GraFlex: Flexible Graph Processing on FPGAs through Customized Scalable Interconnection Network

Chunyou Su
csuae@connect.ust.hk
The Hong Kong University of Science and Technology
Kowloon, Hong Kong

Linfeng Du
linfeng.du@connect.ust.hk
The Hong Kong University of Science and Technology
Kowloon, Hong Kong

Tingyuan Liang
tliang@connect.ust.hk
The Hong Kong University of Science and Technology
Kowloon, Hong Kong

Zhe Lin
linzh235@mail.sysu.edu.cn
Sun Yat-sen University
ShenZhen, Guangdong
China

Maolin Wang
maolinwang@ust.hk
ACCESS
New Territories, Hong Kong

Sharad Sinha
sharad@iitgoa.ac.in
Indian Institute of Technology Goa
Goa, India

Wei Zhang
eeweiz@ust.hk
The Hong Kong University of Science and Technology
Kowloon, Hong Kong

ABSTRACT

Graph processing system design has been widely considered to be a challenging topic due to the mismatch between the computational throughput requirement and the memory bandwidth. Recent works try to deliver better graph processing systems by taking advantage of application-specific architectures and emerging high-bandwidth memory on FPGAs. However, there is still ample room for improvements regarding flexibility, scalability, and usability.

This paper presents GraFlex, a flexible scatter-gather graph processing framework on FPGAs with scalable interconnection networks. It adopts the Bulk-Synchronous Parallel (BSP) paradigm for global control and synchronization, enabling rapid deployment of performant graph processing systems through HLS-based design flows. GraFlex conducts software-hardware co-optimization to boost system performance. It configures the compact graph format, partition scheme, and memory channel allocation strategy to support scalable designs. Resource-efficient multi-stage butterfly interconnection network achieves on-device data communication and facilitates throughput matching. To handle fragmented memory requests, we propose coalesced memory access engines to improve bandwidth utilization. GraFlex is comprehensively evaluated with various graph applications and real-world datasets. Our results show up to **2.09**× average speedup in traversal throughput over the existing state-of-the-art work with a non-negligible reduction in power and resource consumption. A case study of the breadth-first search (BFS) application shows a **6.58**× speedup in average algorithm throughout with proper implementation choices enabled by the scatter-gather mechanism implemented. The BFS study also reports an almost linear throughput scaling versus the number of processing elements (PEs) and memory channels.

CCS CONCEPTS

• **Computer systems organization** → **Interconnection architectures**; • **Theory of computation** → **Graph algorithms analysis**; • **Hardware** → **Hardware accelerators**.

KEYWORDS

Graph Processing, FPGA, Interconnection Network

ACM Reference Format:
Chunyou Su, Linfeng Du, Tingyuan Liang, Zhe Lin, Maolin Wang, Sharad Sinha, and Wei Zhang. 2024. GraFlex: Flexible Graph Processing on FPGAs through Customized Scalable Interconnection Network. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3626202.3637573

1 INTRODUCTION

Graph-based models are a practical way to represent a wide range of real-world relationships, such as social networks, transportation networks, and website links. However, designing hardware accelerators for efficient graph processing is a formidable challenge. Graph applications are typically memory-bound due to their irregular memory access patterns and the significant variation in memory traces among different processing elements (PEs). Consequently, memory bandwidth utilization could be extremely low since the massive consecutive access pattern favored by DDR memory does not exist. Although cutting-edge devices with high-bandwidth memory (HBM) alleviate the issue by exposing more concurrent memory channels to the logic, how to take full advantage of them is still problematic, as the high bandwidth can be best utilized only in the point-to-point memory channel to PE mapping [33]. Another challenge is rooted in the sophisticated implementation techniques for parallel graph applications on hardware, including task partitioning, global synchronization, and run-time load balancing. Although many previous works show promising performance with GPU acceleration [13, 27, 28], they are implemented at the cost of high energy consumption, mainly due to inefficient memory accesses.

