Graphitron: A Domain Specific Language for FPGA-Based Graph Processing Accelerator Generation

Xinmiao Zhang
zhangxinmiao20s@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China

Zheng Feng
fengzheng@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China

Shengwen Liang
liangshengwen@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China

Xinyu Chen
xinyuchen@hkust-gz.edu.cn
Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, Guangdong, China

Lei Zhang
zlei@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China

Cheng Liu*
liucheng@ict.ac.cn
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China

Abstract

Due to hardware customization capabilities, FPGA-based graph processing accelerators achieve significantly higher energy efficiency than many general-purpose computing engines. However, designing these accelerators remains a substantial challenge for high-level users. To overcome the programming barrier, FPGA-based accelerator design frameworks on top of generic graph processing programming models have been developed to automate accelerator generation through pre-built templates. However, they often tightly couple graph processing algorithms, programming models and processing paradigms, and accelerator architectures, which severely limits the expression scope of the algorithms and may also restrict the performance when the generated accelerators fail to suit dynamic processing patterns of the graph processing algorithms.

In this work, we propose Graphitron, a domain-specific language (DSL) that enables the automatic generation of FPGA-based graph processing accelerators without engaging with the complexities of low-level FPGA designs. Graphitron defines vertices and edges as primitive data types and enables users to implement graph processing algorithms by performing various functionalities on top of these primitive data, which greatly eases the algorithm descriptions for high-level users. During compilation, the graph processing functions are naturally classified into either a vertex-centric processing paradigm or an edge-centric processing paradigm according to the target data types, enabling the generation of accelerator kernels of different characteristics. In addition, because of the explicit binding between the graph processing functions and the data types, the Graphitron compiler can automatically infer the computing and memory access patterns of each processing function within graph processing algorithms and apply corresponding hardware optimizations such as pipelining, data shuffling, and caching. Basically, graph semantic information can be utilized to guide algorithm-specific customization of resulting accelerators for higher performance. Our experiments show that Graphitron can generate accelerators for a broader range of graph processing algorithms than prior template-based generation frameworks. Moreover, the accelerators produced by Graphitron achieve performance comparable to, and in some cases exceeding, that of existing frameworks when the combined programming paradigms are beneficial from an algorithmic perspective.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Computer systems organization → Embedded systems**; • **Hardware → Hardware accelerators**.

Keywords: FPGA, Graph Processing, Accelerator Generation, Domain-specific Language, High-level Synthesis

ACM Reference Format:
Xinmiao Zhang, Zheng Feng, Shengwen Liang, Xinyu Chen, Lei Zhang, and Cheng Liu. 2025. Graphitron: A Domain Specific Language for FPGA-Based Graph Processing Accelerator Generation. In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*

*Corresponding Author

This work is licensed under a Creative Commons Attribution 4.0 International License.

LCTES '25, Seoul, Republic of Korea

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1921-9/25/06
https://doi.org/10.1145/3735452.3735533

(LCTES '25), June 16–17, 2025, Seoul, Republic of Korea. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3735452.3735533

1 Introduction

To meet the growing demand for graph processing services, such as web page ranking [27], social network analytics [35], and financial data mining [12], major Internet giants have invested in developing their own graph processing platforms for modern data centers. Examples include Amazon's Neptune [2], Google's Pregel [24] and Alibaba's GraphScope [13]. As a result, graph processing has emerged as a core building block of modern computing platforms.

Despite extensive work exploring graph processing optimizations through software-based techniques [20, 24, 26, 30, 32, 38], general-purpose computing engines suffer from low energy efficiency due to poor hardware utilization, which stems from the mismatch between processor architectures and the irregular characteristics of graph processing. For example, CPU cores fetch data at the cache-line granularity, but frequent and irregular byte-level accesses in graph processing significantly degrade cache efficiency [23]. Likewise, GPUs organize threads at the warp granularity, which can lead to severe workload imbalances when processing graphs with power-law degree distribution, thereby reducing thread-level parallelism [14]. In contrast, FPGAs with tailored hardware designs can offer both high performance and energy efficiency in graph processing [1, 10, 15], making them a promising infrastructure for modern data centers. Nevertheless, designing efficient FPGA accelerators for diverse graph algorithms remains challenging for high-level users, particularly those without expertise in FPGAs.

Many prior graph processing frameworks on FPGAs [7, 8, 11, 16, 17, 31] have been proposed to lower the programming barrier for high-level users by providing easy-to-use programming models compatible to software-based ones. These frameworks typically rely on pre-built accelerator templates [7, 8, 11, 17, 31] or hardware overlays [16] that incorporate various hardware optimizations for graph processing. However, the tight coupling of the graph processing algorithms, programming models and paradigms, and accelerator architectures can limit algorithmic flexibility and lead to suboptimal performance of the generated accelerators. Specifically, only algorithms compatible with the provided programming models, such as the Gather-Apply-Scatter (GAS) model, can be applied to the generation frameworks. The fixed programming model limits the processing manners and workflows to specific graph processing algorithms and fails to support many practical occasions such as attributed graph processing. Moreover, the pre-built accelerator template typically adopts a fixed graph processing paradigm (*e.g.*, edge-centric or vertex-centric) to integrate specialized hardware optimizations. As a result, these approaches may perform poorly for some of the algorithms with distinct preferences. For example, an edge-centric paradigm is effective for algorithms such as PageRank, where all vertices stay active over the different iterations. However, it struggles with BFS, which only activates a subset of vertices, resulting in considerable edge traversal redundancy using an edge-centric paradigm. These limitations underscore the need for a more flexible method of generating FPGA-based graph processing accelerators, which can concurrently smooth the learning curve, supports diverse graph processing algorithms, and enables multiple processing paradigms. Although high-level synthesis tools have made hardware design more accessible, they still demand extensive manual tuning and require intricate low-level hardware expertise, making FPGA acceleration difficult for algorithm developers. Inspired by GraphIt [4, 38], a DSL that decouples graph algorithms from their optimization strategies for CPUs and GPUs, we seek to bring this idea to FPGA acceleration, enabling flexible algorithm expression while ensuring efficient hardware generation.

The challenge in designing DSL lies in balancing abstraction with customization. On the one hand, the DSL should allow users to describe graph algorithms without dealing with low-level FPGA optimizations. On the other hand, removing manual tuning requires the compiler to infer and apply optimizations automatically, which is particularly difficult given the diverse computation and memory access patterns of graph algorithms. Unlike CPUs and GPUs, where the hardware behavior is largely uniform, FPGAs require precise control over execution and data flow, making it challenging to design a general yet efficient optimization strategy.

