TGraph: A Tensor-centric Graph Processing Framework

YONGLIANG ZHANG*, Wuhan University, China
YUANYUAN ZHU*, Wuhan University, China
HAO ZHANG, Huawei Technologies, China
CONGLI GAO, Huawei Technologies, China
YUYANG WANG*, Wuhan University, China
GUOJING LI*, Wuhan University, China
TIANYANG XU*, Wuhan University, China
MING ZHONG*, Wuhan University, China
JIAWEI JIANG*, Wuhan University, China
TIEYUN QIAN*, Wuhan University, China
CHENYI ZHANG, Huawei Technologies, China
JEFFREY XU YU, The Chinese University of Hong Kong, China

Graph is ubiquitous in various real-world applications, and many graph processing systems have been developed. Recently, hardware accelerators have been exploited to speed up graph systems. However, such hardware-specific systems are hard to migrate across different hardware backends. In this paper, we propose the first tensor-based graph processing framework, TGraph, which can be smoothly deployed and run on any powerful hardware accelerators (uniformly called XPU) that support Tensor Computation Runtimes (TCRs). TCRs, which are deep learning frameworks along with their runtimes and compilers, provide tensor-based interfaces to users to easily utilize specialized hardware accelerators without delving into the complex low-level programming details. However, building an efficient tensor-based graph processing framework is non-trivial. Thus, we make the following efforts: (1) propose a tensor-centric computation model for users to implement graph algorithms with easy-to-use programming interfaces; (2) provide a set of graph operators implemented by tensor to shield the computation model from the detailed tensor operators so that TGraph can be easily migrated and deployed across different TCRs; (3) design a tensor-based graph compression and computation strategy and an out-of-XPU-memory computation strategy to handle large graphs. We conduct extensive experiments on multiple graph algorithms (BFS, WCC, SSSP, etc.), which validate that TGraph not only outperforms seven state-of-the-art graph systems, but also can be smoothly deployed and run on multiple DL frameworks (PyTorch and TensorFlow) and hardware backends (Nvidia GPU, AMD GPU, and Apple MPS).

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: Graph Computation System, Tensor Computation Runtimes

*Also with School of Computer Science, Wuhan University, Wuhan, China

Authors' Contact Information: Yongliang Zhang, yongliangzhang@whu.edu.cn, Wuhan University, China; Yuanyuan Zhu, yyzhu@whu.edu.cn, Wuhan University, China; Hao Zhang, zhanghao687@huawei.com, Huawei Technologies, China; Congli Gao, gaocongli@huawei.com, Huawei Technologies, China; Yuyang Wang, yuyangwang@whu.edu.cn, Wuhan University, China; GuoJing Li, guojingli@whu.edu.cn, Wuhan University, China; Tianyang Xu, tianyangxu01@gmail.com, Wuhan University, China; Ming Zhong, clock@whu.edu.cn, Wuhan University, China; Jiawei Jiang, jiawei.jiang@whu.edu.cn, Wuhan University, China; Tieyun Qian, qty@whu.edu.cn, Wuhan University, China; Chenyi Zhang, zhangchenyi2@huawei.com, Huawei Technologies, China; Jeffrey Xu Yu, yu@se.cuhk.edu.hk, The Chinese University of Hong Kong, China.

ACM 2836-6573/2025/1-ART81
https://doi.org/10.1145/3709731

ACM Reference Format:
Yongliang Zhang, Yuanyuan Zhu, Hao Zhang, Congli Gao, Yuyang Wang, GuoJing Li, Tianyang Xu, Ming Zhong, Jiawei Jiang, Tieyun Qian, Chenyi Zhang, and Jeffrey Xu Yu. 2025. TGraph: A Tensor-centric Graph Processing Framework. *Proc. ACM Manag. Data* 3, N1(SIGMOD), Article 81 (January 2025), 27 pages. https://doi.org/10.1145/3709731

1 Introduction

Graph is ubiquitous in various real-world applications [30] such as social networks, recommendation systems, biological analysis, etc. Many graph processing systems have been developed in the past decade, which mainly fall into two categories: (1) *shared-memory systems* including *in-memory* systems such as Ligra [58, 59], Galois [47], GraphMat [61], etc., and *out-of-core* systems such as GraphChi [39], X-Stream [53], GridGraph [76], NXgraph [14], MiniGraph [77], etc.; (2) *distributed systems* such as Pregel [43], PowerGraph [22], GRACE [64], GPS [55], Blogel [68], GraphX [23], GraphD [69], SLFE [60], Grape [17], GraphScope [29, 67], etc.

Recently, to leverage the computation power of new hardware accelerators, GPU-based systems have been developed to simplify GPU programming and balance workloads on massive GPU threads. A pioneering work Medusa [75] develops an edge-message-vertex model to simplify parallel graph processing on GPUs. A vertex-centric framework CuSha [37] uses G-shards and concatenated windows to address irregular memory access. Frog [57] is an asynchronous graph system based on hybrid coloring of graph vertices. Gunrock [65] proposes a data-centric model to manipulate active vertex/edge subsets and several load-balancing strategies at different granularities. cuGraph [18] is a vertex/edge-centric system which provides a new data structure to store and partition graphs. GraphBLAST [70] is a linear-algebra-based graph framework on GPUs that exploits data sparsity to reduce memory access. The CPU-GPU based graph systems adopt different strategies to reduce data transfer, including performance modeling (TOTEM [20] and FinePar [71]), active subgraph evaluation (Scaph [74]), and compact subgraph representation (Subway [54]). GraphGen [48], FPGP [15], and ThunderGP [11] are graph processing systems accelerated by FPGAs.

However, the above hardware-accelerated graph systems are tailored for a specific type of GPU/FPGA such as Nvidia GPU or Xilinx FPGA, and cannot be easily migrated to other hardware backends such as Mac MPS and AMD GPU, not to mention other emerging hardware accelerators driven by the development of Deep Learning (DL), such as Tensor Processing Unit (TPU), Neural Processing Unit (NPU), etc. The underlying reason is that the characteristics and primitives of these hardware are inherently different, and we have to exploit the hardware-specific features through specialized low-level kernels such as NVIDIA CUDA and Xilinx Vitis to boost the performance.

In this paper, we propose the first general graph processing framework, TGraph, which can be easily deployed and run on multiple hardware accelerators, including GPUs, TPUs, NPUs, and others, uniformly called .XPUs in this paper. We build TGraph based on tensors, the computation and manipulation of which have been significantly accelerated by the DL frameworks (PyTorch [49], TensorFlow [1], MXNet [9], etc.) and their associated compilers and runtimes (TVM [10], ONNX [45], etc.), which are collectively referred to as Tensor Computation Runtimes (TCRs). TCRs provide a set of tensor operators for users to leverage the acceleration capabilities of underlying specialized hardware, such as GPU, TPU, and NPU, for batch data processing without considering their specific characteristics and primitives. Additionally, TCRs [10, 41] offer hardware-independent optimizations, including operator fusion, operator sinking, and algebraic simplification, to enhance the Intermediate Representation (IR) of the tensor program. Meanwhile, TCRs also provide hardware-specific optimizations to enable efficient code generation for different hardware targets. For example, PyTorch utilizes CUDA Streams (which is transparent to PyTorch users) to manage the execution of the tensor operators to generate the highly optimized GPU codes. Besides the native support of

DL models, TCRs have also exhibited their effectiveness in supporting non-DL systems, such as Hummingbird [46] for traditional ML models and TQP [28] for relational queries.

Despite the above progress on non-DL data processing tasks, building an efficient graph processing system that fully leverage the parallelism of tensor operators is a non-trivial task facing the following challenges:

(1) *Expressivity*. How to design a unified tensor-based graph computation model to support a wide range of graph algorithms? Existing GPU-based systems usually manipulate vertices/edges with a finer granularity to balance the workload on massive GPU threads, which is not applicable to our tensor-based system that prefers a large batch of data to fully leverage the parallelism of tensor operators and reduce kernel launch overhead in tensor operators.

(2) *Extensibility*. How to shield the computation model from the detailed tensor operators so that TGraph can be easily migrated and deployed across different TCRs and hardware backends? We need to abstract a high-level computational model to make the system decoupled from the underlying tensor operators in different TCRs.

(3) *Scalability*. How to efficiently handle large-scale graphs that cannot fit into the limited memory of hardware accelerators? Compared to CPU, the memory capacities of XPU are usually limited, thus scaling strategies based on tensors need to be designed.

To tackle the above challenges, we make the following efforts. First, we propose a *tensor-centric computation model* which decomposes graph algorithms into a series of iterations consisting of two steps: TENSORIZE to organize active vertices and their neighbors as tensors, and COMPUTE to perform computation based on tensor operators. Furthermore, to maximally leverage the parallelism of tensor computation, we abstract a set of graph operators (neighborSelect, vertexSelect, reconstruct, etc.) based on tensor operators. TENSORIZE and COMPUTE can be easily implemented by these graph operators to support a wide range of graph algorithms, thus achieving high *expressivity*. Meanwhile, the abstraction of the two steps and five graph operators shields the system from the detailed tensor operators so that TGraph can be easily migrated across different platforms, and thus achieving high *extensibility*. Finally, to achieve high *scalability*, we propose tensor-based graph compression and computation strategies to save space and accelerate computation, and tensor-based partition and pipelined scheduling strategies to support efficient out-of-XPU-memory computation. Our main contributions are summarized below.

- We propose a *tensor-centric computation model*, which provides two interfaces TENSORIZE and COMPUTE to support the implementation of graph algorithms. We also abstract a set of graph operators to shield the computation model from the detailed tensor operators so that TENSORIZE and COMPUTE can be easily implemented.
- We propose scaling strategies to deal with large graphs, which include: a tensor-based compression and computation strategy, and an out-of-XPU-memory computation strategy including balanced partitioning and pipelined scheduling.
- We implement TGraph, the first tensor-based graph processing framework that can be smoothly deployed and run on different DL frameworks and hardware. It allows users to easily implement graph algorithms without delving into the underlying parallelization.
- We validate the outperformance of TGraph by extensive comparison studies with seven state-of-the-art shared-memory and GPU-based graph systems. Moreover, we validate the extensibility of TGraph by deploying and running it on two ML frameworks (PyTorch and TensorFlow) and three hardware accelerators (Nvidia GPU, AMD GPU, and MAC MPS).

Road map. Section 2 introduces the preliminaries. Section 3 gives the system overview of TGraph. Section 4 introduces the tensor-centric computation model. Section 5 presents the scaling strategies. Experimental studies are reported in Section 6. Section 7 reviews the related work, and Section 8 concludes the paper.

Table 1. Basic tensor operators

Category & Functionality	Representative Operators
Initialization: create a tensor based given data.	`zeros`, `ones`, `empty`, `fill`, `arange`, `repeat_interleave`, etc.
Indexing: choose specific elements from a tensor.	`index_select`, `mask_select`,etc.
Comparison: compare a tensor with another tensor.	`eq`, `lt`, `gt`, `le`, `ge`, `searchsorted`, etc.
Arithmetic: perform arithmetic operations on tensors.	`add`, `mul`, `div`, `sub`, etc.
Reorganization: rearrange elements from tensors.	`sort`, `cat`, `stack`, `roll`, etc.
Aggregation: aggregate over a tensor or groups.	`max`, `min`, `mean`, `sum`, `cumsum`, `unique`, `scatter_reduce`, `segment_csr`, etc.