One solution for efficient graph processing is application-specific architectures, making FPGA an ideal platform due to its fast design customization and flexible memory access manipulation, contributing to higher power efficiency accordingly. Early-stage studies

Table 1: A Review of Recent Graph Processing Frameworks on FPGAs

Works	Program. Model	Interconnect	PL[1]	App[2]	Graph Fmt.
Graphlily [18]	Linear Algebra	Crossbar	HLS	4	CPSR
GraphScale [12]	Vertex-Centric	Two-level Crossbar	HDL	3	Inverse CSR
ThunderGP [6, 7]	GAS	Shuffling Logic [4]	HLS	7	COO
ReGraph [5]	GAS	Multi-stage Butterfly	HLS	3	COO
ACTS [21]	GAS	*None*	HLS	4	ACTPACK
GraFlex	Scatter-Gather	Multi-stage Butterfly	HLS	6	CSR

[1] Required Programming Language for development.
[2] Number of evaluated graph applications with the system.

of graph processing on FPGAs achieved high performance with RTL design, targeting parallel graph applications originally executed on general-purpose computing clusters [15]. For example, breath-first search (BFS) [1, 3, 16], single-source shortest path (SSSP) [24, 34], and sparse matrix-vector multiplication (SpMV) [17, 20], where many application-specific optimization techniques are applied. There are also graph processing frameworks on FPGAs to support more than one graph applications [9, 30, 35]. Nevertheless, the conventional RTL-based design approach requires non-trivial efforts either to adapt these frameworks to new applications or to process graphs of noticeably different scales. Besides, early works used to buffer an entire graph in the scratchpad memory to overcome the memory wall problem [2], making them less attractive when facing real-world graph processing requirements.

More recently, the introduction of high-level synthesis (HLS) design flow and HBM devices promoted the revolution in graph processing on modern FPGAs. The HLS flow drastically reduces the manual effort required and speeds up the design iteration process [7]. Another breakthrough in graph processing on FPGAs comes from the integration of emerging HBM devices with programmable logic. With cutting-edge HBM-equipped FPGA acceleration boards, pioneer works show significant performance improvement over traditional counterpart platforms with multiple DDR memory banks [6, 18, 26].

We scrutinize recent works on FPGA-based graph processing frameworks and summarize critical observations in Table 1. First, in terms of the programming model, Graphlily [18] borrows the linear algebra application programming interface (API) from GraphBLAS [23], and GraphScale adopts the classic vertex-centric model. Both of them exploit the vertex-level parallelism. The remaining three works adhere to the Gather-Apply-Scatter (GAS) model, benefiting from the edge-level parallelism. Second, interconnect is an indispensable component of graph processing systems in most cases for on-device data redirection. Besides, although HLS-based design flow grants faster deployment in recent works, they still require noticeable manual effort to adapt more applications to their framework. Lastly, sparse graph representations are utilized to support the specific parallelism type, either vertex-centric (CPSR & Inverse CSR) or edge-centric (COO & ACTPACK).

In light of the observations above, we outline the remaining challenges to be resolved in terms of flexibility, scalability, and usability. Firstly, adhering to the linear algebra or the GAS model may limit the frameworks' expressive efficiency to certain graph applications [22]. Existing works with the most flexible vertex-centric model [11, 12] are implemented in RTL, which hinders fast deployment compared to the HLS-based flows. Besides, prior works' customized crossbars and shuffle logic are resource-intensive and require non-trivial effort to scale up. Moreover, although works using edge parallelism [5–7, 21] bring superior performance, they impose much higher storage requirements ($2|E|$) than the vertex-parallel ones ($|V|+|E|$) [12, 18], which drastically limits their scalability regarding the input graph size.

This paper proposes GraFlex, a flexible FPGA-based graph processing framework with scalable interconnection networks. GraFlex allows users to easily create performant parallel graph processing systems using entry-level high-level language (C/C++) descriptions under the Bulk-Synchronous Parallel (BSP) [32] execution paradigm. It adopts the scatter-gather programming model to enhance flexibility and mitigate the vertex-parallel performance downgrade by solving the load imbalance issue. With software-hardware (SW-HW) co-design and co-optimizations, GraFlex can derive highly scalable designs with minimal manual effort.

Specifically, this work makes the following contributions:

- GraFlex provides an end-to-end solution for efficient vertex-parallel graph processing systems on FPGAs. It offers a user-friendly design flow based on HLS[1], enabling fast development under the Bulk-Synchronous Parallel (BSP) paradigm.
- GraFlex employs the vertex-parallel scatter-gather programming model to enhance the expressiveness of graph applications. It overcomes the performance downgrade due to degree skew with the proposed coalesced memory access engines, which improve memory bandwidth utilization while exposing user-friendly interfaces at the HLS level.
- GraFlex conducts SW-HW co-design and co-optimization to address the scalability concerns. It configures the compact graph format, partition scheme, and memory channel allocation strategy to establish the SW foundation. For the HW design, GraFlex incorporates a resource-efficient multi-stage butterfly interconnection network to achieve on-device data communication and to facilitate throughput matching.
- We carry out evaluations on various real-world graph applications. The experimental results show that GraFlex achieves up to **2.09×** speedup in average traversal throughput, with a non-negligible reduction in power and resource consumption compared to the state-of-the-art work. A case study on BFS shows a **6.58×** algorithm throughput speedup with proper implementation choices enabled by the adopted scatter-gather model. It also reports an almost linear throughput scale-up versus the PE number and memory channels.