To address this, we introduce Graphitron, a domain-specific language (DSL) for FPGA-based graph processing accelerators. Graphitron provides a high-level abstraction by treating vertices and edges as fundamental data types and allowing users to define various algorithmic operators. The compiler automatically analyzes algorithmic structures, maps them to either vertex-centric or edge-centric processing paradigms, and applies a suite of optimizations such as pipelining, caching, and data shuffling. By eliminating the need for low-level hardware expertise, Graphitron significantly reduces development effort while achieving high performance across diverse graph workloads. Our evaluation shows that it not only simplifies accelerator generation but also outperforms existing FPGA-based frameworks in some representative graph processing algorithms like BFS and SSSP. Our contributions can be summarized as follows.

- We present Graphitron, a domain-specific language designed for agile development of FPGA-based graph processing accelerators. Graphitron defines vertices and edges as fundamental data types, and allows users to describe graph algorithms flexibly by customizing

operators on the data types and arranging their workflow, without exposing them to the complexities of low-level FPGA designs.
- We develop a compiler that generates efficient, end-to-end FPGA accelerators from Graphitron specifications. It automatically maps vertex and edge operators to vertex-centric or edge-centric processing paradigms, then infers and integrates a suite of specialized hardware optimizations, including pipelining, caching, and shuffling, into each processing kernel to produce high-performance accelerators.
- Graphitron can generate accelerators for a wider range of graph processing algorithms with less effort than state-of-the-art FPGA-based graph processing frameworks. Furthermore, the accelerators produced by Graphitron achieve performance on par with, and in some cases exceeding, existing frameworks, thanks to the performance advantages from both vertex-centric and edge-centric processing paradigms.

2 Background and Related Work

In this section, we firstly introduce two typical processing paradigms for graph processing and highlight their distinct characteristics. We then survey related work, identifying their limitations in flexibility and performance, and then motivate a DSL for FPGA-based graph processing accelerators.

2.1 Graph Processing Paradigms

Given a graph $G = (V, E)$, where V is a set of vertices and E includes all the edges connecting the vertices, a graph processing task reveals hidden structures or features of graph G by running a specific graph processing algorithm. Graph processing systems are designed to achieve high throughput for various graph processing algorithms. Two widely adopted processing paradigms in these systems are vertex-centric processing (VCP) and edge-centric processing (ECP). The VCP paradigm operates on vertices, which either push updates to their neighbors or pull updates from them along edges. Due to its simplicity in programming and ease of filtering inactive vertices, VCP is commonly employed in graph processing systems. However, the random adjacency among vertices in VCP leads to frequent random accesses to vertex properties, significantly stressing the cache system. To address this limitation, the ECP paradigm processes edges in a streaming manner: it sequentially streams edges to generate updates, then streams these updates to gather and apply them to vertices. Thus, ECP effectively reduces random memory accesses, alleviating cache-system bottlenecks. Nevertheless, ECP struggles to efficiently skip streaming edges associated with inactive vertices, resulting in redundant computations and reduced efficiency, particularly in algorithms where only part of vertices are active, such as breadth-first search (BFS) and single-source shortest path (SSSP).

2.2 Related Work

Graph processing is extensively employed in big data applications, stimulating advancements in hardware accelerator designs [7, 15, 16, 19, 31, 34, 39]. Among them, FPGA-based implementations stand out due to their superior performance and energy efficiency. Key performance bottlenecks in graph processing, such as excessive random memory accesses and low data utilization, have driven extensive optimizations on FPGA-based accelerators. FPGP [9] and ForeGraph [10] leveraged graph partitioning and optimized data placement to enhance the utilization of on-chip BRAMs, significantly improving memory access efficiency. Cygraph [3] introduced a customized CSR format tailored for the BFS algorithm to maximize memory bandwidth utilization on FPGAs. FPGA-based graph accelerators have also been integrated into heterogeneous architectures. Zhou et al.[40] utilized a CPU-FPGA heterogeneous architecture, distributing vertex-centric and edge-centric tasks strategically between CPUs and FPGAs to leverage the strengths of both computing engines. GraFBoost[18] and ExtraV [21] embedded FPGA-based graph processing accelerators within the programmable logic region of SSDs, significantly reducing data transfer overhead of disk-based graph processing. Despite these advancements significantly improving the performance and energy efficiency of FPGA-based graph processing accelerators, developing specialized accelerators for diverse graph algorithms on FPGAs remains notably more challenging and less productive compared to that on general-purpose computing engines, such as GPUs or CPUs. This is especially true for high-level users lacking extensive expertise in FPGA design.

Over the years, design productivity challenges in FPGA have been addressed by both industry [37] and academia [29, 36]. High-Level Synthesis (HLS)[37] bridges the gap between high-level programming languages and Hardware Description Languages (HDL) by abstracting hardware complexities, thereby reducing programming barriers. Despite these advancements, efficiently implementing algorithms with intensive control logic and irregular memory accesses such as graph processing remains challenging. To mitigate this complexity, FPGA overlays[29] abstract specific functionalities of underlying configurable hardware accelerators into interfaces accessible from high-level languages, facilitating more efficient FPGA utilization. These overlays have demonstrated notable performance in domains such as deep learning and dataflow graphs. However, their flexibility is limited when applied to graph processing, primarily due to the diverse operations and varying computational patterns exhibited by different graph algorithms.

Graph Processing Frameworks on FPGAs. Several FPGA-based graph processing frameworks [7, 8, 11, 16, 17, 28, 31] have been proposed to provide software-like programming models, enabling users to customize FPGA-based graph accelerators. As summarized in Table 1, existing frameworks

offer user-friendly APIs and leverage predefined accelerator templates with fixed processing paradigm to generate FPGA-based accelerators. For example, ThunderGP [8] supports diverse graph processing algorithms through the Gather-Apply-Scatter (GAS) programming model, and employs an HLS-based template with edge-centric paradigm to generate efficient accelerators. However, reliance on programming models inherently limits the scope of graph processing algorithms. Moreover, built-in templates typically adopt a single processing paradigm, restricting their performance across varying graph workloads. For instance, ECP and VCP paradigms are respectively suited to BFS iterations with by more or less active vertices. Additionally, fixed workflow and processing paradigm in the template constrain the design space exploration, limiting the performance potential on FPGA. Therefore, a novel methodology is needed to enhance the applicability, flexibility, and efficiency in the agile development of FPGA-based graph accelerator.

