journals

- Accelerated Graph Processing with Multiple FPGAs. In this paper we would propose our ideas for accelerated graph processing using multiple FPGAs as discussed in chapter 5.

- A Dynamic Graph Accelerator For HBM-Enabled FPGAs. In this journal we would propose our ideas for dynamic graph processing as discussed in chapter 6.

A.3 Patents

- **O. Jaiyeoba**, N. Elyasi (Samsung SSI), C. Choi (Samsung SSI) "A technique to convert low locality and random memory accesses into sequential accesses in efficient time" 20210255793, August 19, 2021.
- **O. Jaiyeoba**, N. Elyasi (Samsung SSI), C. Choi (Samsung SSI) "System and method for managing conversion of low-locality data into high-locality data" 11429299, August 30, 2022.

A.4 Awards

- **Outstanding Teaching Assistant Award**, Charles L. Brown Department of Electrical and Computer Engineering, University of Virginia, May 2017

Bibliography

[1] "The fpga architecture," https://www.ni.com/docs/en-US/bundle/labview-nxg-fpga-targets/page/intro-fpga-resources.html, accessed: 2023-03-15.

[2] R. Li, "Pipelined asynchronous high level synthesis for general programs," Ph.D. dissertation, Yale University Graduate School of Arts and Sciences, 2021.

[3] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "Hbm (high bandwidth memory) dram technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, 2017, pp. 1–4.

[4] S. Beamer, K. Asanović, and D. Patterson, "Gail: The graph algorithm iron law," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA¡sup¿3¡/sup¿ '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833179.2833187

[5] D. Ediger, R. Mccoll, J. Riedy, and D. Bader, "Stinger: High performance data structure for streaming graphs," 09 2012, pp. 1–5.

[6] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, pp. 305–324, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:7589970

[7] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Comput. Surv.*, vol. 51, no. 3, jun 2018. [Online]. Available: https://doi.org/10.1145/3199523

[8] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 11 2013.

[9] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," 10 2016, pp. 1–13.

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

[11] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds. Cham: Springer International Publishing, 2014, pp. 451–462.

[12] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proc. VLDB Endow.*, vol. 7, no. 14, p. 1981–1992, oct 2014. [Online]. Available: https://doi.org/10.14778/2733085.2733103

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence, UAI 2010*, 06 2010.

[14] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," 01 2009, p. 48.

[15] M. M. A. P. S. R. D. M. J. A. S. G. V. D. D. N. Sundaram, N. Satish and P. Dubey, "Graphmat: High performance graph analytics made productive." Proceedings of the VLDB Endowment, 2015.

[16] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," 02 2017, pp. 217–226.

[17] X. Chen, H. Tan, Y. Chen, B. He, W. Wong, and D. Chen, "Thundergp: Hls-based graph processing framework on fpgas," in *FPGA 2021 - 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA 2021 - 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Association for Computing Machinery, Inc, Feb. 2021, pp. 69–80, publisher Copyright: © 2021 ACM.; 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2021 ; Conference date: 28-02-2021 Through 02-03-2021.

[18] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. Hoe, J. Martinez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," 05 2014, pp. 25–28.

[19] 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, 2019.

[20] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An fpga framework for edge-centric graph processing," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 69–77. [Online]. Available: https://doi.org/10.1145/3203217.3203233

[21] 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)*, 2016, pp. 103–110.

[22] Y. Zhuo, X.-L. Wu, J. P. Haldar, T. Marin, W. mei W. Hwu, Z.-P. Liang, and B. P. Sutton, "Chapter 44 - using gpus to accelerate advanced mri reconstruction with field inhomogeneity compensation," in *GPU Computing Gems Emerald Edition*, ser. Applications of GPU Computing Series, W. mei W. Hwu, Ed. Boston: Morgan Kaufmann,

2011, pp. 709–722. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123849885000449

[23] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "Hbm (high bandwidth memory) dram technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, 2017, pp. 1–4.

[24] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *European Conference on Computer Systems*, 2012.

[25] G. Feng, X. Meng, and K. Ammar, "Distinger: A distributed graph data structure for massive dynamic graph processing," 10 2015, pp. 1814–1822.

[26] O. Green and D. A. Bader, "custinger: Supporting dynamic graph algorithms for gpus," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–6.

[27] P. Macko, V. Marathe, D. Margo, and M. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," *Proceedings - International Conference on Data Engineering*, vol. 2015, pp. 363–374, 05 2015.