The rest of the paper is organized as follows. Section 2 examines the background of the scatter-gather programming model and the BSP execution paradigm to sketch the problem. Section 3 overviews the GraFlex framework and elaborates on the architecture of the accelerator template. Section 4 elaborates on the design methodology of GraFlex in detail. Section 5 presents comprehensive evaluation results, comparisons with previous works, and analysis. Section 6 includes a review of related works. Lastly, we conclude the paper and list some potential directions for future work in Section 7.

2 BACKGROUND

2.1 The Scatter-Gather Model

The scatter-gather model [31] provides a vertex-parallel abstraction for distributed graph processing. The user-defined scatter function

[1]GraFlex will be open-sourced at https://github.com/eecysu/GraFlex.

Algorithm 1: Scatter-Gather Model with BSP Paradigm

```
Input: Graph $G = (V, E)$ partitioned as $\{G_i = (V_i, E_i)\}$
bspSuperstep = 0;
foreach partition $G_i$ assigned to $CU_i$ do
    Initialize vertex state $S_i$;
    Prepare $activeVertices_i$ from $V_i$;
    while $activeVertices_i \neq \emptyset$ do
        foreach vertex $v_i \in activeVertices_i$ do
            do in parallel
                $outPacket_i \leftarrow$ Scatter($v_i$);
                bspNetwork($outPacket_i$, $inPacket_j$);
                $S_j \leftarrow$ Gather($inPacket_j$, $S_j$);
        Update $activeVertices_i$;
        bspSynchronize();
        $bspSuperstep = bspSuperstep + 1$;
```

sends out messages from active source vertices to their adjacent destination vertices. The user-defined gather function collects the messages and updates the vertex state accordingly. Compared with the classic vertex-centric model, its main difference is that both message sending and receipt are finished within the same iteration step [22]. From the users' perspective, the scatter-gather model inherits the applicability and expressiveness of the vertex-centric model, which facilitates straightforward implementation of graph applications in a vertex-parallel manner.

The main challenge of implementing FPGA-based graph processing systems with the scatter-gather model is that the vertex degree skew in real-world power-law graphs may lead to load imbalance issues in vertex parallel implementations. The GAS model avoids the inefficiency by embracing edge-level parallelism instead, where parallel edge streams are processed in heterogeneous pipelines [22]. However, edge-centric processing sacrifices compact storage and expressive efficiency for applications. GraFlex adheres to the vertex-level parallelism and resolves the problem with coalesced memory access and amortized traffic, which will be elaborated with detail in Sec. 4.4 and Sec. 4.5.

2.2 The BSP Paradigm

The BSP [32] execution paradigm comprises (a) multiple parallel computing units (CUs) for local computation, (b) interconnection networks that allow data communication among CUs, and (c) a central controller that enables global synchronization among multiple CUs. In BSP, a computational task is decomposed into numerous global iterations known as super-steps. Within a super-step, one CU completes its own local computation and communicates with other CUs for data exchange. At the end of each super-step, concurrent CUs are synchronized at the global barrier to ensure that the subsequent super-step is initiated only after the completion of all previous local computations with proper handling of exchanged data. Interested readers can refer to [32] for more details.

GraFlex maps one Scatter PE and a group of Gather PEs to a BSP CU, which can be managed by the central BSP controller. It incorporates the BSP paradigm to adapt to the scatter-gather programming model, paving the way for rapid deployment of parallel graph applications. Algorithm 1 sketches our graph processing problem described in the scatter-gather model with the BSP paradigm.

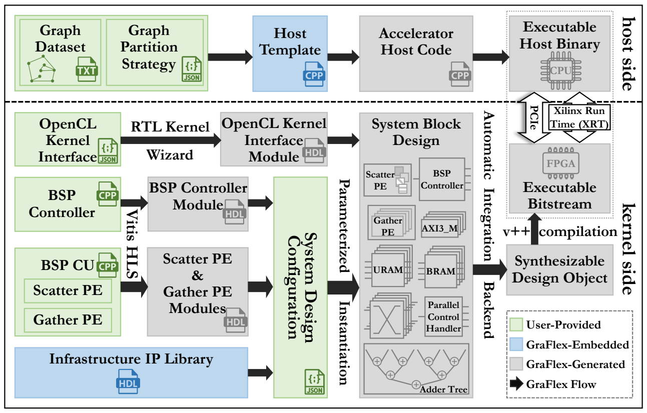

Figure 1: Overview of GraFlex Framework

3 GRAFLEX FRAMEWORK

3.1 GraFlex Overview

From the users' perspective, Fig. 1 shows an overview of the proposed GraFlex framework. To derive a solution for a target graph application, a user first tackle the overall specification on the host side—to analyze the essential characteristics of the application and the input graph dataset, thus formulating a proper graph partition strategy, i.e., src/dst partition and the partition factor. Then, the accelerator host code can be finalized based on the GraFlex-embedded template and compiled into the host binary.