DSLs for Graph Processing. GraphIt [4, 38] is a domain-specific language (DSL) designed to generate efficient software implementations of graph processing algorithms on CPUs and GPUs. GraphIt provides an algorithm language and a scheduling language, enabling users to effortlessly describe graph algorithms and flexibly customize optimization strategies. It supports both VCP and ECP paradigms, allowing flexible orchestration of their workflows. Inspired by GraphIt, we propose Graphitron, a DSL tailored for FPGA-based graph processing accelerators. Graphitron adopts programming models similar to GraphIt's algorithm language, providing users with flexible descriptions of graph processing algorithms. Unlike GraphIt which requires users to explicitly specify optimizations, Graphitron's compiler automatically infers a suite of hardware optimizations and integrates them into HLS-based accelerators. Compared to existing graph processing frameworks on FPGA, Graphitron supports both ECP and VCP paradigms, and also flexible orchestration across different processing kernels.

Table 1. A survey of existing graph processing frameworks on FPGA and graph processing DSLs.

Framework / DSL	Program. Model	Paradigms	Output
Graphlily [16]	Linear Algebra	ECP	HLS
ReGraph [7]	GAS	ECP	HLS
ACTS [17]	GAS	ECP	HLS
GraFlex [31]	Scatter-Gather	ECP	HLS
GraphScale [11]	Vertex-centric	VCP	HDL
ThunderGP [8]	GAS	ECP	HLS
GraphIt [38]	Algorithm and scheduling languages	ECP+VCP	C++ (CPU or GPU)
Graphitron(Ours)	**Vertex and Edge Operators**	**ECP+VCP**	**HLS**

3 Graphitron Overview

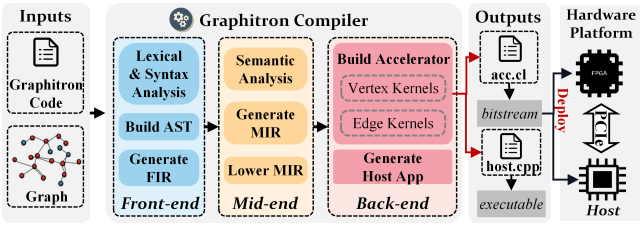

Figure 1. Overview of Graphitron.

Figure 1 illustrates the development workflow of Graphitron. From a high-level user's perspective, Graphitron is not constrained by rigid programming models typical of FPGA-based graph processing frameworks. Instead, it introduces a self-developed syntax serving as its programming model, enabling flexible algorithm descriptions by customizing vertex and edge operators and orchestrating their workflows through Graphitron code. This approach allows users to focus exclusively on algorithm design without needing detailed knowledge of FPGA-specific hardware architectures.

During the compilation stage, the Graphitron compiler accepts Graphitron code and graph data as input, producing high-performance FPGA accelerators along with the corresponding host program through three standard compilation phases: front-end, middle-end, and back-end. The front-end ensures code correctness by performing rigorous grammar checks, thus resolving ambiguous expression scope issues commonly encountered in FPGA-based graph processing frameworks. Subsequently, the middle-end conducts semantic analysis, initially mapping vertex and edge operators into Vertex-Centric Processing (VCP) and Edge-Centric Processing (ECP) paradigms. Since explicit hardware optimization directives are not required from algorithm designers, the compiler automatically extracts semantic information from operators to infer suitable hardware optimizations, annotating these optimizations as tags in the intermediate representation (IR). For example, the compiler implements memory access optimizations to improve execution efficiency, such as caching strategies for random vertex access and burst accesses for sequential edge streaming. Additionally, it explores loop-level parallelization techniques, including loop pipelining and unrolling, for further parallelism enhancement. In the back-end stage, the Graphitron compiler integrates these hardware optimizations with graph algorithm customizations to generate graph-processing accelerators capable of competitive performance. Rather than directly producing RTL code, the compiler generates Xilinx OpenCL code, abstracting away low-level platform-specific details and simplifying the compilation process. Moreover, the compiler generates the host program responsible for workflow control and FPGA device management. More comprehensive details about the compiler can be found in Section 4.

Table 2. APIs to define operators on `vertexset` and `edgeset` in Graphitron.

Vertexset APIs	Return Type	Description
`size``()`	int	Returns the size of the vertexset.
`init``(func vp_func)`	none	Applies `vp_func(vertex)` to each vertex to initialize vertex-related graph properties.
`process``(func vp_func)`	none	Applies `vp_func(vertex)` to each vertex to process vertex-related graph operations.
Edgeset APIs	**Return Type**	**Description**
`size``()`	int	Returns the size of the edgeset.
`init``(func ep_func)`	none	Applies `ep_func(edge)` to each edge to initialize edge-related graph properties.
`process``(func ep_func)`	none	Applies `ep_func(edge)` to each edge to process edge-related graph properties.

During execution, the host program deploys the accelerator bitstream onto the FPGA, loads the specified graph dataset, and partitions large graph data to ensure each partition fits into the FPGA's global memory for preprocessing. In each processing iteration, the host program transfers necessary data to the FPGA's global memory and invokes FPGA computation in a push-button manner accordingly.

3.1 Syntax Definition

Graphitron is an object-oriented programming language designed specifically for graph processing on FPGA. It abstracts the fundamental graph elements, vertices and edges, as basic data types used to instantiate graph objects. User-defined operators for these objects describe the computational steps involved in graph processing algorithms. This abstraction enables Graphitron to flexibly support VCP, ECP, and hybrid graph processing paradigms through user customization. To illustrate the detailed syntax of Graphitron, we take a Graphitron code of BFS algorithm implemented in the ECP paradigm as example, which is shown in Listing 1.

Data Types. The fundamental data types in Graphitron are `Vertex` and `Edge`, representing vertex and edge entities in the graph, respectively. As shown in Line 1-2 of Listing 1, user should specify the graph file and construct the `vertexsets` and `edgesets` at the beginning of Graphitron code, which contains all the vertices and edges of the graph. The data type of elements in these sets is denoted using `"{}"`. Edges in the graph can be either weighted or unweighted, as further specified by `"()"`. An unweighted edge is represented by a vertex pair (`Vertex, Vertex`), while a weighted edge which includes an additional weight parameter can represented as (`Vertex, Vertex, WeightType`). Additionally, users can define properties of vertices or edges by using `"{}"` to specify its owner and `"()"` to specify its type as illustrated in Lines 3-5 of Listing 1.