2 Preliminary

Graph Definitions and Representations. A simple undirected graph is denoted by $G = (V, E)$, where V is the vertex set and $E \subseteq V \times V$ is the edge set. We use $V(G)$, $E(G)$, $n = |V(G)|$, and $m = |E(G)|$ to denote the vertex set, edge set, vertex number, and edge number in G, respectively. The neighbors and degree of a vertex v in graph G are defined as $N(v, G) = \{u \in V(G) | (u, v) \in E(G)\}$ and $deg(v, G) = |N(v, G)|$. These definitions can also be extended to direct graphs. When the context is clear, we omit G in above notations for simplification. One of the basic graph representations is the *adjacency matrix* with the space cost of $O(n \times n)$. Since real-world graphs are usually sparse, *Coordinate List* (COO) or *Compressed Sparse Row* (CSR) are used to reduce space overhead. COO represents a graph by two arrays, *src* and *dest* of size m, indicating the source and destination vertices of edges. CSR offers a more compact space utilization, where the array *neighbors* of size m stores the neighbors of each vertex in V and the array $offset$ of size $n + 1$ stores the starting and ending positions of neighbors. Specifically, $offset[i]$ and $offset[i + 1]$ indicate the starting and ending positions of neighbors for vertex v_i in *neighbors*.

Graph Computation Model. The graph computation model defines the computational units and programming interfaces to help users program with the system and distribute the program for parallel computation. The *vertex-centric model* is widely adopted by many frameworks, such as Pregel [43], Giraph [21], Powergraph [22], Cusha [37], etc. It provides a vertex-specific programming interface by hiding the partition and coordination details so that users can easily write the program from the perspective of a vertex. The *edge-centric model* was proposed to deal with power-law graphs where the computational unit is an edge instead of a vertex, e.g., X-Stream [53] and Chaos [53]. The computation in each iteration can be divided into two or three phases, such as push-pull [13], scatter-gather [53], advance-compute [12], gather-apply-scatter [22] [52], etc. The *component/subgraph-centric model* manipulates components/subgraphs consisting of a collection of vertices/edges to achieve coarse-grained parallelism. It was initially introduced by Giraph++ [62], and then adopted by other systems such as Grape [17], Minigraph [77], etc.

Tensor and its Operators. A tensor is a multi-dimensional array, where a scalar is a 0-dimension array, a vector is a 1-dimensional array and a matrix is a 2-dimensional array. To accelerate tensor computation, hardware makers and cloud vendors (Intel, AMD, Apple, etc.) have made great efforts to develop specialized hardware and primitives to speed up tensor computation, such as GPU, TPU, NPU, etc. Atop hardware and their primitives, a set of DL frameworks (PyTorch [49], TensorFlow [1], MXNet [9], etc.) along with their compilers and runtimes (TVM [10], ONNX runtime [45], etc.) are developed and collectively referred to as TCRs. TCRs offer a rich set of operators to compute and manipulate tensors, which significantly simplify the process of exploiting the parallel capabilities offered by the specialized hardware. Table 1 lists the common tensor operators provided by different

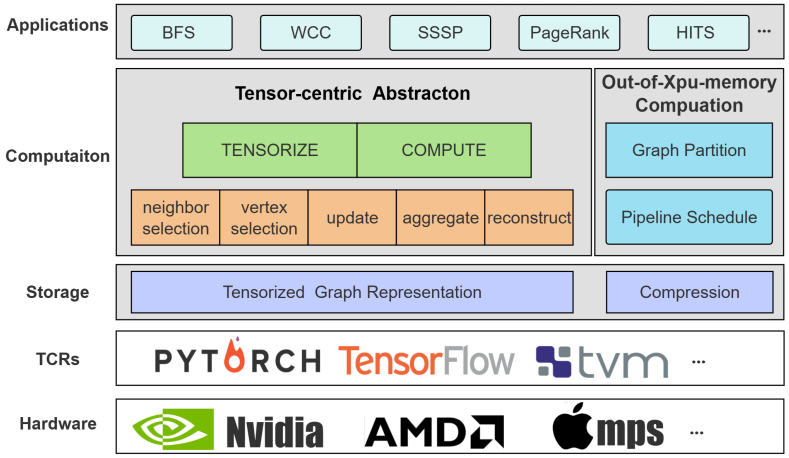


Fig. 1. Overview of system architecture

TCRs, and the specific meaning of each tensor operator will be introduced later at usage. Here, we use PyTorch as an example, and the operators in other TCRs such as TensorFlow and MXNet are basically the same.

3 System Overview

The overview of TGraph is shown in Fig. 1. We build TGraph atop TCRs with three layers. At the core of the system, the *computation layer* provides the tensor-centric computation module with abstracted interfaces and the out-of-XPU-memory computation module for large-scale graphs. On one side, the *computation layer* can support the implementation of graph algorithms in the upper *applications layer*; on the other side, it is highly compatible with the graph representations in the lower *storage layer*. To further save storage space and accelerate the computation, we provide a tensor-based compression module in the storage layer. Next, we will give a brief overview of the tensor-centric computation model and scaling strategies, including graph compression and out-of-XPU-memory computation, leaving more details to Sections 4 and 5.

Tensor-centric Computation Model. We propose a *tensor-centric computation model* which depicts graph algorithms as an iterative process and organizes active vertices and their adjacent edges in each iteration as tensors. Although these active vertices/edges can also be considered as subgraphs, TGraph differs from existing subgraph-centric systems that use specialized data structures and operations tailored for specific hardware (e.g. Giraph++ and Grape on CPUs, MiniGraph on GPUs). Such specialization makes it hard to migrate them to other hardware platforms. TGraph is built on TCRs, which utilize tensor as the data structure and thus can be supported by diverse hardware. However, organizing active vertices/edges as tensors and efficiently manipulating them is challenging as we need to maximally leverage the parallelism of tensor operators and reduce the kernel launch overhead of tensor operators to achieve high performance. Thus, we provide two programming interfaces: TENSORIZE to organize active vertices/edges as large one-dimensional tensors, and COMPUTE to perform the computation by tensor operators. We also provide a set of highly optimized graph operators, including vertexSelect, neighborSelect, reconstruct, aggregate, and update to help users easily implement the highly efficient interfaces while shielding graph algorithms from tedious tensor operators. Such an abstraction can enable the easy migration of TGraph across different TCRs.

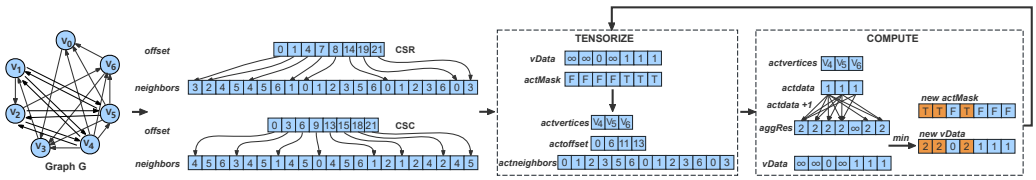


Fig. 2. An example of running BFS on graph G through TENSORIZE and COMPUTE.

Algorithm 1: The conceptual computation process of TGraph

Input : A graph $TG = (vertices, offset, neighbors)$
Output: A computation result tensor $vData$

```
vData, actMask ← INIT(TG);
while any (actMask) ≠ False do
    TSubgraph ← TENSORIZE(TG, actMask);
    vData, actMask ← COMPUTE(TSubgraph, vData, actMask);
return vData;
```

Scaling Strategies. The memory of the hardware accelerators (i.e., XPUs) is quite limited compared to the host memory, which may hinder the application of TGraph on large-scale graphs. A possible solution is to compress the large graph and compute the compressed data. How to achieve this based on tensors is nontrivial. We provide a tensor-based compression strategy which can save a significant amount of storage space and meanwhile speed up the computation. For a large-scale graph that still cannot fit into the XPU memory, we provide the out-of-XPU-memory computation strategy, where the XPU memory is used to accelerate the tensor operators and the host memory is used to store graphs. We propose the tensor-based partition strategies to divide the input graph into balanced subgraphs. Then, these subgraphs are scheduled into the XPU in a pipeline manner until the entire computation converges.

4 Tensor-centric Computation Model

In this section, we will first introduce the tensor-centric programming model and the abstracted graph operators, and then show how to implement graph algorithms based on them and how to map graph operators to tensor operators. Finally, we will discuss the potential optimizations brought by the tensor-centric model.

4.1 Tensor-centric Programming Abstraction

TGraph abstracts graph applications as an iterative process, where each iteration involves a series of steps. Specifically, TGraph divides each iteration into two main steps: (1) TENSORIZE to organize the subgraph consisting of active vertices and edges into tensors. We need to efficiently organize active vertices/edges into large one-dimensional tensors instead of multiple small tensors to maximize the parallelism of hardware backend and reduce the kernel launch overhead for each tensor operator in the following COMPUTE step. (2) COMPUTE to perform computation on the tensorized subgraph. For each vertex within the subgraph, we need to aggregate the data received from the neighboring vertices, update its value, and identify the active vertices in the next iteration.

The conceptual computation process of TGraph is shown in Alg. 1. The input is a tensor representation of graph G. TGraph can support existing graph representations such as CSR and COO. Here, we adopt CSR, which is compact and can preserve the locality of neighbors. COO can also be integrated into our system in a similar way, if needed. Thus, the tensor representation of G

Table 2. Graph operators in TGraph v.s. graph primitives in linear-algebra-based graph systems

Systems	*Creation*	*Transpose*	*BFS*	*Extract*	*Insert*	*Union/Intersect*	*Reduce*	*Apply*
Combinatorial BLAS [8]	-	-	`SpGEMM`, `SpMV`	`SpRef`	`SpAsgn`	`SpEWiseX`	`Reduce`	`Apply`
Graph BLAS [34]	`build`	`transpose`	`MxM`, `MxV`, `VXM`,	`extract`	`assign`	`eWiseAdd`, `eWiseMult`	`reduce`	`apply`
GraphBLAST [70]	-	`transpose`	`mxm`, `mxv`, `vxm`	`extract`	`assign`	`eWiseAdd`, `eWiseMult`	`reduce`	`apply`
TGraph	-	-	✓	vertexSelect neighborSelect, reconstruct	-	-	aggregate	update

is $TG = (vertices, offset, neighbors)$, where $neighbors$ and $offset$ are 1-dimensional tensors to represent the neighbors and the index position of vertices in CSR. To efficiently locate the vertices, we reassign the vertex IDs by a continuous integer sequence starting from 0 to $n-1$ and use an additional tensor $vertices$ of size n to store all the vertex IDs. We use a bool tensor $actMask$ of size n to indicate which vertices are active in the current iteration where $True$ represents the status of active and use a one-dimensional tensor $vData$ of size n to keep the current value of all the vertices. These two tensors can be initialized by a user-defined function INIT for different applications (line 1). Then we extract the subgraph consisting of the active vertices and their neighbors and organize them as tensors through TENSORIZE (line 3). Then we use COMPUTE to conduct computation on the tensorized subgraph $TSubgraph$ (line 4), including aggregating information from neighbors, updating the value for each vertex in $vData$, and updating the masks for active vertices stored in $actMask$. The iteration terminates when there are no active vertices, i.e., $\text{any}(actMask) = False$.

Through the abstracted interfaces TENSORIZE and COMPUTE, we can develop a wide range of algorithms that can be executed in an iterative way such as BFS, WCC, SSSP, PageRank, etc. Fig. 2 takes BFS as an example to show how the algorithm runs through these two steps. Suppose that v_2 is the source vertex, and we are at the second iteration where $\{v_4, v_5, v_6\}$ are currently active. In TENSORIZE, we extract the subgraph from the tensor-based graph TG based on $\{v_4, v_5, v_6\}$ and their neighbors and reorganize them as tensors. In COMPUTE, we send current data $\{1, 1, 1\}$ in $\{v_4, v_5, v_6\}$ to their neighbors, respectively, and increase the value by 1. Thus, $\{v_0, \ldots, v_4, v_5, v_6\}$ obtain the aggregated data, but only $\{v_0, v_1, v_3\}$ with smaller value than $vData$ will be updated and activated.