On the kernel side, users need to provide (a) system design configurations, including graph partition schemes, AXI DATA_WIDTH, CSR edge format, and interconnection network settings; (b) an OpenCL kernel interface for argument passing and host-kernel communication via the OpenCL APIs; (c) a source file description of the BSP controller and BSP CU in C/C++. Here, GraFlex decomposes BSP CUs into Scatter PEs and Gather PEs following the scatter-gather graph programming model.

With the initialization of the GraFlex flow, the kernel interface module is generated according to users' definitions. Meanwhile, both the BSP controller and the BSP CU source file are transformed into synthesizable RTL modules within minutes via HLS. According to the system design configurations, a precise number of module instances will be generated and exported to the system block design, including the infrastructure IP instances customized by user-provided parameters.

Next, the automatic integration backend combines module instances by handling connections among them. The system block design also includes an instance of the kernel interface module to be combined into the RTL kernel. Finally, the system-level block design is synthesized and exported as a synthesizable design object.

3.2 Architecture of GraFlex Accelerators

Fig. 2 illustrates the architecture of GraFlex accelerator template corresponding to the system block design shown in Fig. 1. It mainly comprises a central BSP controller, multiple BSP CUs, multi-banked on-chip RAMs, AXI interfacing engines, and customized interconnection network. Each CU is associated with an AXI3 master and an independent memory channel with the AXI interface. In practice, the on-chip memory resources are organized in a group manner.

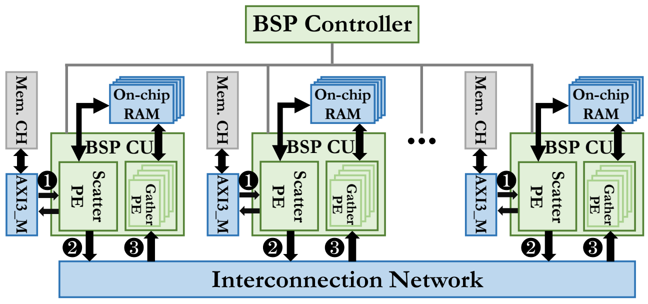

Figure 2: Architecture of GraFlex Accelerator Template

Each group is assigned to a specific CU and the multi-banking design guarantees exclusive access from multiple Gather PEs. The customized interconnection network is integrated for on-device data communication, which will be further elaborated in Section 4.3.

At the start of each BSP super-step, the controller concurrently invokes multiple CUs. The Gather PEs are awakened to continuously detect valid data from the interconnection network output, while the Scatter PEs are activated to initialize the contents of on-chip RAMs and generate memory requests as the application progresses. Data is accessed through the designated memory channel via the AXI interface. Once the requested data is retrieved, it is unpacked, reassembled, and fed into the interconnection network as individual packets.

After some network latency, the packets are routed to their destined Gather PEs. Upon capturing a packet on the output side of the network, the Gather PE performs the user-defined operation and updates the vertex state in the corresponding multi-banked on-chip RAM. As the finalization of a BSP super-step, control signals will be collected from multiple CUs to perform global synchronization between adjacent super-steps at the controller side.

In order to transfer the results from the on-chip RAMs to the main memory, the controller switches the Scatter PEs to write-back mode. The final vertex states are read out consecutively and written back to the corresponding memory channel in a streaming manner. Note that the visits from Scatter PEs and Gather PEs to the on-chip RAMs do not overlap in time, naturally avoiding any potential address conflict or data hazard.

4 GRAFLEX DESIGN METHODOLOGY

4.1 Bottom-up Design Philosophy

GraFlex is designed in a bottom-up manner structured as three layers. The bottom layer is the RTL-based infrastructure IP library, the middle layer is the automatic system integration, and the top layer is the intact C/C++ level HLS design paradigm.