User-defined Operators. Graphitron includes a suite of APIs to operate on the `vertexsets` and `edgesets` as listed in Table 2. These APIs are derived from computational methodologies outlined in [25], including **`size()`**, **`init(func)`**, and **`process(func)`**. In this way, users can customize the initialization and processing of graph processing algorithms by defining `func` function for **`init(·)`** and **`process(·)`** APIs. Graphitron's user-defined functions support fundamental mathematical operations, including integer and floating-point computations, as well as common reduction operations such as `sum`, `min`, and `max`.

We illustrate the usage of Graphitron with an example of a BFS implementation following the edge-centric processing paradigm, shown in Listing 1. The algorithm starts with an initialization operation invoked via the **`init()`** API (Line 26), which internally calls the `reset()` function (Lines 8-10) to set initial values for each vertex's property in the `vertexsets` `vertices`. The **`process()`** API, demonstrated in Lines 30-32, accepts user-defined functions as input to specify the operations executed on `vertices` or `edges` during graph processing. Detailed implementations of the functions `VertexUpdate()`, `VertexApply()`, and `EdgeTraversal()` are presented in Lines 11-24. Specifically, `EdgeTraversal()` iterates over each edge to generate updates for the target vertices, `VertexUpdate()` aggregates these updates for the new vertex properties, and `VertexApply()` finalizes these updates to the existing vertex properties. This illustrates the characteristics of the edge-centric processing paradigm.

Discussion. Graphitron can flexibly support VCP, ECP and their hybric processing paradigms by customizing traversal functions to edges and vertices, and scheduling their workflows. For instance, Graphitron code for BFS with hybrid VCP and ECP paradigm is shown in Listing 2, which implements seamless transition between two paradigms based on frontier size (Line 10).

The implementation in Graphitron of graph processing algorithms which can be expressed by GAS model have a large fraction of similarities. Specifically, users need to firstly choose vertex-centric, edge-centric or hybrid processing paradigms to choose a main function template, then customize the `EdgeTraversal` and `VertexTraversal` interfaces for graph traversal, `VertexUpdate` for gathering updates, `VertexApply` for applying updates to vertex properties. Therefore, the programming barrier of Graphitron remains low while improving its flexibility.

```
const edges: edgeset{Edge}(Vertex, Vertex)=load(argv[1]);
const vertices: vertexset{Vertex} = edges.getVertices();
old_level: vector{Vertex}(int); % properties
new_level: vector{Vertex}(int);
tuple: vector{Vertex}(int);
frontier_size: int = 1;
level: int = 1;
func reset(v: Vertex)
    old_level[v] = new_level[v] = -1;
end
func EdgeTraversal(src: Vertex, dst: Vertex)
    if (old_level[src] == level)
        tuple[dst] min= level+1;
    end
end
func VertexUpdate(v: Vertex)
    if (tuple[v]==(level+1)&&(old_level[v]==-1))
        new_level[v] = tuple[v];
        frontier_size = frontier_size+1;
    end
end
func VertexApply(v: Vertex)
    old_level[v] = new_level[v];
end
func main()
    vertices.init(reset);  % Initialization
    old_level[1] = new_level[1] = 1;
    while(frontier_size>0)
        frontier_size = 0;
        edges.process(EdgeTraversal); % edge traversal
        vertices.process(VertexUpdate);
        vertices.process(VertexApply);
        level += 1;
    end
end
```

Listing 1. Graphitron code for BFS of edge-centric processing paradigm.

```
func VertexTraversal(v: Vertex)
    if (old_level[v] == level)
        for ngh in v.getNeighbors()
            tuple[ngh] min= level + 1;
        end
    end
end
func main()
    ...
    if (frontier_size<0.05*vertices.size())
        vertices.process(VertexTraversal); % VCP
    else
        edges.process(EdgeTraversal); % ECP
    end
    ...
end
```

Listing 2. Graphitron code for BFS of hybrid vertex-centric and edge-centric paradigms.

4 Graphitron Compiler

The Graphitron compiler accepts Graphitron code and graph data as input and generates FPGA accelerators through the front-end, middle-end, and back-end stages.

4.1 Front-end

The front-end of the Graphitron compiler performs lexical analysis and syntax parsing on the Graphitron code, constructs an *Abstract Syntax Tree* (AST), and generates the *Front-end Intermediate Representation* (FIR). Our front-end implementation involves a self-developed Lexer and Parser tailored specifically for handling graph-centric abstractions (vertex, edge, property), thus directly mapping these concepts to FPGA hardware optimization tags without excessive pragmas or hints. Although currently independent from compiler frameworks like LLVM or MLIR, Graphitron's semantic abstractions are designed with portability, making it feasible to compile into an MLIR in the future. This integration potential highlights Graphitron's capability to connect seamlessly with broader compiler ecosystems.

4.2 Middle-end

The middle-end of the compiler performs semantic analysis based on FIR to create *Middle-end Intermediate Representation* (MIR), and then refines MIR to map the structure of target FPGA and adds hardware optimizations as tags into MIR.

Firstly, the AST generated by the front-end has inherent limitations, as each node can access information only from its immediate children. To address this, the compiler globally traverses the FIR nodes within the AST, performing semantic analysis and establishing enriched MIR contexts. This approach allows the Graphitron compiler to comprehensively interpret the programmer's intent. For instance, given the statement `edges.process(EdgeTraversal)`, FIR can only recognize a reference to the `EdgeTraversal` function via the **`process`** API call. In contrast, MIR supplies additional semantic details about the function, including parameter types, function bodies, and return types.

Furthermore, the compiler identifies objects within Graphitron code and allocates appropriate memory space on the target hardware for these objects and their associated properties. Specifically, it determines the data types and sizes of graph properties used within the graph-processing algorithms to estimate the required memory resources accurately. The compiler then automatically assigns memory units and channel IDs to these graph properties, enabling the FPGA to efficiently allocate memory and manage pointers for transferring graph data to hardware kernels. For large graphs whose sizes exceed the available FPGA memory capacity, the compiler first devises a memory allocation strategy for graph properties to maximize FPGA memory utilization. Subsequently, it formulates a graph-partitioning scheme to ensure each partition fits within the allocated FPGA memory space.