4.2 Graph Operator Abstraction and Analysis

Our high-level tensor-centric abstraction with two interfaces provides flexibility to users to manipulate the tensorized subgraphs for different graph algorithms. However, it also requires users to be quite familiar with the detailed tensor operators to implement TENSORIZE and COMPUTE efficiently. Thus, to bridge this gap, we abstract a set of graph operators: vertexSelect, neighborSelect, reconstruct, etc., which hide the detailed tensor operators from users and enable easy migration across different TCRs. Here, we briefly introduce the function and parameters of each graph operator, leaving the implementation details to Section 4.4. Without loss of generality, all the parameters are 1-dimensional tensors.

- vertexSelect. The function of this operator is to select a subset from a vertex set. It takes a vertex set $vertices$ and a selection indicator $actMask$ as input and outputs a smaller tensor $actvertices$ to represent the selected vertices. Usually, vertexSelect is used in TENSORIZE to select active vertices.
- neighborSelect. The neighborSelect operator is designed to efficiently select the neighbors for some specific vertices. It takes a tensorized graph $TG = (vertices, offset, neighbors)$ and the active vertices $actvertices$ as input, and outputs a tensor $actneighbors$ to represent neighbors of $actvertices$. neighborSelect is usually invoked in TENSORIZE to select the incoming or outgoing neighbors of the active vertices.

Algorithm 2: TENSORIZE

Input : A tensorized graph $TG = (vertices, offset, neighbors)$, and the vertex mask $actMask$
Output: An extracted tensorized subgraph $TSubgraph$

```
actvertices ← vertexSelect(vertices, actMask);
actneighbors ← neighborSelect(TG, actvertices);
TSubgraph ← reconstruct(TG, actvertices, actneighbors);
return TSubgraph;
```

- reconstruct. Given a vertex subset and its neighbors, we use reconstruct to organize them into a tensorized subgraph for the next COMPUTE step. It takes a tensorized graph $TG = (vertices, offset, neighbors)$, the selected vertices $actvertices$ and their neighbors $actneighbors$ as input, and outputs the tensor representation of the subgraph, $TSubgraph$.
- aggregate. The aggregate operator is used to aggregate the data received from the neighboring vertices for each vertex within the subgraph in the COMPUTE phase. It takes a tensorized subgraph $TSubgraph$ obtained by reconstruct and a tensor $aggData$ that represents the data of each vertex as input, and outputs the aggregate results $aggRes$.
- update. The update operator is used to update the status of each vertex based on the aggregated data in a user-specific way in the COMPUTE phase. It takes the current active mask $actMask$, current vertex data $vData$, and an update function with aggregated result $aggRes$ as input, and updates the values of $vData$ and $actMask$.

Expressiveness Analysis of Graph Operators. These graph operators can maximize the parallelism of tensor computation and ease the implementation of graph algorithms for users. However, implementing graph algorithms based on these five graph operators also brings constraints to algorithm designing. A natural question arising is whether these graph operators are expressive enough to represent a wide range of graph algorithms? Specifically, expressiveness refers to how many graph algorithms can be supported by these graph operators. In other words, we need to consider what kinds of graph operators are required to support a wide range of graph algorithms. Graph systems based on linear algebra, such as CombBLAS [8], GraphBLAS [44] [35] [34], and GraphBLAST [70] propose a set of graph operators/primitives. As shown in Table 2, their primitives are slightly different due to differences in the implementations, but generally fall into the following categories: *Create* to build a graph from a set of vertices and edges, *Transpose* to change the direction of edges, *BFS* to conduct single-source, multi-source or weighted breadth-first search, *Extract* to extract a subgraph from a graph, *Insert* to insert a subgraph to a large graph, *Union/Intersect* to union or intersect two graphs, *Reduce* to aggregate vertex neighboring information, *Apply* to update edge/vertex values. These graph primitives have been shown to be able to support a wide range of graph algorithms [36]. Our operators can well align with these primitives except *Creation*, *Transpose*, *Insert*, *Union/Intersect*, which are not currently involved in the single-graph analytical algorithms studied in this paper but can be easily implemented with tensor operators if they are involved in other graph algorithms in the future. Besides, the *BFS* is not considered as a graph operator in TGraph, as it can be naturally supported by our graph operators through TENSORIZE and COMPUTE functions. The above analysis shows that our graph operators are sufficient to support a large range of graph algorithms. Equipped with graph operators, together with the highly abstracted interfaces TENSORIZE and COMPUTE, with the assistance of basic tensor operators, users can easily implement a wide range of graph algorithms without knowing tedious details of the complex tensor operators.

Algorithm 3: Implementation of BFS, WCC, PageRank

```
// Implementation of BFS
Function INIT(TG = (vertices, offset, neighbors))
    vData ← ones(|vertices|);
    vData ← mul(vData, INF);
    vData[source] ← 0;
    actMask ← full(|vertices|, False);
    actMask[source] ← True ;
    return vData, actMask;
Function updfunc(G, vData, actMask)
    actMask ← lt(aggRes, vData);
    vData ← min(vData, aggRes);
Function COMPUTE(TSubgraph, vData, actMask)
    aggData ← add(vData, 1);
    aggRes ← aggregate(TSubgraph, aggData, 'min');
    update(vData, actMask, aggRes, updfunc);
// Implementation of WCC
Function INIT(TG = (vertices, offset, neighbors))
    vData ← arange(|vertices|);
    actMask ← full(|vertices|, True);
    return vData, actMask;
Function updfunc(aggRes, vData, actMask)
    actMask ← lt(aggRes, vData);
    vData ← min(vData, aggRes);
Function COMPUTE(TSubgraph, vData, actMask)
    aggData ← vData;
    aggRes ← aggregate(TSubgraph, aggData, 'min');
    update(vData, actMask, aggRes, updfunc);
// Implementation of PageRank
Function INIT(TG = (vertices, offset, neighbors))
    vData ← ones(|vertices|) /|vertices|;
    actMask ← full(|vertices|, True);
    return vData, actMask;
Function updfunc(aggRes, vData, actMask)
    vData ← mul(aggRes, α);
    vData ← add(vData, 1 − α);
Function COMPUTE(TSubgraph, vData, actMask)
    aggData ← div(vData, G.degrees);
    aggRes ← aggregate(TSubgraph, aggData, 'sum');
    update(vData, actMask, aggRes, updfunc);
```

4.3 Implementation of Graph Algorithms

Before getting into the implementation of any specific graph algorithms, we first show how to extract the subgraph constituted by activated vertices along with their neighbors and organize them into tensors in the TENSORIZE function, which is independent of any specific graph algorithms

studied in this paper. As shown in Alg. 2, TENSORIZE first selects the active vertices through the vertexSelect operator (line 1), then uses neighborSelect to fetch the neighbors of active vertices (line 2), and finally extracts and represents the subgraph consisting of *actvertices, actoffset, actneighbors* via reconstruct operator (line 3). The implementation of graph algorithms mainly differs in the COMPUTE function, which involves data aggregation and updates. Specifically, updfunc is a user-defined function used in graph operator update to specify how to update $vData$ and $actMask$ based on the aggregated result. Due to space limitation, we show three representative algorithms (BFS, WCC, and PageRank) in Alg. 3.

In BFS, all vertices except for the source are initialized to INF, and only the source vertex is active(lines 2-6). During the computation, each vertex adds 1 to its current value, sends the value to its neighbors, and then performs the min aggregation (lines 12-13). The updfunc function update the value of each vertex and mark it as active if the aggregated value is less than its current value (lines 9-10).

In WCC, each vertex is assigned a unique value to represent the connected component it belongs to, and every vertex is initialized to be active (lines 16-17). During the computation, the minimum aggregation function is executed to aggregate the data. Function updfunc updates the value of each vertex and mark it as activated if the aggregated value is less than its current value (lines 20-21).

In PageRank, the value of each vertex is initialized to $1/|vertices|$, and all vertices are set to be active (lines 27-28). During the computation, every vertex equally distributes its value to its neighbors and performs the sum aggregation (lines 34-35). In the updfunc function, we compute the new value for a vertex based on the data aggregated from neighbors (lines 31-32).

4.4 Implementation of Graph Operators

Despite the simple function of these graph operators, efficiently implementing them based on tensors is not always straightforward, especially for neighborSelect and aggregate. The underlying reason is that we need to well organize the data and carefully choose the tensor operators to avoid high computational costs and maximally leverage the parallelism of the tensors. That's why these functionally simple operators can be easily implemented sequentially on the CPU but need careful design on TCRs. In the following, we will introduce how to implement these graph operators using tensor operators and briefly analyze their computation cost.

Vertex Selection. vertexSelect is used to choose a subset of vertices from a given vertex set. Given *vertices* of size n and a selection indicator, we can directly select the vertices from *vertices* by the tensor operator `mask_select`. Thus, the step complexity of vertexSelect is $O(\log(n))$ and the work complexity is $O(n)$.

Neighbor Selection. Given a tensorized graph $TG = (vertices, offset, neighbors)$ and a selected vertex set V_s, neighborSelect aims to obtain and organize the neighbors of V_s into a tensor. A naive solution is to retrieve the neighbors $N(v_i)$ from *neighbors* for each vertex v_i in V_s and then combine them by the concatenate tensor operator `cat`. However, sequentially processing vertices in V_s will severely undermine the parallelism of TCR computation, leading to poor performance. Hence, we propose a new way to implement neighborSelect, as shown in Algorithm 4. First, we obtain *start_Index* and *end_Index* of size $|V_s|$ to record the starting and ending indices of neighbors for all vertices in V_s and obtain the number of neighbors for each vertex in V_s, denoted by tensor *sizes* (lines 1-3). Then we compute the index of neighbors for V_s from tensor *neighbors* in lines 5-9. First, we create a tensor *temp* with values from 0 to $\texttt{sum}(sizes)$ by *arange*, which generates a one-dimensional tensor with equally spaced values within a specific range. Obviously, *temp* will be the index to be retrieved from *neighbors* when V_s is a continuous integer sequence starting from 0 to $|V_s| - 1$. However, the elements in V_s are usually discrete. Thus, there is an offset between the

Algorithm 4: Graph Operator neighborSelect

Input : A tensorized graph $TG = (vertices, offset, neighbors)$, and a vertex subset V_s
Output: The neighbors N_s of V_s

```
// Calculate the number of neighboring vertices
1  start_index ← index_select(offset, V_s);
2  end_index ← index_select(offset, V_s + 1);
3  sizes ← end_index − start_index;
// Calculate the index of neighboring vertices
4  temp ← arange ( sum (sizes));
5  initial_index ← roll ( cumsum (sizes, dim=0), 1);
6  initial_index[0] ← 0;
7  vertex_offset ← repeat_interleave (start_index− initial_index, sizes);
8  neighbors_index ← temp + vertex_offset;
// Select neighbors
9  N_s ← index_select (neighbors, neighbors_index);
10 return N_s;
```

actual index of a neighbor in *neighbors* and *temp*. Since neighbors of the same vertex have the same offset (called neighbor offset), we only need to compute the offset of the starting neighbor for each vertex and then expand it to the subsequent neighbors of this vertex. Here, we use the tensor *initial_index* of size $|V_s|$ to represent the index of the starting neighbor of each vertex in V_s when V_s is a continuous sequence starting from 0. The index of each starting neighbor is the sum of the degrees of all preceding vertices for the corresponding vertex. When `cumsum` is applied to *sizes*, it yields the sum of degrees of preceding vertices, including the vertex itself. Therefore, we need to shift the tensor obtained by `cumsum` to the right by one space, which can be achieved using the tensor `roll`, followed by adding 0 at the index 0 to obtain *initial_index*(lines 5-6). Since we have already computed the actual index *start_index* for starting neighbors, the neighbor offset for each vertex in V_s can be computed as *start_index - initial_index* with length $|V_s|$, which can then be expanded into *vertex_offset* with length `sum`(*sizes*) by tensor operator `repeat_interleave` (line 7). Thus, we can obtain the actual index for neighboring vertices of V_s, *neighbors_index*, by adding *vertex_offset* to *temp*. Finally, we perform `index_select` on *neighbors* to get N_s (line 9).