[28] C. Yin, J. Riedy, and D. A. Bader, "A new algorithmic model for graph analysis of streaming data," in *Proceedings of the 14th International Workshop on Mining and Learning with Graphs (MLG), held in conjunction with 24th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, August 2018, pp. 1–8. [Online]. Available: http://www.mlgworkshop.org/2018/papers/MLG2018_paper_23.pdf

[29] "Open addressing," 2023, [Online; accessed 29-September-2012]. [Online]. Available: https://en.wikipedia.org/wiki/Open_addressing

[30] P. Celis, "Robin hood hashing," Ph.D. dissertation, CAN, 1986.

[31] K. Iwabuchi, S. Sallinen, R. Pearce, B. Van Essen, M. Gokhale, and S. Matsuoka, "Towards a distributed large-scale dynamic graph data store," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 892–901.

[32] P. Celis, P.-A. Larson, and J. I. Munro, "Robin hood hashing," in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, 1985, pp. 281–288.

[33] 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)*, 2021, pp. 1–9.

[34] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," 2021. [Online]. Available: https://arxiv.org/abs/2111.12555

[35] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 65–77. [Online]. Available: https://doi.org/10.1145/3490422.3502357

[36] Z.-k. Wang, J. Paul, B. He, and W. Zhang, "Multikernel data partitioning with channel on opencl-based fpgas," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. PP, pp. 1–13, 02 2017.

[37] G. Dai, Y. Chi, Y. Wang, and H. Yang, "Fpgp: Graph processing framework on fpga a case study of breadth-first search," 02 2016, pp. 105–110.

[38] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, "Parallel fpga-based all-pairs shortest-paths in a directed graph," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 2006, pp. 10 pp.–.

[39] N. Engelhardt and H. K.-H. So, "Gravf: A vertex-centric distributed graph processing framework on fpgas," *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, 2016.

[40] M. Delorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. Uribe, T. Knight, and A. Dehon, "Graphstep: A system architecture for sparse-graph algorithms," 05 2006, pp. 143 – 151.

[41] O. Mencer, Z. Huang, and L. Huelsbergen, "Hagar: Efficient multi-context graph processors," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, M. Glesner, P. Zipf, and M. Renovell, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 915–924.

[42] 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," *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 25–28, 2014.

[43] Z. Shao, R. Li, D. Hu, X. Liao, and H. Jin, "Improving performance of graph processing on fpga-dram platform by two-level vertex caching," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 320–329. [Online]. Available: https://doi.org/10.1145/3289602.3293900

[44] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An fpga framework for edge-centric graph processing," ser. CF '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 69–77. [Online]. Available: https://doi.org/10.1145/3203217.3203233

[45] 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)*, 2016, pp. 103–110.

[46] M. Besta, D. Stanojevic, J. Licht, T. Ben-Nun, and T. Hoefler, "Graph processing on fpgas: Taxonomy, survey, challenges," 02 2019.

[47] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 820–831.

[48] "Pagerank." [Online]. Available: https://en.wikipedia.org/wiki/PageRank

[49] "Hyperlink induced topic search (hits)." [Online]. Available: https://en.wikipedia.org/wiki/HITS_algorithm

[50] "Single source shortest path (sssp)." [Online]. Available: https://en.wikipedia.org/wiki/Shortest_path_problem

[51] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. Owens, "Gunrock: a high-performance graph processing library on the gpu," 02 2016, pp. 1–12.

[52] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," vol. 6, 04 2004.

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

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

[55] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 08 2015.

[56] 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*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 111–117. [Online]. Available: https://doi.org/10.1145/2847263.2847337

[57] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," 06 2014.

[58] N. G. A. L. n. NVIDIA Corporation. (2020) https://developer.nvidia.com/nvgraph. [Online]. Available: https://developer.nvidia.com/nvgraph

[59] W. Zhong, Y. Cao, J. Li, J. Sun, and H. Chen, "Specialization or generalization: A study on breadth-first graph traversal on gpus," 12 2017, pp. 294–301.

[60] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," 10 2012, pp. 31–46.

[61] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc," 08 2013, pp. 77–85.

[62] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," 10 2012, pp. 17–30.

[63] 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.

[64] W. Jaiyeoba, N. Elyasi, C. Choi, and K. Skadron, "Acts: A near-memory fpga graph processing framework," 02 2023, pp. 79–89.