Graphitron supports both edge-centric and vertex-centric processing paradigms. Therefore, for graph preprocessing, the compiler generates two distinct graph data formats in the host program: it either vertically partitions the graph and stores it in standard coordinate format (COO) for edge-centric processing (ECP), or horizontally partitions the graph and stores it in compressed sparse row format (CSR) for vertex-centric processing (VCP). The compiler analyzes the equal vertex size of each partition based on FPGA memory capacity and determines the appropriate data format according to operators used in the algorithm. For example, when the compiler detects that the `func` argument in edge operator API **`process(·)`** accesses vertex data, it classifies the function as edge-centric processing and accordingly generates preprocessing code for COO format in the host program.

Finally, the compiler infers a suite of hardware optimizations, and adds tags to MIRs related to the accelerator kernel, which are collected by traversing the children of **`process`**`(·)`. These hardware optimizations include memory optimizations for graph properties, and parallel and pipelining optimization for `function` kernels. We take `edges.`**`process`**`(EdgeTraversal)` in Listing 1 as an example, and the identified MIRs are shown in Table 3. The compiler automatically unrolls loops of `func` to maximize data parallelism, and pipelines instructions to explore the instruction-level parallelism. Since the `func` frequently accesses data from off-chip memory, memory bandwidth becomes the throughput bottleneck. The `unroll` pragma can mitigate this issue by unfolding loops into multiple parallel logic paths, enabling simultaneous processing of multiple data streams and thus improving overall throughput. To determine the optimal unroll factor, the compiler sets the corresponding parameters as the number of memory channels where the primary data structure of the `process` API (e.g. `edges`) is allocated. For pipeline optimization, we rely on the HLS tool to automatically determine the minimal feasible initiation interval, rather than manually specifying it by the compiler.

Different graph properties exhibit distinct memory access patterns. Properties associated with edges are typically accessed sequentially, whereas vertex-related properties predominantly involve random access patterns. Consequently, we adopt tailored memory access optimizations based on these differences. To maximize bandwidth utilization, we apply burst read and write operations (*BR* or *BW*) for graph properties stored in FPGA memory. For vertex-related properties characterized by frequent random accesses, such as `old_level` and `tuple`, we allocate a dedicated caching module using FPGA's UltraRAM to enhance throughput of random accesses. Additionally, for vertex properties holding random writes, such as `tuple`, we implement a shuffling module before burst writing to further improve throughput.

Table 3. An example of hardware optimization tags for MIRs in `edges.`**`process`**`(EdgeTraversal)`. *BR* and *BW* indicate burst read and write, respectively.

MIRs	Memory Tags	Parallel Tags	Pipeline Tags
`func EdgeTraversal()`	<⊥, ⊥, ⊥>	<*Unroll=#Channel*>	<*yes*>
`edgeset edges`	<*BR*,⊥,⊥>	<⊥>	<⊥>
`vector{Vertex} old_level`	<*BR, cache*, ⊥>	<⊥>	<⊥>
`vector{Vertex} tuple`	<*BW,cache,shuffle*>	<⊥>	<⊥>

4.3 Back-end

The back-end primarily generates hardware accelerators to the target FPGA platforms, and generates corresponding host program for data movement and device management.

Figure 2 illustrates the overall framework of the generated accelerator by Graphitron compiler's back-end. Instead of relying on a built-in accelerator template, it employs a generic architecture comprising various basic graph-processing operators designed to accommodate a broad spectrum of graph-processing algorithms. These operators are subsequently translated into hardware components implemented using Xilinx OpenCL. The workflow is detailed as follows: ❶ Graph properties are first streamed from FPGA memory (HBM in this example) via the *Burst Read* module, and then directed into the *cache* module for vertex or edge processing. Note that the cache module may be omitted in the case of edge operators. ❷ Within the *Edge Process* module, edges are read in COO format, with edge weights selectively loaded depending on whether edge weights are required. The *Edge Operation* module then produces an update stream comprising destination vertex indices and corresponding update values. Similarly, in the *Vertex Process* module, vertex properties are cached, and active vertices (frontiers) are identified through the *Frontier Check* module. Neighboring edges of these frontiers in CSR format are burst-read from FPGA memory. Subsequently, the *Vertex Operation* module outputs an update stream containing destination vertex indices and their associated update values according to user-defined computing logic. ❸ If the output property is not marked as *shuffle*, the update stream is directly written back to FPGA memory using the *Burst Write* module, as exemplified by the `old_level` property in the `VertexApply` function, and this computing kernel finished its computation. ❹ Conversely, if the output property is designated as *shuffle*, indicating that the stream involves random writes, the stream is further routed to the *Stream Duplicate* module, such as `tuple` property in the `EdgeTraversal` function, and this computing kernel goes to step ❺. ❺ The update stream is duplicated, sliced and distributed across multiple Processing Element (PE) units. ❻ Within each PE unit, the *Shuffle* module reorganizes the update stream and filters out unnecessary data. Subsequently, conflicts in updates are resolved by the *RAW* module, and redundant updates are aggregated by the *Reduce* module. The resulting updates are then stored into the destination graph properties cached in the on-chip URAM.

❼ Finally, the updated graph properties stored sequentially in the URAM cache are written back to HBM via the *Burst Write* module.

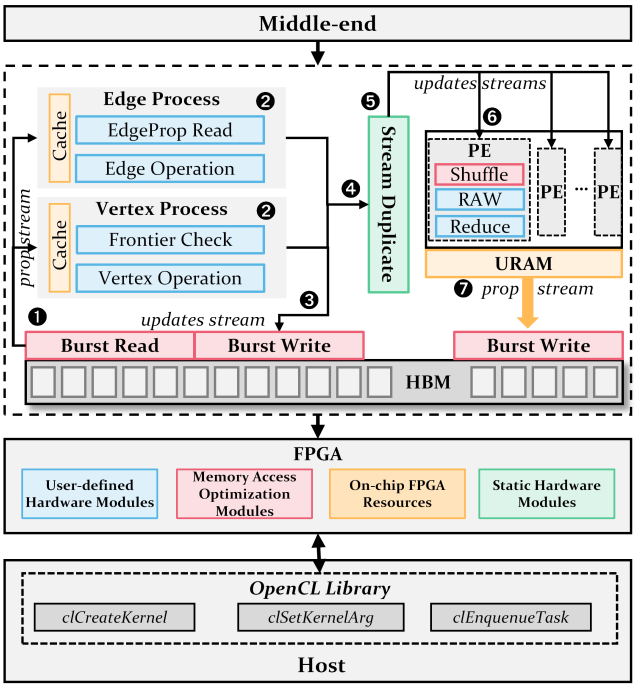

Figure 2. Back-end Framework of Graphitron compiler

Since algorithm-specific optimizations would limit Graphitron's flexibility, we focus on generic hardware optimizations, such as pipelining, unrolling, and memory optimizations in the compiler back-end to enhance the performance of generated accelerator while maintaining Graphitron's flexibility.