The operator `repeat_interleave` dominates the complexity of neighborSelect. In Alg. 4, the tensor operator `repeat_interleave` duplicates the elements within *size* whose length is $|V_s|$, and generates $vertex\_offset$ with length $|N_s|$. Thus, the step complexity and work complexity of neighborSelect are $O(\log(|V_s|))$ and $O(|V_s| + |N_s|)$, respectively.

Reconstruction. The reconstruct operator aims to reorganize a set of vertices V_s and their neighbors N_s into a tensor representation TS of the subgraph G_s constituted by V_s and N_s for the next COMPUTE step. Since neighbors tensor N_s of size m' has been obtained, we only need to compute the new offset $offset_s$ for G_s. For a vertex v_i, the starting index of its neighboring vertices in N_s corresponds to the cumulative sum of the degrees of the preceding i-1 vertices. Similarly, the ending index of its neighboring vertices is the sum of the degrees of the preceding i (including itself) vertices. Hence, we use the tensor operator *cumsum* to accumulate the degree tensor, which records the degree of each vertex in the subgraph, and then use `cat` to prepend a zero element to the accumulated tensor as the starting neighbor index of the first vertex. The step complexity of *reconstruct* is $O(\log(|V_s|))$ and the work complexity of *reconstruct* is $O(|V_s|)$.

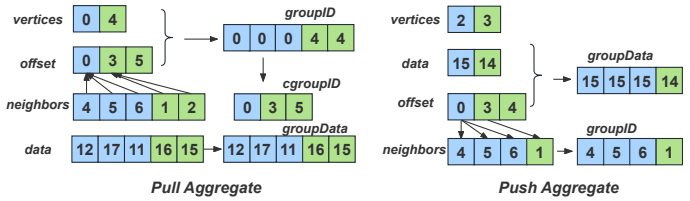


Fig. 3. Pull and Push aggregate operators

Aggregate. The aggregate graph operator is invoked when we need to aggregate the neighboring data of the same vertex. aggregate only focuses on the subgraph obtained in the TENSORIZE step and can be implemented in either push or pull mode. In *push* mode, each vertex pushes its data to the outgoing neighbors. In *pull* mode, each vertex pulls the data from its incoming neighbors.

The implementation of aggregate for these two modes is shown in Alg. 5. We use a 1-dimensional tensor *groupData* to store the data to be aggregated and use *groupID* of the same length to indicate which group each data in *groupData* belongs to. In *push* mode, the *groupID* is just the *neighbors* as every vertex in *vertices* sends the data to its neighbors (line 2). The *groupData* can be obtained by expanding the data sent from *vertices* using the `repeat_interleave` operator (lines 3-4). The repeat number of every data depends on the out-degree of its vertex. After obtaining *groupID* and *groupData*, we use `scatter_reduce` to aggregate the values in *groupData* within the same group(line 5). In *pull* mode, the *groupData* is the data of neighbors, which can be selected from the data array *aggData* by `index_select` (line 7). The *vertices* can be expanded to *groupID* in a similar way adopted in *push* mode. However, since the data within the same group is consecutively stored in *groupData*, we can compress *groupID* into the compressed format *cgroupID*, which only records the starting and ending index in *groupData* for each group. The *cgroupID* is just the $offset$ of the extracted graph (line 8). Thus, we use `segment_csr` instead of `scatter_reduce` for data aggregation as `segment_csr` can aggregate a continuous segment of elements more efficiently (line 9). The size of aggregation results *aggRes* is determined by the maximum group ID in *push* mode and the numbers of groups in *pull* mode. We expand it to a tensor of size n for the unified update (line 10). In push mode, the step complexity of aggregate is $O(\log(|vertices|) + \log(|neighbors|)$ and the work complexity is $O(|neighbors| + |vertices|)$. In pull mode, we use d_{max} to represent the maximum degrees of subgraph vertices. The step complexity of aggregate is $O(\log(d_{max}))$, and the work complexity is $O(|neighbors|)$.

Fig. 3 illustrates how to get *groupData* and *groupID* in two modes. For *pull* mode, the *groupData* is $[12, 17, 11, 16, 15]$, which is the aggregated data of *neighbors*. The vertices $[0, 4]$ can be expanded to $[0, 0, 0, 4, 4]$ as the *groupID*. The *groupID* can be compressed into *cgroupID*, which is just the $offset$. For *push* mode, the *groupID* is just the *neighbors*, i.e. $[4, 5, 6, 1]$. The aggregated data $[15, 14]$ can be expanded to $[15, 15, 15, 14]$ as the *groupData* since vertex 2 has three neighbors and vertex 3 has one neighbor.

Update. update is used to update the data of every vertex in the subgraph based on the tensor *aggRes*. It allows users to define an update function updfunc where a series of arithmetic operators or comparison operators will be executed on *aggRes* to generate *updateRes* with the same size as *aggRes* for updating. The step complexity of update is $O(1)$, and the work complexity of update is $O(|aggRes|)$ as the step complexity and work complexity of arithmetic/comparison operators are $O(1)$ and $O(|aggRes|)$, respectively.

4.5 Further Optimizations

We further propose two strategies for acceleration: dynamic switch between pull and push, and optimization of tensor operators.

Algorithm 5: Graph Operator aggregate

Input : A subgraph $TSubgraph = (vertices, offset, neighbors)$, aggregate data $aggData$, option opt, and mode $mode$
Output : The aggregated result $aggRes$

```
if mode ==′ push′ then
    // aggregate in push mode
    groupID ← neighbors ;
    groupData ← index_select(aggData, vertices);
    groupData ← repeat_interleave(groupData, diff(offset));
    aggRes ← scatter_reduce(groupData, groupID, opt);
else
    // aggregate in pull mode
    groupData ← index_select(aggData, neighbors);
    cgroupID ← offset;
    aggRes ← segment_csr(cgroupID, groupData, opt);
expand aggRes;
return aggRes
```

Dynamic Switch Between Pull and Push. The pull and push modes previously introduced in aggregate can also be supported by other graph operators. Their difference mainly lies in neighborSelect and aggregate, since other graph operators remain the same. For neighborSelect, we select the out-neighbors of active vertices in push mode and select the in-neighbors of all vertices in pull mode. For aggregate, the push mode involves atomic operations because multiple vertices may simultaneously update the value of the same vertex. In contrast, pull mode can avoid atomic operations but introduces unnecessary computational overhead due to involving all vertices for computation. Thus, for iterations with a large number of active vertices, pull mode performs better as it avoids atomic operators; for iterations with a smaller number of active vertices, push mode performs better as it avoids computations of inactive vertices. Thus, we propose to dynamically switch the mode based on the workload in each iteration to achieve better performance.

Optimization of Tensor Operators. TGraph provides two interfaces and five graph operators to shield the graph algorithms from the detailed tensor operators, which enables the seamless integration of optimized tensor operators. We further optimize some of the tensor operators to deal with the unbalanced workload in graph computation. For example, in the GPU implementation of `segment_csr`, only one thread is assigned to each vertex for the reduction of their neighboring data, which will lead to workload imbalance at the thread level. Thus, we further optimize `segment_csr` by assigning different numbers of threads to each vertex based on the size of their neighboring data to achieve work balance.

5 Scaling Strategies

The limited storage space of existing hardware accelerators poses challenges for TGraph in processing large graph datasets. We propose tensor-based graph compression and the out-of-XPU-memory computation strategies to deal with large-scale graphs.

5.1 Tensor-based Graph Compression

Our graph compression module is driven by the fact that many real-world graphs contain a lot of redundancies [6, 66]. Although a number of approaches have been proposed to compress graphs

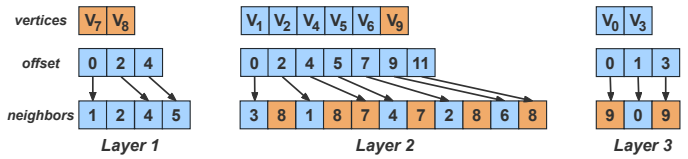


Fig. 4. The compressed graph storage

[4, 5, 7, 12, 16, 59], it is not straightforward to migrate existing methods to our tensor-based computation architecture, as we need to represent the compressed graph in tensors and expand our computation model to deal with the compressed graphs efficiently.

Compressed Graph Storage based on Tensors. Our compression strategy is inspired by [12], i.e., replacing repeated neighbor sequences by virtual vertices. The repeated neighbor sequences above a given frequency can be recursively identified from length 2. Thus, the neighbor set of a virtual vertex may contain other virtual vertices, thereby leading to the nesting of virtual vertices. Given a graph G, we denote the compressed graph as G^c with vertices $V(G) \cup V'$, where V' is the set of virtual vertices. G^c can be represented by multiple-layered CSRs according to the nesting depth of vertices. The nesting depth of vertex v_i is $nest(v_i) = \max_{v_j \in VN(v_i)} nest(v_j) + 1$, where $VN(v_i)$ is the set of virtual neighbors of v_i. If the neighbor set of v_i does not contain virtual vertices, $nest(v_i) = 1$. We use the tensorized CSR to store the subgraph in each layer for the compressed graph G^c. i.e., $G^c = \{G_1^c, G_2^c, \ldots, G_l^c\}$, where l is the maximum nesting depth, and the depths of $neighors_i$ in G_i^c are less than i. Besides, we also need to store a bool tensor $virtulMask$ to indicate whether the vertex is virtual.

Fig. 4 gives the tensorized compression storage for Graph G^c. The repetitive vertex sequences $(1, 2)$ and $(4, 5)$ can be represented as the virtual vertices v_7 and v_8. Since $(4, 5, 6)$ is also a repetitive sequence, we can further abstract a virtual vertex v_9 to represent $(8, 6)$. As shown in Fig. 4, v_7 and v_8 have the nesting level of 1, which means that none of these vertices contain virtual vertex neighbors. Similarly, vertices v_1, v_2, v_4, v_5, v_6, and v_9 have nesting level of 2 while v_0 and $v3$ have the nesting level of 3.

Computation on Compressed Graph. Based on the compression graph G^c, we need to adjust our general computation framework in Alg. 1 to complete the computation on the compressed graphs. Note that the neighbors of a vertex may be distributed in multiple layers. Thus, we need to make sure each vertex can correctly collect the data from all the neighbors in the original graphs in pull mode and each vertex can correctly send the data to all its neighbors in the push node. Now we first take pull mode as an example to illustrate such dependency. The update of vertex v_1 depends on the values of vertex v_3 and the virtual vertex v_8, while the state of v_8 is decided by vertices v_4 and v_5. This means that before updating v_1 at level 2, we need to first update v_8 at level 1. Since the active vertices may be distributed in multiple layers, we need to traverse each layer to get a part of the active subgraph. Specifically, we first check layer 1, call the TENSORIZE and COMPUTE function. Then, all the original and virtual vertices have been updated, and we can safely move to the next layer. By traversing from layer 1 to layer l, we can make sure that when we pull the data from a vertex v in layer i, v has already been updated in previous $i-1$ layers. In the push mode, we will traverse the layers from l to 1, to make sure that the data will be pushed down to the bottom level, we add the virtual vertices that need to be further pushed down into the active vertex set.

5.2 Out-of-XPU-memory Computation

Even in the compression form, large graphs may still not fit into the hardware accelerator (XPU). We further study the out-of-XPU-memory computation to extend TGraph for large graphs, where

Algorithm 6: Edge-balanced Partition

Input : A graph $TG = (vertices, offset, neighbors)$ and the partition number p
Output : A set of subgraphs $TS_1, TS_2, \ldots, TS_p$

1 $Avg_E \leftarrow \texttt{len}(neighbors)/p + 1$;
2 $targetPos \leftarrow \texttt{arange}(0, \texttt{mul}(Avg_E, p+1), Avg_E)$;
3 $realPos \leftarrow \texttt{searchsorted}(offset, targetPos)$;
4 **for** $i = 0$ *to* $p-1$ **do**
5 $\quad subvertices \leftarrow \texttt{arange}(realPos[i], realPos[i+1])$;
6 $\quad TS_{i+1} \leftarrow \texttt{extractSubgraph}\ (TG, subvertices)$;
7 **return** $TS_1, TS_2, \ldots, TS_p$;

the host memory serves as secondary storage, and the graph is partitioned and scheduled into XPU for computation. In the following, we will first introduce our tensor-based graph partition strategy and then elaborate on the pipelined scheduling strategy.

5.2.1 Tensor-based Graph Partition. We provide two strategies to partition the input graph into multiple subgraphs.

Edge-Balanced Partition (EBP). We first propose EBP, which can quickly create a balanced partition of edges within each subgraph. The EBP process can be efficiently implemented using tensor operators, as illustrated in Alg. 6. First, we obtain the ideal average number of edges Avg_E and the ideal partition position $targetPos$ on $neighbors$ (lines 1 to 2). Since the ideal position may partition the neighbors of a vertex into two parts, we need to find the real partition position $realPos$ close to $targetPos$ based on $offset$ to make sure that the neighbors of a vertex are in the same subgraph. We can achieve this using tensor operator `searchsorted`, which performs a binary search to search the closest element in the ordered sequence $offset$ for each element in $targetPos$ in parallel (line 3). The values in $realPos$ are the real boundaries for the vertices partition, based on which we can obtain the vertices set for each subgraph (line 5). Then, based on these vertices, we can extract the subgraph from TG, which consists of three steps, i.e., vertexSelect, neighborSelect, and reconstruct (line 6).

Well-Connected Partition (WCP). For algorithms with computational locality such as WCC, TGraph employs WCP to maximize the internal connectivity within the resulting subgraphs. The main idea is to conduct multiple rounds of Multi-source BFS on graph G to partition G into a relatively large number of connected blocks and then merge these blocks into a number of subgraphs to balance connectivity and subgraph size. Specifically, we use a tensor $blockID$ of size n to record the block ID to which each vertex belongs, which is initially unassigned. Then we randomly select multiple source vertices from unassigned vertices and conduct BFS search, vertices traversed by source vertices belong to the same block where their block Id is the source vertex ID. When the current multiple-source BFS terminates, the number of vertices in the block that exceeds a predefined size will be marked as unassigned. Then, another round of multi-source BFS will be performed on those unassigned vertices. After several rounds of multi-source BFS, some vertices may still remain unassigned. TGraph performs WCC on the subgraph formed by these vertices, with each connected component forming a block. Finally, TGraph merges these blocks into several subgraphs to obtain size balanced and well connected subgraphs. To speed up the process of WCP, we can first partition the graph by EBP and then conduct multi-source BFS and WCC in subgraphs on GPU by a scheduling strategy.

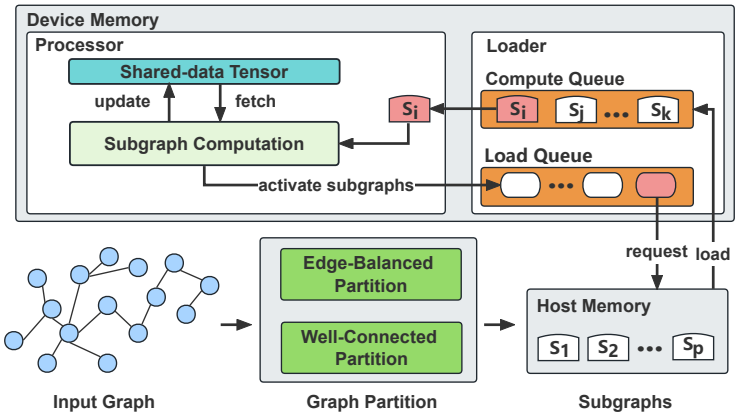


Fig. 5. Workflow of out-of-XPU-memory computation

5.2.2 The pipeline architecture. We first present the workflow of the pipeline architecture and then introduce the scheduling strategy.

Workflow. The workflow of the out-of-XPU-memory computation is shown in Fig. 5. First, we partition an input graph G into several subgraphs $S_1, S_2, ..., S_p$ and put them into the *LoadQueue*. Then, we load the subgraph from the *LoadQueue* into XPU for computation until the computation of the entire graph converges, i.e., the *LoadQueue* is empty. Specifically, the processing of every subgraph S_i consists of two steps: (1) load S_i into XPU, (2) perform computation on S_i on XPU. We propose a pipeline architecture to overlap the subgraph loading and subgraph computation to improve the overall performance, i.e., while a subgraph is undergoing computation, the subsequent subgraph is being loaded from the host to the device. There are two threads responsible for these two steps. The loading thread is responsible for retrieving the ID of the subgraph that needs to be loaded from the *LoadQueue* and loading the corresponding subgraph into *ComputeQueue*. The processing thread retrieves a subgraph from *ComputeQueue* and does computation on this subgraph. We utilize the shared data to record the latest vertex data for inter-subgraph message passing. The shared data is located in device (XPU) memory for efficient data access. For each subgraph, we first fetch the latest vertex data from the shared area previously updated by other subgraphs, then compute the subgraph until convergence, and finally update computed results back to the shared data, thus enabling message passing.

Queue-based Scheduling Strategy. *LoadQueue* is used to store the id of the graphs that need to be scheduled. However, not all subgraphs necessitate computation within an iteration. TGraph selectively schedules the subgraphs requiring updates into XPU for computation, thereby avoiding unnecessary subgraph scheduling. Specially, following the computation of a subgraph, a "push" operator will be executed on this subgraph to update the outgoing neighbors of vertices within the subgraph, and then TGraph identifies the vertices that can be updated. The subgraph which contains these vertices will be pushed into *LoadQueue*, waiting for scheduling. It is challenging to identify the subgraph to which each vertex belongs efficiently by tensor operators due to the absence of support for HashTable. Hence, by renumbering the vertex IDs to attain contiguous vertex IDs within each subgraph, TGraph can efficiently retrieve the subgraph ID for each vertex in a given vertex set using `searchsorted`, as `searchsorted` can locate the closest boundary vertex Id for a given vertex.

Table 3. Graph datasets

Number	Datasets	Abbr.	|V|	|E|
1	cit-Patents	CP	3,774,768	16,518,948
2	hollywood-09	HW	1,139,905	115,031,232
3	indochina-04	IC	7,414,866	304,472,122
4	kron_g500-logn21	KG	2,097,152	182,084,020
5	soc-LiveJournal	LJ	4,847,571	137,987,546
6	soc-orkut	OR	2,997,166	212,698,418
7	soc-sinaweibo	SW	58,655,849	522,642,142
8	soc-twitter	ST	21,297,772	530,051,618
9	arabic-2005	AR	22,744,080	639,999,458
10	it-2004	IT	41,291,594	1,150,725,436
11	Twitter	TW	41,652,231	2,936,637,846
12	gsh-2015	GS	68,660,142	3,561,308,734
13	sk-2005	SK	50,636,151	3,620,126,660

6 Experiments

In this section, we conduct extensive experimental studies to evaluate the performance of TGraph, including comparison with existing systems, scalability, extensibility, performance breakdown, cost-effective analysis, and optimizations evaluation.

Datasets. Table 3 shows the statistics of 13 real-world graph datasets evaluated in this paper, where $|V|$ is the number of vertices and $|E|$ is the number of edges. Datasets 1-8 are graphs of normal size, which are evaluated in most tested cases, while datasets 9-13 are large and super-large datasets for the scalability evaluation. These datasets can be downloaded from SNAP [40], WebGraph [6], or Network Repository [51]. We convert these datasets into symmetric graphs to extend the number of edges [65]. For simplicity, we remove duplicate edges and self-loops from these datasets following the setting of [56]. Note that TGraph can also handle directed/undirected graphs with duplicate edges and self-loops.

Baselines. We compare TGraph with seven state-of-the-art graph systems in three categories: CPU-only [47, 58], GPU-only [3, 18, 65, 70], and CPU-GPU [54] cooperative systems. Correspondingly, TGraph has three versions: TGraph_C, TGraph_G, and TGraph_CG for CPU, GPU, and out-of-XPU-memory computation, respectively.

- Gunrock. Gunrock is a GPU-only graph processing system that adopts the data-centric computation model. The source code of Gunrock 2.0 is available at [25].
- cuGraph. cuGraph is a part of Nvidia rapids data analytics ecosystem which focuses on graph analytics tasks on GPU. We evaluate cuGraph 22.10, which is available at [50].
- Groute. Groute is a GPU-only graph processing system that supports multi-GPU computation. The official GitHub repository of Groute is at [24].
- GraphBlast. GraphBlast is a GPU-only graph system based on linear algebra. The source code is available at [26].
- Subway. Subway is a CPU-GPU cooperative graph system, which only loads the subgraph composed of active edges into GPU memory for computation based on the subgraph-centric model. The source code is available at [2].
- Galois. Galois is a state-of-the-art CPU-only shared memory graph system that implements an amorphous data parallelism computation model. Its GitHub repository is at [32].
- Ligra. Ligra is a lightweight CPU-only graph system on a single machine, and the source code of Ligra can be found at [33]. We use CilkPlus as its multithreading implementation.