Pipelining and Loop Unrolling. Pipelining and loop unrolling are common hardware optimization techniques that achieve instruction-level parallelism and enhance spatial parallelism, respectively. However, high-level code often contains data conflicts that impede effective pipelining or loop unrolling. For example, Listing 3 demonstrates a typical read-write conflict involving the variable *SP* in the single-source shortest path algorithm. To resolve such conflicts, the compiler's middle-end automatically eliminates data conflicts by introducing temporary variables and auxiliary functions to handle intermediate computations. Listing 4 illustrates the modified code, where the conflict is resolved by introducing a temporary variable `tmp` and a separate function `sssp0()` dedicated to assigning `tmp`.

Memory Access Optimizations. Graphitron incorporates several optimizations aimed at improving memory

```
1 func sssp(src:Vertex,dst:Vertex,weight:int)
2     SP[dst] min= (SP[src]+weight);
3 end
```

Listing 3. The original function of SSSP algorithm.

```
1 func sssp0(v:Vertex)
2     tmp[v] = SP[v];
3 end
4 func sssp1(src:Vertex,dst:Vertex,weight:int)
5     SP[dst] min= (tmp[src]+weight);
6 end
```

Listing 4. The decoupled function of SSSP algorithm.

access efficiency, which is crucial for enhancing the performance of graph-processing accelerators. The *Burst Read-/Write* module optimizes memory bandwidth utilization of sequential access to FPGA memory. To accelerate random memory accesses, we implemented a dedicated on-chip caching module, significantly improving data reuse efficiency. Additionally, we employ a data shuffling module [6] to convert random data writes into sequential writes, facilitating conflict-free operations and efficiently distributing sequential writes across multiple PEs targeting different memory banks for higher bandwidth. In this way, we improve the bandwidth of random memory accesses to FPGA memory.

4.4 Host Program and System Integration

Graphitron provides an end-to-end framework for graph processing on a hybrid CPU-FPGA architecture. It achieves this by automatically generating a host program responsible for graph preprocessing, accelerator management, and coordinating data transfers between the CPU and FPGA. The system integration is facilitated by leveraging the OpenCL framework, specifically utilizing Xilinx Runtime (XRT) APIs, such as *clEnqueueMigrateMemObjects*, *clSetKernelArg*, and *clEnqueueTask*, to efficiently manage data transfers and accelerator execution from the CPU side. Additionally, Graphitron introduces implicit programming interfaces for tasks such as graph loading, partitioning, and data migration between CPU and FPGA. These interfaces, while essential for seamless system integration, remain transparent to the developers. In summary, these supporting interfaces enable straightforward deployment of diverse hardware accelerators, significantly enhancing Graphitron's adaptability and flexibility.

5 Evaluation

We evaluate Graphitron from three perspectives: (1) **Performance:** We compare Graphitron's performance against ThunderGP, a state-of-the-art graph processing framework on FPGA. (2) **Flexibility:** We demonstrate Graphitron's

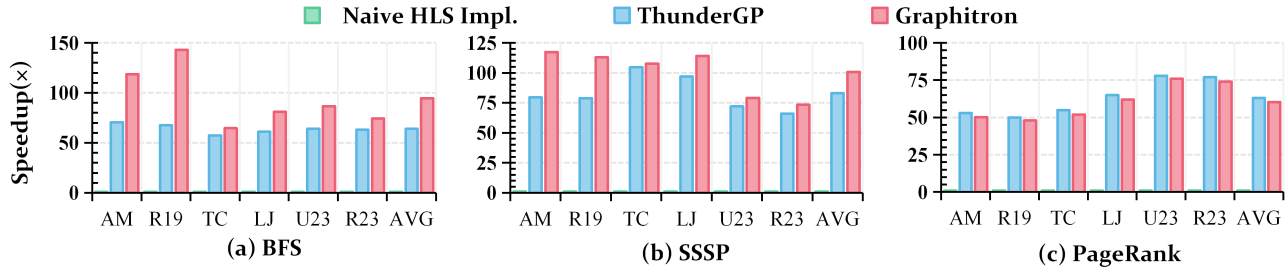

Figure 3. Speedups to naïve HLS implementation of ThunderGP and Graphitron when running **BFS**, **SSSP** and **PageRank**.

capability to implement efficient accelerators for graph-processing algorithms that cannot be expressed using existing graph processing frameworks on FPGA. (3) **Productivity:** We assess Graphitron's programming complexity compared to ThunderGP to highlight productivity benefits.

5.1 Experiment Setup

Testbed. We implement the graph processing accelerators on an AMD Xilinx Alveo U280 FPGA board, which features 8 GB of HBM2 memory capable of accessing 32 HBM pseudo-channels through AXI3 interfaces. The synthesis, placement, routing, and simulation of the graph processing accelerators are conducted using the AMD Xilinx Vitis 2019.2 suite on a CentOS Linux 7 system. The host server is powered by an Intel Xeon E5-2680 V2 CPU and equipped with 128 GB of DDR3 DRAM as main memory. The frequency of all the implementations is set to 200MHz for comparison.

Graph Datasets. We list all the tested datasets in Table 4, including social networks, web link and synthetic graphs. Among them, *U23* is characterized by a uniform degree distribution.

Table 4. Graph datasets.

Graph Dataset	Abbr.	$\|V\|$	$\|E\|$	Type
Amazon0601 [22]	*AM*	403K	3.4M	Social Network
rmat-19-32 [5]	*R19*	524K	16.8M	Synthetic Graph
wiki-topcats [22]	*TC*	1.8M	28.5M	Web Link
soc-LiveJournal [22]	*LJ*	4.8M	69.0M	Social Network
Uniform-23-16 [5]	*U23*	8.4M	134.2M	Synthetic Graph
Rmat-23-16 [5]	*R23*	8.4M	134.2M	Synthetic Graph

Graph Processing Algorithms. We implement a diverse set of graph processing algorithms for evaluation, including three classical algorithms: **PageRank**, Breadth-First Search (**BFS**), and Single-Source Shortest Path (**SSSP**). For BFS and SSSP, Graphitron implements a hybrid vertex-centric and edge-centric processing paradigms, whereas it uses an edge-centric processing paradigm for PageRank. Besides, we include two emerging graph processing algorithms, including Personalized PageRank (**PPR**) for recommendation system and Graph Attention (**GATN**), a key component of graph attention neural network [33]. To ensure clarity, the pseudo codes of **PPR** and **GATN** are detailed in Algorithm 1 and Algorithm 4.