Applications. We evaluate five graph algorithms: PageRank (PR), Breadth-First Search (BFS), Single Source Shortest Path (SSSP), Weakly Connected Components (WCC), and Hyperlink-Include

Table 4. Overall comparison of execution time (ms)

Alg.	G	CPU-based system			GPU-based system					
		$TGraph_C$	Galois	Ligra	$TGraph_G$	Gunrock	cuGraph	GraphBlast	Subway	Groute
PR	CP	18.88	**15.61**	61.43	**2.71**	5.02	3.55	12.27	10.84	9.87
	HW	23.24	**12.85**	90.19	**0.48**	3.91	2.79	21.32	28.88	3.91
	IC	69.10	**55.38**	250.19	**3.22**	15.38	6.97	68.78	43.26	20.37
	KG	**45.15**	45.91	120.17	**1.35**	15.64	3.72	-	125.55	24.12
	LJ	49.56	**29.45**	90.31	**2.54**	6.86	5.86	20.74	54.12	17.37
	OR	**41.68**	57.73	249.84	**1.70**	15.46	11.13	-	59.37	22.62
	SW	567.88	**367.42**	1399.69	**51.94**	93.53	67.13	135.33	148.08	129.68
	ST	325.45	**159.35**	544.64	**13.07**	421.76	47.33	oom	171.40	59.37
BFS	CP	288.31	67.35	**48.87**	**6.71**	7.88	56.42	13.86	59.01	13.93
	HW	253.80	27.25	**19.33**	6.12	9.67	59.37	23.11	57.88	**5.15**
	IC	1901.11	**91.24**	450.47	27.72	44.94	227.31	65.88	243.76	**17.37**
	KG	380.31	77.68	**9.81**	**5.74**	26.50	62.9	36.10	142.24	19.54
	LJ	623.08	76.27	**61.78**	**11.41**	15.01	109.33	19.39	85.18	27.41
	OR	515.07	88.87	**29.05**	**7.93**	30.97	83.33	44.12	101.83	27.26
	SW	3680.31	982.34	**468.50**	**77.49**	143.47	457.48	85.11	609.34	139.64
	ST	2838.24	460.85	**170.25**	**47.72**	109.35	335.15	oom	480.64	93.51
HITS	CP	38.83	**35.47**	117.24	**4.74**	9.82	7.89	22.20	27.92	-
	HW	55.28	**27.51**	238.94	**2.34**	24.48	4.81	42.33	239.96	-
	IC	152.35	**91.75**	621.84	**5.85**	176.79	17.04	127.94	627.31	-
	KG	96.66	**77.38**	318.87	**8.27**	41.75	8.71	68.66	417.36	-
	LJ	105.63	**76.28**	170.81	**7.62**	19.58	14.41	40.31	328.63	-
	OR	**86.27**	88.54	458.12	**14.16**	34.24	30.13	91.48	537.72	-
	SW	1126.11	**982.54**	2308.34	**85.50**	330.65	120.23	160.10	1899.58	-
	ST	534.71	**396.34**	1928.65	**41.83**	249.27	80.41	oom	1447.01	-
SSSP	CP	359.21	**90.47**	149.64	20.06	9.51	468.56	75.43	77.39	**16.49**
	HW	358.16	**28.45**	280.39	15.93	11.72	553.30	141.28	91.76	**5.44**
	IC	3167.24	**175.34**	969.12	59.31	47.84	762.27	371.74	420.11	**21.06**
	KG	376.59	135.45	**91.42**	**18.71**	31.44	653.04	74.70	247. 59	23.72
	LJ	841.31	145.38	**111.31**	19.91	**11.79**	678.71	100.61	152.87	24.20
	OR	509.98	**198.54**	310.46	**25.19**	38.45	900.54	265.05	186.05	30.65
	SW	3519.31	**1651.25**	1760.34	**129.83**	270.42	oom	219.61	777.34	168.98
	ST	2662.24	**651.28**	728.30	**81.81**	122.08	oom	oom	885.17	102.60
WCC	CP	303.02	190.25	**168.81**	26.78	**25.73**	342.58	60.85	77.17	33.45
	HW	334.49	214.27	**131.07**	**11.15**	17.68	381.07	98.28	182.09	39.19
	IC	2689.36	**319.68**	732.85	**50.21**	121.13	518.13	262.02	624.38	128.02
	KG	537.78	411.28	**128.51**	**17.26**	47.08	389.98	126.85	176.31	58.87
	LJ	1068.34	271.48	**150.64**	29.43	**18.75**	378.45	-	141.36	42.69
	OR	498.38	**218.24**	229.47	**37.02**	60.38	387.32	-	289.62	67.19
	SW	3278.64	**1650.31**	2457.05	**168.77**	516.79	421.76	-	727.13	299.94
	ST	3218.52	1268.21	**767.45**	**87.58**	350.13	299.14	-	703.34	129.91

Topic Search (HITS), which have been used in many graph processing systems [12, 25, 47, 54, 58]. We select the same vertex as the source across different systems when evaluating BFS and SSSP. We run each algorithm 10 times and report the average running time. All PageRank and HITS times are standardized to a single iteration.

Hardware and Software Setup. We evaluate TGraph on a cloud server equipped with 160GB of RAM, an Intel Xeon Gold 6330 CPU with 14 cores, and an Nvidia GeForce RTX 3090 GPU with 24 GB device memory. The operating system is Ubuntu 20.04. TGraph is implemented with PyTorch 1.11, torch-scatter 2.11, CUDA 11.3. The details of the hardware and software for extensibility testing will be introduced in Section 6.4.

Table 5. Detailed performance profiling

Alg	Systems	G	Time (ms)	L1 Cache Throughput(%)	L2 Cache Throughput(%)	DRAM Throughput(%)	Atomic Cycles
PR	TGraph_G	OR	1.64	49.33	31.41	69.61	0
		SW	51.52	27.18	27.93	37.31	0
	Gunrock	OR	15.26	23.22	29.25	46.77	197,760,472
		SW	91.53	15.58	17.86	25.30	446,767,085
	Groute	OR	21.34	16.63	14.67	29.23	200,380,262
		SW	116.54	11.38	18.74	24.77	532,228,806
BFS	TGraph_G	OR	1.98	54.62	61.35	48.00	0
		SW	17.25	42.32	45.36	73.37	0
	Gunrock	OR	15.79	20.66	24.59	43.37	128,539,250
		SW	119.80	9.75	9.05	13.98	127,795,237
	Groute	OR	15.95	17.61	25.90	41.30	128,861,613
		SW	72.69	15.89	18.70	26.22	330,390,219

6.1 Overall Performance Comparison

Overall Performance Evaluation. We compare the overall performance of the evaluated systems on both CPU and GPU. The execution time is reported in Table 4, where bolded values highlight the best result, underlined values represent the second-best result, "-" denotes that the system cannot run successfully, and oom represents out-of-memory. First, the GPU-based systems generally outperform the CPU-based systems. However, for computationally sparse graph applications such as BFS, where the parallelism of GPUs cannot be fully exploited, the CPU-based systems perform comparably to the GPU-based systems and, in some cases, even better. Second, among the CPU-based systems, TGraph_C do not achieve the best performance, which is within expectation, as the Pytorch optimization for CPU is very limited. Some tensor operators such as `scatter_reduce` and `segment_csr` even fail to utilize multi-core parallelization. Third, among the GPU-based systems, TGraph_G performs best in most cases and is still competitive compared to the optimal results in the remaining cases. Specifically, TGraph_G achieves significant improvements for PageRank and HITS.

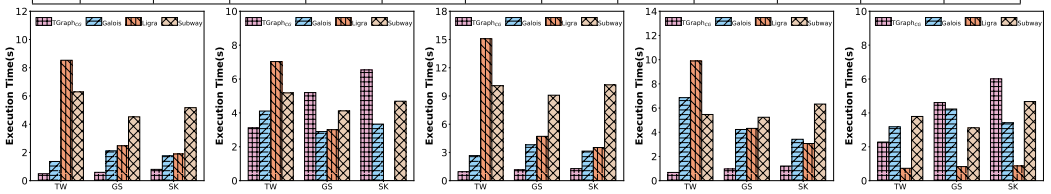
Detailed Performance Profiling. We further analyze the performance of TGraph_G by profiling it and the two most competitive baselines (Gunrock and Groute) using Nsight Compute. Here, we select two representative algorithms, PR and BFS, and profile the iteration with the most active vertices at the kernel level. As shown in Table 5, we profile systems on the following metrics: the execution time of the profiled kernels, memory throughput at different levels including L1 Cache Throughput, L2 Cache Throughput, and DRAM Throughput, and atomic cycles which is the number of clock cycles involving atomic operators. Gunrock and Groute involve numerous atomic operations as they only use the push mode while TGraph_G can switch to the pull computation mode to avoid atomic operators. Since excessive atomic operations will affect the memory access efficiency, TGraph_G outperforms Gunrock and Groute in L1 Cache Throughput, L2 Cache Throughput, and DRAM Throughput. Such memory throughput leads to better performance as BFS and PR are both memory-intensive tasks whose performance is mainly affected by memory access. For BFS on SW datasets, Gunrock and Groute take a total time of 143.47 ms and 139.64 ms, while the iteration with the most active vertices cost a time of 119.80 ms and 72.69 ms, respectively, occupying the majority of the time. Consequently, employing the pull mode during these intense computation iterations can effectively improve the overall performance of graph applications.

6.2 Scalability Evaluation

We evaluate the scalability of TGraph in handling large graphs, including graph compression and out-of-XPU-memory computation.

Table 6. Graph compression evaluation on TGraph_G

G	Depth	Comp. Ratio	Execution Time (ms)									
			PR		BFS		HITS		SSSP		WCC	
			Origin	Comp.	Origin	Comp.	Origin	Comp.	Origin	Comp.	Origin	Comp.
AR	11	4.35	66.24	11.83	1034.24	679.81	108.24	32.81	2228.29	822.21	1348.47	674.31
IT	11	4.65	-	18.35	-	962.65	-	37.84	-	1167.18	-	714.14



(a) PR (b) SSSP (c) HITS (d) WCC (e) BFS

Fig. 6. Comparison of execution time (s) for the out-of-XPU-memory computation

Graph Compression. We evaluate TGraph_G on AR and IT that have relatively large sizes but cannot fit into the XPU memory. The results are shown in Table 6, where depth represents the maximum nesting depth of virtual vertices in the compressed graph, and the compression ratio represents the ratio of storage size of the original graph to that of the compressed graph. We list the results of five graph applications. TGraph_G achieves a better speedup in PR and HITS because the computational load of PR and HITS is heavier, allowing better utilization of data redundancy. Moreover, the original graph of IT contains around two billion edges and cannot be handled by TGraph_G due to the limited device memory space. However, the compressed form of IT can be efficiently processed by TGraph_G, indicating the capability of graph compression and compressed graph computation for handling large-scale graphs.

Out-of-XPU-memory Computation. We evaluate the effectiveness of our out-of-XPU-memory computation mechanism by comparing TGraph_CG with two CPU-based graph systems (Ligra [58] and Galois [47]) and an out-of-GPU-memory based graph system Subway[54]. Specifically, we evaluate five graph applications on TW, GS, and SK, which cannot fit into the device (NVIDIA RTX 3090 with 24GB memory). Fig. 6 reports the end-to-end execution time for these four systems, where Ligra fails to execute the SSSP on SK due to the high memory demand. We have the following observations. (1) Among the evaluated algorithms, TGraph_CG achieves the most significant improvements for PR and HITS. Taking PR as an example, TGraph_CG is up to 3.57×, 18.42×, and 12.83× faster than Galois, Ligra, and Subway, respectively. This shows that pipeline architecture in TGraph_CG can achieve superior end-to-end performance through overlapping subgraph computation and transfer, especially for computationally intensive applications such as PR and HITS. (2) TGraph_CG also performs the best for the WCC algorithm due to the WCP graph partition method, which ensures a high intra-subgraph connectivity and substantially reduces the computational workload of WCC. (3) For BFS and SSSP, Ligra and Galois exhibit the best performance, respectively. Subway costs more time as it includes both data transfer and subgraph generation. TGraph_CG does not perform well on BFS and SSSP as these two algorithms need more iterations to converge in the out-of-XPU-memory computation, which will cause more redundant vertex updates. Overall, there is no best choice among the systems for out-of-XPU-memory computation. Among the five algorithms, TGraph_CG performs the best for 3 out of 5, and Ligra performs the best for 1 out of 5.

6.3 Extensibility Evaluation

We evaluate the extensibility of TGraph by deploying and running it on different hardware backends and software frameworks.

Table 7. Execution time (ms) on machine 1

Systems	PR			HITS			BFS		
	CP	HW	LJ	CP	HW	LJ	CP	HW	LJ
TGraph	**4.52**	**8.31**	**10.47**	**8.31**	**16.12**	**20.09**	135.21	139.54	322.73
Ligra	175.21	60.93	230.27	176.38	104.82	236.84	**42.07**	**12.51**	**37.62**
Galois	29.58	20.41	54.56	102.71	32.16	146.65	79.82	53.21	114.24

Table 8. Execution time (s) on machine 2

Systems	PR			HITS			WCC		
	CP	HW	LJ	CP	HW	LJ	CP	HW	LJ
$TGraph_G$	**0.069**	0.121	**0.169**	**0.121**	0.246	**0.327**	0.681	0.573	0.772
Ligra	0.169	**0.114**	0.283	0.268	**0.204**	12.58	**0.053**	**0.015**	**0.043**
Networkx	26.228	88.201	68.681	28.328	66.588	63.877	1.945	0.877	2.263

Table 9. Execute time (ms) on machine 3

systems	PR			SSSP			BFS		
	CP	HW	LJ	CP	HW	LJ	CP	HW	LJ
$TGraph_G$	**4.78**	**1.39**	**2.75**	25.71	16.23	25.51	**7.58**	7.84	**13.98**
Gunrock	8.23	5.14	6.88	**14.08**	15.23	**17.64**	11.02	13.09	15.22
Groute	22.69	13.79	40.49	20.87	**8.73**	26.99	16.76	**7.53**	24.83
GraphBlast	16.92	27.36	27.39	45.04	39.49	51.02	28.96	34.16	41.49

Table 10. Execute time (ms) on different framework

Framework	PR	BFS	HITS	SSSP	WCC
Tensorflow	12.55	278.91	21.94	298.25	289.67
Pytorch	1.70	7.93	14.16	25.19	37.02

Extensibility on Hardware Backends. Besides Nvidia GeForce RTX 3090 GPU, we also deploy and run $TGraph_G$ on two desktop computers and one cloud server with different hardware backends: Machine 1 - A machine equipped with an AMD 7700XT GPU, a 12400f CPU, and 16GB memory; Machine 2 - A Mac Mini equipped with an M2 chip and 16GB memory; Machine 3 - A machine equipped with a datacenter-level GPU V100. We conduct experiments using PyTorch on these three machines. (1) For machine 1, we compare $TGraph_G$ accelerated by AMD 7700XT with two CPU-based graph systems (Ligra and Galois) since existing GPU-based systems are developed on CUDA and thus cannot run on this machine. We show the results in Table 7, where $TGraph_G$ performs the best for PR and HITS. The performance of $TGraph_G$ on BFS is not the best, and it might be because the operator `scatter_reduce` used in BFS on the ROCm backend is not well optimized and CPU-based systems inherently perform well for BFS. (2) For machine 2, we compare $TGraph_G$ with Ligra and Networkx [27], as we fail to compile Galois successfully. As shown in Table 8, $TGraph_G$ and Ligra exhibit similar performance for PR and HITS, whereas Ligra outperforms $TGraph_G$ in BFS. The underlying reason might also be the inadequate optimization of PyTorch on the MPS backend. However, $TGraph_G$ still outperforms Networkx in all the cases. (3) For machine 3, we compare $TGraph_G$ with the most competitive three GPU-based baselines Gunrock, Groute, and GraphBlast. As shown in Table 9, the performance of $TGraph_G$ on the datacenter-level GPU remains excellent, consistent with the results obtained on Nvidia RTX 3090. Overall, we can successfully deploy and run $TGraph_G$ on different hardware backends and outperform the state-of-the-art in-memory systems in most cases.

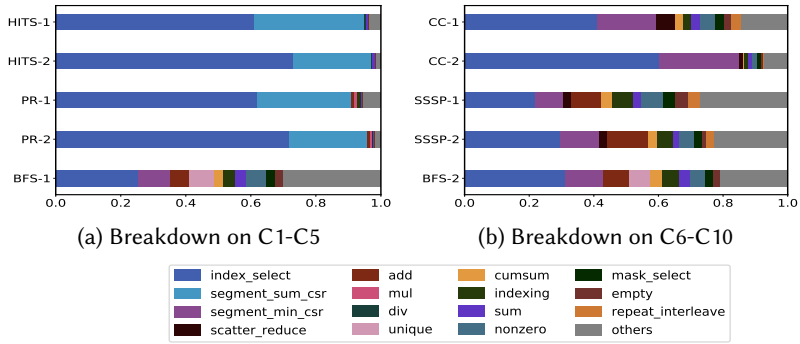
Extensibility on Software Frameworks. To verify the extensibility of TGraph across software frameworks, we also built $TGraph_G$ on TensorFlow by slightly modifying the code. Table 10 shows the execution time of five applications on PyTorch and TensorFlow for the dataset OR. Compared to PyTorch, TensorFlow is much slower, indicating the inadequate optimization of the required tensor operators within TensorFlow.

The above analysis shows that TGraph can perform graph computation across different hardware backends and software frameworks. With the ongoing advancement of DL, we believe that TCRs will provide more efficient operators on diverse heterogeneous hardware, enabling TGraph to accelerate graph computation.

6.4 Performance Breakdown

Fig. 7 shows the breakdown of the execution time for the ten computation combinations formed by two datasets (LJ and OR) and five graph applications at the level of tensor operators. From this figure, we have the following observations.

(1) Across all computation combinations, the selection operators (`index_select` and `indexing`) account for the majority of the execution time. In $TGraph_G$, the select operator dominates the execution time due to its heavy usage in both TENSORIZE and COMPUTE phases, including the vertexSelect, neighborSelect, and aggregate graph operators. The proportion of the select operator



(a) Breakdown on C1-C5 (b) Breakdown on C6-C10

Fig. 7. Execution time breakdown of tensor operators

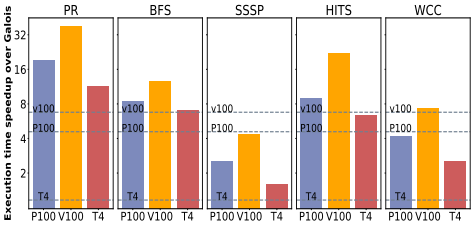


Fig. 8. Comparison of cost-effective

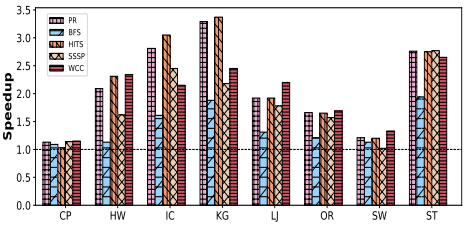


Fig. 9. Evaluation of the optimized operator

to some extent reflects the computational workload of the algorithm. For instance, the average select operator overhead in PR reaches 68%, which implies a substantial need for data selection.

(2) Operator overhead is influenced by both data scale and data distribution. In HITS, the proportion of `segment_csr` varies between the two datasets, while the overhead of `index_select` scales with data size, implying that the overhead of `segment_csr` is also associated with data distribution. `segment_csr` is used to aggregate data, the performance of which diminishes with highly imbalanced data distribution within each group, leading to unbalanced thread workloads. Effectively implementing the distribution-sensitive operators is worth to be explored in the future.

6.5 Cost-effective Analysis

Here, we provide a comparison of cost-effectiveness between TGraph_G and Galois on CPU. The cost metric is the price of the VM in Azure. For Galois, we select the CPU-only VM D2ds_v5 with 8 cores and 32GB memory. For TGraph_G, we choose the VM equipped with GPU, matching the CPU core number and memory size of D2ds_v5. Specifically, we select three GPU VMs: NC4as_T4_v3 (with an NVIDIA T4 GPU), NC6s_v2 (with an NVIDIA P100), and NC6s_v3 (with an NVIDIA V100). The cost of these three VMs is that of D2ds_v5 multiplied by 1.15×, 4.6×, and 6.6×, respectively. Fig. 8 shows the speedup of TGraph_G compared to Galois, where the dashed lines represent the execution time required by TGraph_G to be more cost-effective than Galois on three VMs. The execution time of each algorithm is the total time over datasets 1-8. Taking NC4as_T4_v3 as an example, TGraph_G is more cost-effective than Galois only when its speedup on this VM compared to Galois exceeds 1.15×. As shown in Fig. 8, TGraph_G is more cost-effective compared to Galois: 5 out of 5 algorithms for T4, 3 out of 5 algorithms for P100, and 4 out of 5 algorithms for V100.

6.6 Optimization Techniques Evaluation

We evaluate the two optimization strategies in Section 4.5 and discuss the effect of the DL compiler technique on TGraph_G.

Table 11. Execution time(ms) of different computation modes

Datasets		CP	HW	IC	KG	OR	LJ	SW	ST
BFS	TGraph_{pull}	24.48	19.61	94.72	31.87	44.99	40.46	294.58	327.73
	TGraph_{push}	35.05	28.31	77.47	38.99	51.03	39.01	238.57	153.46
	TGraph_{G}	**6.71**	**6.12**	**27.27**	**5.74**	**7.93**	**11.41**	**77.49**	**47.47**
SSSP	TGraph_{pull}	34.98	19.39	91.72	32.24	46.58	38.96	169.66	319.34
	TGraph_{push}	39.56	32.63	89.31	45.76	58.39	44.06	256.55	167.87
	TGraph_{G}	**20.06**	**15.93**	**59.31**	**18.71**	**25.19**	**19.91**	**129.83**	**81.81**
WCC	TGraph_{pull}	54.12	35.56	108.63	50.94	322.24	230.56	391.38	423.97
	TGraph_{push}	155.17	127.73	-	164.69	92.91	65.73	-	-
	TGraph_{G}	**26.78**	**11.15**	**50.21**	**17.26**	**37.02**	**29.43**	**168.77**	**87.58**

Table 12. Effect of DL compiler on neighborSelect

	CP	HW	IC	KG	LJ	OR	SW	ST
neighborSelect	3.31 ms	3.77 ms	11.94 ms	1.57 ms	3.79 ms	2.57 ms	5.05 ms	7.21 ms
$\text{neighborSelect}_{\text{compile}}$	2.61 ms	3.01 ms	7.86 ms	1.34 ms	3.11 ms	2.14 ms	4.61 ms	5.21 ms
SpeedUp	1.12×	1.25×	1.52×	1.17×	1.22×	1.21×	1.09×	1.38×

Evaluation of the Computation Mode. To evaluate the effectiveness of the computation mode adopted by TGraph_{G}, we compare TGraph_{G} with its two variants, $\text{TGraph}_{\text{push}}$ and $\text{TGraph}_{\text{pull}}$, which adopt the push-only and pull-only computation mode, respectively. Table 11 shows the performance of three representative algorithms: BFS, SSSP, and WCC. $\text{TGraph}_{\text{push}}$ fails to complete the computation of WCC on IC, SW, and ST due to the large memory demand for *groupID* and *groupData* as all vertices are active during the first iteration. In the remaining cases, TGraph_{G} outperforms both $\text{TGraph}_{\text{push}}$ and $\text{TGraph}_{\text{pull}}$, which proves the effectiveness of the dynamic mode switch strategy.

Evaluation of the Tensor Operator Optimization. Fig. 9 shows the overall speedup of TGraph_{G} when using the optimized tensor operator on the five graph applications. Generally, TGraph_{G} with optimized *segment_csr* gains the speedup across all test datasets, with particularly significant improvement on the IC, KG, and ST, which is consistent with their highly skewed degree distribution. Among the five algorithms, BFS and SSSP have a relatively lower speedup because they adopt push mode in most iterations, which does not involve *segment_csr* while the remaining three algorithms achieve significant improvement as they use pull mode in more iterations, which can be accelerated by optimized *segment_csr*.

Effect of the DL Compiler. We evaluate the effect of the built-in compiler in PyTorch on our system performance. We only compile the graph operator `neighborSelect` as the custom tensor operator `segment_csr` used in graph operator `aggregate` cannot be compiled [63], and the cost of the remaining operators is negligible. Table 12 shows the total execution time of neighborSelect with/whithout compilation for the BFS algorithm, where the speedup is 1.12×-1.52× across these datasets. However, neighborSelect only accounts for an average of 20.54 % of the total execution time of graph algorithm, and the compilation overhead far exceeds the total execution time of the algorithm. Thus, TGraph does not employ DL compilers by default.