Algorithm 1: Personalized PageRank(**PPR**).

$PR_{old} \leftarrow \{0, score_{init}, 0, \ldots, 0\}$;
$map \leftarrow \{0, 1, 0, \ldots, 0\}$, $m \leftarrow 0.85$, $\epsilon \leftarrow 0.001$;
while *not all v have Converged* **do**
  for $e = (s, d) \in E$ **do**
    $contrib[d] \leftarrow contrib[d] + \frac{PR_{old}[s]}{deg[s]}$;
  for $v \in V$ **do**
    $PR_{new}[v] \leftarrow (1-m) \times map[v] + m \times contrib[v]$;
    if $\frac{|PR_{new}[v] - PR_{old}[v]|}{PR_{old}[v]} < \epsilon$ **then**
      $Converge(v)$;
  $Swap(PR_{new}, PR_{old})$;

Algorithm 2: Graph Attention(**GATN**).

for $e = (s, d) \in E$ **do**
  $accum[s] \leftarrow accum[s] + w_e$;
for $e = (s, d) \in E$ **do**
  $attention_e \leftarrow \frac{w_e}{accum[s]}$;

Baselines. We provide two baselines to evaluate Graphitron. To benchmark the **performance**, we implement ThunderGP [8], a state-of-the-art FPGA-based graph processing framework optimized for efficient memory access at both on-chip and off-chip levels. Additionally, to measure **flexibility**, we introduce a naïve HLS-based FPGA graph accelerator, developed directly using HLS code to describe graph processing algorithms with some basic HLS-specific optimizations for loops, including pipelining and loop unrolling.

5.2 Performance Evaluation

As illustrated in Figure 3, both Graphitron and ThunderGP significantly outperform the naïve HLS implementation when executing classical graph processing algorithms, primarily due to their extensive integrated memory access optimizations. Furthermore, Graphitron consistently achieves better performance than ThunderGP across all datasets for BFS and SSSP, yielding average speedups of 1.48× and 1.21×, respectively. The primary reason for these improvements is that Graphitron enables users to implement accelerators with hybrid processing kernels of both VCP and ECP paradigms. Moreover, Graphitron can dynamically transition between these paradigms based on the number of frontiers during the runtime, whereas ThunderGP supports only the edge-centric paradigm. When the number of frontiers is low, Graphitron's vertex-centric kernel effectively reduces redundant workloads, resulting in higher overall throughput. However, Graphitron's performance in PageRank is slightly (4.2%) lower compared to ThunderGP. This minor performance gap arises because ThunderGP, due to its fixed interfaces, only supports returning the *cumulative sum* of differences between the old and new PageRank values. In contrast, Graphitron provides developers the flexibility to individually record these differences as a separate vertex property. Although this enhances programmability and flexibility, it introduces additional vertex property accesses, resulting in a modest performance overhead.

The most significant performance improvements in Graphitron compared to naïve HLS implementation stem from memory access optimizations. We designate Graphitron variants that individually integrate burst access, shuffling, or caching optimizations as **+BurstAccess**, **+Shuffle**, and **+Cache**, respectively. Figure 4 illustrates the speedups obtained by these variants over the naïve HLS implementation when executing BFS. The results indicate that variants of Graphitron, which incorporate only a single memory access optimization, yield limited performance gains. This limitation occurs because graph processing workloads involve diverse memory access patterns. Therefore, optimizing only one pattern leaves others as performance bottlenecks. By integrating all three memory access optimizations simultaneously, Graphitron can effectively mitigate inefficiencies across multiple dimensions, resulting in substantial performance improvement.

5.3 Flexibility Evaluation

Due to reliance on a fixed GAS programming interface, ThunderGP cannot support algorithms such as PPR and GATN. Specifically, ThunderGP is incapable of defining a variable number of graph properties, such as the mapping arrays for vertices required by PPR, making it unsuitable for implementing the PPR algorithm. Additionally, ThunderGP cannot handle writes to edge properties, such as attentions in GATN,

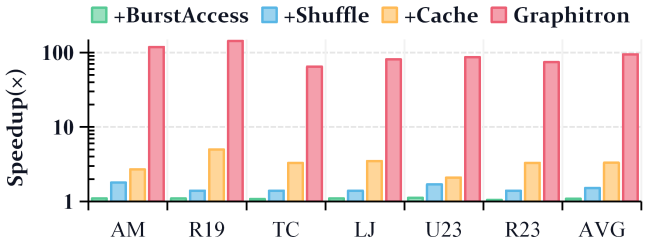

Figure 4. Speedups to naïve HLS implementation with different memory access optimizations when running BFS. (**Y-axis is logarithmic.**)

thereby precluding its support for GATN. In contrast, both the naïve HLS implementation and Graphitron can effortlessly generate FPGA-based accelerators for these algorithms. As detailed in Table 5, Graphitron achieves average speedups of 71.6× and 243.5× over the naïve HLS implementations for PPR and GATN, respectively, demonstrating its strong performance combined with high flexibility.

Table 5. Speedups of Graphitron to naïve HLS implementation on **PPR** and **GATN**.

Algorithm	*AM*	*R19*	*TC*	*LJ*	*U23*	*R23*	*Avg.*
PPR	86.1×	68.1×	85.1×	56.7×	66.4×	67.4×	71.6×
GATN	220.0×	100.5×	129.8×	202.7×	427.1×	379.0×	243.5×

Graphitron provides greater flexibility in algorithm design compared to ThunderGP, as summarized in Table 6. Specifically, Graphitron supports VCP, ECP, and hybrid combinations of these paradigms, whereas ThunderGP exclusively supports ECP, requiring traversing all edges in each iteration. This limitation results in wasted bandwidth and computation, particularly for algorithms with fewer frontiers. Furthermore, ThunderGP treats edge weights as constants, preventing dynamic modification by the accelerators. In contrast, Graphitron enables developers to dynamically update edge weights efficiently by the accelerator, which is crucial for GNN-related algorithms. Additionally, conventional frameworks impose strict constraints on the number and type of hardware kernels and graph properties. Graphitron, however, allows developers to flexibly define hardware kernels and graph properties, bounded only by the resource constraints of the FPGA, significantly enhancing the design flexibility of accelerator development.