7 Related Work

CPU-based Graph Processing Systems. (1) *shared-memory* graph systems process graphs in a single machine, which consists of one processing unit (host) with one or more CPU cores and physical memory shared across all the cores. The *in-memory* systems can load the whole graph into memory for computation, such as Ligra [58, 59], Galois [47], GraphMat [61], etc. The *out-of-core* systems only maintain a small portion of vertices/edges in memory, focusing on partitioning graphs to reduce disk I/O, such as GraphChi [39], X-Stream [53], GridGraph [76], FlashGraph [73], NXgraph

[14], Mosaic [42], MiniGraph [77], etc. (2) *distributed* graph systems aim to accommodate larger graphs into multiple machines through load balance and communication reduction. Many parallel computation models were developed, including *vertex-centric* model (Pregel [43], Giraph [21], Powergraph [22], etc.), *edge-centric* model (X-stream [53], Chaos [53], etc.), *block/subgraph-centric* model (Giraph++ [62], Grape [17], GraphScope [67] [29], etc.), and *linear-algebra based* model (CombBLAS [8], GraphBLAS [34], etc.).

GPU/FPGA-accelerated Graph Processing Systems. A number of graph systems have been proposed to leverage the parallelism of GPUs. Medusa [75] develops an edge-message-vertex model tailored for simplifying parallel graph processing on GPUs. CuSha [37] is a vertex-centric system utilizing G-shards and concatenated windows to address irregular memory access. Frog [57] proposes a hybrid coloring model to partition vertices, enabling asynchronous computation. Gunrock [65] introduces a data-centric model to process vertex/edge subsets relevant to computation on GPU and several load-balancing strategies at different granularity. Groute [3] proposes an asynchronous programming model for irregular graph algorithms on multi-GPUs. cuGraph [18] is a vertex/edge-centric system with a new data structure to store and partition graphs. GraphBLAST [70] is a linear-algebra-based framework on GPUs which explored the input and output sparsity to reduce memory access. Some CPU-GPU collaborated graph systems have also been proposed, such as TOTEM [20], FinePar [71], Scaph [74], Largegraph [72], and Subway [54]. GraphGen [48], FPGP [15], and ThunderGP [11] are graph systems accelerated by FPGAs. The above systems are tailored for specific hardware, which cannot be easily migrated to other hardware accelerators.

Tensor-based Data Processing. The development of DL has created a growing demand for tensor computation acceleration. Hence, many DL frameworks as well as their compilers and runtimes are developed, which are referred to as TCRs. To take a free ride on the development of TCRs, Hummingbird [46] is proposed for accelerating traditional machine learning models by compiling their pipelines into tensor computations. [38] have made an initial attempt to transform PageRank into tensor operators on TCRs. TCUDB [31] maps compute-intensive database queries into matrix multiplication by leveraging NVIDIA Tensor Core Units. TQP [28] transforms relational queries into tensor operators based on TCRs and it is extended to support mixed SQL/ML workloads [19]. Despite the above progress on non-DL data processing tasks, building an efficient graph processing system based on tensors with high expressivity, extensibility, and scalability is still challenging.

8 Conclusion

In this paper, we propose the first tensor-based graph processing framework, TGraph, which can be smoothly deployed and run on different DL frameworks and hardware accelerators with high expressivity, extensibility, and scalability. We achieved this by: designing a tensor-centric computation model, which takes tensors as the fundamental unit of graph computation to maximize the parallelism of tensor operators; abstracting a set of graph operators, which can shield the computation model from the detailed tensor operators and support the easy implementation of graph algorithms; proposing scaling strategies including graph compression and out-of-XPU-memory computation to handle large-scale graphs. Our extensive experimental results show that TGraph not only outperforms the existing seven state-of-of-the art graph systems in most cases, but also can run on multiple DL frameworks and hardware backends.

Acknowledgments

The work was supported in part by grants of National Natural Science Foundation of China (No. 62272353 and No. 62276193), the Research Grants Council of Hong Kong (No. 14205520), and Huawei Cloud Database Innovation Lab. Yuanyuan Zhu and Hao Zhang are corresponding authors.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.

[2] AutomataLab. 2020. Subway. https://github.com/AutomataLab/Subway

[3] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *PPoPP*, Vivek Sarkar and Lawrence Rauchwerger (Eds.). ACM, 235–248.

[4] Maciej Besta and Torsten Hoefler. 2018. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799* (2018).

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *WWW*. 587–596.

[6] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW*. 595–602.

[7] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. 2009. k2-Trees for Compact Web Graph Representation. In *SPIRE (Lecture Notes in Computer Science, Vol. 5721)*. 18–30.

[8] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*. 578–594.

[11] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based Graph Processing Framework on FPGAs. In *FPGA*. 69–80.

[12] Zheng Chen, Feng Zhang, Jiawei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. 2023. CompressGraph: Efficient Parallel Graph Analytics with Rule-Based Compression. *Proc. ACM Manag. Data* 1, 1 (2023), 4:1–4:31.

[13] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 85–98.

[14] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. NXgraph: An efficient graph processing system on a single machine. In *ICDE*. 409–420.

[15] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *FPGA*. 105–110.

[16] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1535–1544.

[17] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. 2017. GRAPE: Parallelizing Sequential Graph Computations. *Proc. VLDB Endow.* 10, 12 (2017), 1889–1892.

[18] Alex Fender, Brad Rees, and Joe Eaton. 2022. Rapids cugraph. In *Massive Graph Analytics*. Chapman and Hall/CRC, 483–493.

[19] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, and Matteo Interlandi. 2023. The Tensor Data Platform: Towards an AI-centric Database System. In *CIDR*.

[20] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *PACT*. 345–354.

[21] Apache Giraph. 2012. http://giraph.apache.org.

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*. 17–30.

[23] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. 599–613.

[24] groute. 2020. Groute: An Asynchronous Multi-GPU Programming Framework. https://github.com/groute/groute

[25] gunrock. 2016. Gunrock: CUDA/C++ GPU Graph Analytics. https://github.com/gunrock/gunrock

[26] gunrock. 2022. GraphBLAST. https://github.com/gunrock/graphblast

[27] Aric Hagberg, Pieter J Swart, and Daniel A Schult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).

[28] Dong He, Supun Chathuranga Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* 15, 11 (2022), 2811–2825.
[29] Tao He, Shuxian Hu, Longbin Lai, Dongze Li, Neng Li, Xue Li, Lexiao Liu, Xiaojian Luo, Bingqing Lyu, Ke Meng, Sijie Shen, Li Su, Lei Wang, Jingbo Xu, Wenyuan Yu, Weibin Zeng, Lei Zhang, Siyuan Zhang, Jingren Zhou, Xiaoli Zhou, and Diwen Zhu. 2024. GraphScope Flex: LEGO-like Graph Computing Stack. In *SIGMOD*. ACM, 386–399.
[30] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. 2018. Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges. *ACM Comput. Surv.* 51, 3 (2018), 60:1–60:53.
[31] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. Tcudb: Accelerating database with tensor processors. In *SIGMOD*. 1360–1374.
[32] IntelligentSoftwareSystems. 2013. Galois. https://github.com/IntelligentSoftwareSystems/Galois
[33] jshun. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. https://github.com/jshun/ligra
[34] Jeremy Kepner. 2017. Graphblas mathematics-provisional release 1.0. *GraphBLAS. org, Tech. Rep.* (2017).
[35] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *HPEC*. 1–9.
[36] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra.* SIAM.
[37] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *HPDC*. 239–252.
[38] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: An abstraction for general data processing. *Proc. VLDB Endow.* 14, 10 (2021), 1797–1804.
[39] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*. 31–46.
[40] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[41] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2021. The Deep Learning Compiler: A Comprehensive Survey. *TPDS* 32, 3 (2021), 708–727.
[42] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *EuroSys*. 527–543.
[43] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146.
[44] Tim Mattson, David Bader, Jon Berry, Aydin Buluc, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, et al. 2013. Standards for graph algorithm primitives. In *HPEC*. 1–2.
[45] Microsoft. 2022. ONNX Runtime. https://github.com/microsoft/onnxruntime
[46] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *OSDI*. 899–917.
[47] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.
[48] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. 2014. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *FCCM*. 25–28.
[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NIPS*. 8024–8035.
[50] Rapids. 2022. cuGraph. https://github.com/rapidsai/cugraph/tree/branch-22.04
[51] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com
[52] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: scale-out graph processing from secondary storage. In *SOSP*. 410–424.
[53] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *SOSP*. 472–488.
[54] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *EuroSys*. ACM, 12:1–12:16.
[55] Semih Salihoglu and Jennifer Widom. 2013. GPS: a graph processing system. In *SSDBM*. 22:1–22:12.
[56] Peter Sanders, Christian Schulz, and Dorothea Wagner. 2014. Benchmarking for graph clustering and partitioning. *Encyclopedia of social network analysis and mining Springer* (2014).

[57] Xuanhua Shi, Junling Liang, Sheng Di, Bingsheng He, Hai Jin, Lu Lu, Zhixiang Wang, Xuan Luo, and Jianlong Zhong. 2015. Optimization of asynchronous graph processing on GPU with hybridvcoloring model. In *PPoPP*. 271–272.
[58] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*. 135–146.
[59] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *DCC*. 403–412.
[60] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. 2018. Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction. *Proc. VLDB Endow.* 12, 2 (2018), 154–168.
[61] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow.* 8, 11 (2015), 1214–1225.
[62] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.* 7, 3 (2013), 193–204.
[63] torch_scatter. 2022. compilation issue of torch_scatter. https://github.com/rusty1s/pytorch_scatter/issues/440
[64] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. 2013. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*.
[65] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *PPoPP*. 11:1–11:12.
[66] Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393 (1998), 440–442.
[67] Jingbo Xu, Zhanning Bai, Wenfei Fan, Longbin Lai, Xue Li, Zhao Li, Zhengping Qian, Lei Wang, Yanyan Wang, Wenyuan Yu, and Jingren Zhou. 2021. GraphScope: A One-Stop Large Graph Processing System. *Proc. VLDB Endow.* 14, 12 (2021), 2703–2706.
[68] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *Proc. VLDB Endow.* 7, 14 (2014), 1981–1992.
[69] Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. 2018. GraphD: Distributed Vertex-Centric Graph Processing Beyond the Memory Limit. *TPDS* 29, 1 (2018), 99–114. doi:10.1109/TPDS.2017.2743708
[70] Carl Yang, Aydin Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Softw.* 48, 1 (2022), 1:1–1:51.
[71] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. 2017. FinePar: irregularity-aware fine-grained workload partitioning on integrated architectures. In *CGO*. 27–38.
[72] Yu Zhang, Da Peng, Xiaofei Liao, Hai Jin, Haikun Liu, Lin Gu, and Bingsheng He. 2021. LargeGraph: An Efficient Dependency-Aware GPU-Accelerated Large-Scale Graph Processing. *TACO* 18, 4 (2021), 58:1–58:24.
[73] Da Zheng, Disa Mhembere, Randal C. Burns, Joshua T. Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *FAST*. 45–58.
[74] Long Zheng, Xianliang Li, Yaohui Zheng, Yu Huang, Xiaofei Liao, Hai Jin, Jingling Xue, Zhiyuan Shao, and Qiang-Sheng Hua. 2020. Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven Differential Scheduling. In *ATC*. 573–588.
[75] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *TPDS* 25, 6 (2014), 1543–1552.
[76] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *ATC*. 375–386.
[77] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. 2023. MiniGraph: Querying Big Graphs with a Single Machine. *Proc. VLDB Endow.* 16, 9 (2023), 2172–2185.

Received July 2024; revised September 2024; accepted November 2024