5.4 Design Productivity Evaluation

As shown in Table 6, whereas ThunderGP demands modifications across at least five different files, Graphitron enables developers to implement a complete graph algorithm in a single file with fewer than 100 lines of code. Furthermore, encapsulating algorithms within ThunderGP's GAS model necessitates an in-depth understanding of graph algorithms, whereas defining operations using ECP or VCP in Graphitron

is more intuitive. Additionally, ThunderGP's configuration files often require extensive FPGA-specific expertise for accurate implementation, and introducing new hardware kernels involves substantial manual debugging effort, adding considerable complexity. Graphitron empowers even inexperienced users to efficiently define graph algorithms using a concise, high-level language, significantly lowering the barrier to accelerator design. The compilation time of Graphitron is 5.9% longer than that of ThunderGP. This modest overhead arises from the compiler's capability to accommodate flexible developer-defined hardware descriptions. For instance, introducing vertex kernel introduces pointers and access operations to the generated hardware modules, slightly increasing design complexity and consequently resulting in additional synthesis overhead.

Table 6. Design productivity and flexibility comparison.

Systems		ThunderGP	Graphitron
Programming Complexity	*Code Length*	modify to ≥ 5 files	≈100 lines in one file
	Programming Interface	Only GAS Model	Edge/Vertex-centric Operations
	FPGA Understanding	Proficient	Inexperienced
Programming Flexibility	*VCP*	✗	✓
	ECP	✓	✓
	Hybrid(VCP+ECP)	✗	✓
	Weight Modification	✗	✓
	Variable kernels	✗	✓
	Variable properties	✗	✓
Compilation Overhead		4h55min	5h12min

6 Conclusion

In this work, we presented Graphitron, a DSL for agile development of graph accelerators on FPGAs. Graphitron abstracts away low-level hardware complexities to the users, and defines vertices and edges as primitive data types to supports both edge and vertex operations for convenient algorithm description. The back-end of Graphitron compiler incorporates hardware optimizations like pipelining, caching, burst accessing, and shuffling to automatically generate high-performance graph accelerators. Experiments on various graph processing algorithms show that Graphitron-generated accelerators deliver comparable performance with state-of-the-art graph processing framework on FPGA, while offering superior design productivity and flexibility.

Acknowledgments

This work is in part supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, under Grant No. XDB0660000, XDB0660100, XDB0660102, and XDB0660103, and the National Key Research and Development Program of China, under Grant No.2022YFB4500405.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.

[2] Amazon. Accessed Feb. 25, 2024. Amazon Neptune. https://aws.amazon.com/cn/neptune.

[3] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. 2014. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 228–235.

[4] Ajay Brahmakshatriya, Emily Furst, Victor A Ying, Claire Hsu, Changwan Hong, Max Ruttenberg, Yunming Zhang, Dai Cheol Jung, Dustin Richmond, Michael B Taylor, et al. 2021. Taming the zoo: The unified graphit compiler framework for novel architectures. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 429–442.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[6] Xinyu Chen, Ronak Bajaj, Yao Chen, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2019. On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 67–73.

[7] Xinyu Chen, Yao Chen, Feng Cheng, Hongshi Tan, Bingsheng He, and Weng-Fai Wong. 2022. ReGraph: Scaling graph processing on HBM-enabled FPGAs with heterogeneous pipelines. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1342–1358.

[8] Xinyu Chen, Feng Cheng, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2022. ThunderGP: resource-efficient graph processing framework on FPGAs with hls. *ACM Transactions on Reconfigurable Technology and Systems* 15, 4 (2022), 1–31.

[9] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph processing framework on FPGA a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 105–110.

[10] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 217–226.

[11] Jonas Dann, Daniel Ritter, and Holger Fröning. 2024. GraphScale: Scalable processing on FPGAs for HBM and large graphs. *ACM Transactions on Reconfigurable Technology and Systems* 17, 2 (2024), 1–23.

[12] Marcos Lopez De Prado. 2018. *Advances in financial machine learning*. John Wiley & Sons.

[13] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[14] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. 2019. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology* 34 (2019), 339–371.

[15] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[16] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[17] Wole Jaiyeoba, Nima Elyasi, Changho Choi, and Kevin Skadron. 2023. Acts: a near-memory FPGA graph processing framework. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 79–89.

[18] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 411–424.

[19] Nachiket Kapre. 2015. Custom FPGA-based soft-processors for sparse graph acceleration. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 9–16.

[20] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. {GraphChi}:{Large-Scale} Graph Computation on Just a {PC}. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 31–46.

[21] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. 2017. Extrav: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1706–1717.

[22] Jure Leskovec and Andrej Krevl. Accessed Feb. 25, 2024. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[23] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[24] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[25] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[26] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 456–471.

[27] Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd, et al. 1999. The pagerank citation ranking: Bringing order to the web. (1999).

[28] Amin Sahebi, Marco Barbone, Marco Procaccini, Wayne Luk, Georgi Gaydadjiev, and Roberto Giorgi. 2023. Distributed large-scale graph processing on FPGAs. *Journal of big Data* 10, 1 (2023), 95.

[29] Runbin Shi, Yuhao Ding, Xuechao Wei, He Li, Hang Liu, Hayden K-H So, and Caiwen Ding. 2020. FTDL: a tailored FPGA-overlay for deep learning with high scalability. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[30] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[31] Chunyou Su, Linfeng Du, Tingyuan Liang, Zhe Lin, Maolin Wang, Sharad Sinha, and Wei Zhang. 2024. GraFlex: Flexible Graph Processing on FPGAs through Customized Scalable Interconnection Network. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 143–153.

[32] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241* (2015).

[33] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[34] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[35] Stanley Wasserman and Katherine Faust. 1994. Social network analysis: Methods and applications. (1994).

[36] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[37] Xilinx Inc. Accessed Feb. 25, 2024. Xilinx High-Level Synthesis (HLS). https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[38] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[39] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. 2019. Hitgraph: High-throughput graph processing framework on fpga. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2249–2264.

[40] Shijie Zhou and Viktor K Prasanna. 2017. Accelerating graph analytics on CPU-FPGA heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 137–144.

Received 2025-03-21; accepted 2025-04-21