Bottom Layer. The infrastructure IP library aims to provide efficient RTL implementations of building block IPs commonly shared by graph applications. They can be classified into three categories. (a) The global communication IP mainly comprises switch nodes, distributors, and adder nodes to build the customized interconnection network for data communication. (b) The BSP control handler IP is also available. Since the current HLS tools cannot support multi-thread management, the control handler IP enables the invocation of multiple concurrent CUs and provides a global synchronization barrier to achieve BSP abstraction. They are designed according to the Xilinx block-level control protocol *ap_ctrl* [19] to ensure compatibility with HLS kernels. Consequently, they can interact with HLS-generated BSP CU instances to carry out the control mechanism. (c) The memory access IP. The typical memory access pattern for graph applications exhibits random and fragmented reads and sequential writes. To enable fine-grained memory access optimizations, customized AXI interfacing engines are designed.

Middle Layer. The middle layer assembles all exported module instances using the automatic integration backend to finalize the system block design. The integration flow begins by creating the block design with instantiated modules in Vivado, as shown in Fig. 1. The backend Tcl and shell scripts follow the pre-defined GraFlex naming standard and automate the integration flow with negligible time overhead. Once all internal connections among IP instances are settled, the entire design is exported as a synthesizable design object for compiling to the executable FPGA bitstream.

Top Layer. GraFlex provides a comprehensive HLS-based design paradigm. GraFlex provides a set of customized *pragmas* to carry out the BSP execution diagram. Regarding the BSP controller design, a function-associated *pragma* is created to force an empty placeholder function specified by a fixed signature. This preserves the invocation interface for concurrent BSP CUs. To implement the BSP barrier mechanism, several interface *pragmas* are proposed to constrain the relevant signals. The behaviors of these interfacing signals are guaranteed to meet the BSP expectation. They can be identified by the automatic backend in the Middle Layer, thus facilitating the automatic system integration.

4.2 Graph Data Organization

Efficient graph data organization is fundamental to systematic software-hardware co-optimization. It is closely related to the data allocation strategy across multiple memory channels and can directly affect the on-chip data exchange behavior. GraFlex manages the graph data based on the compressed sparse row (CSR) data structure to address several concerns.

Efficient storage for sparse graph. Since most real-world graphs are sparse, general graph data structures like the adjacency matrix or list introduce too much redundancy. Although the coordinate (COO) format is friendly for edge-parallel processing to overcome the degree skew challenge for vertex-parallel models, it requires $2|E|$ storage space, which significantly restricts the scalability of the whole system since $|E|$ is an order of magnitude larger than $|V|$ in a typical real-world graph. On the contrary, the CSR format requires a much smaller $|V|+|E|$ (offset array and edge array combined) storage space to represent graphs compactly.

Pre-processing and partitioning. The graph representation in CSR format can be constructed easily by traversing the edges in ascending order of source-vertex indices, which induces negligible pre-processing overhead. Moreover, the CSR data structure supports convenient partitioning to split the original graph on demand. Fig. 3 shows a tiny example of radix-2 modulo partitioning based on either source or destination node indices. GraFlex provides convenient host APIs for users to perform off-device partition automatically with parameterized configuration. For example, a 4-src, 8-dst partition scheme partitions the CSR according to the

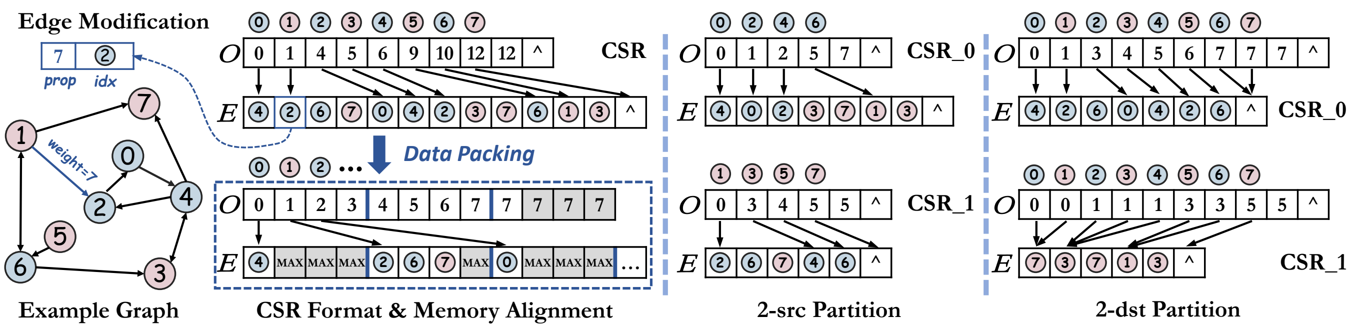

Figure 3: Graph Data Organization: Partition and Data Packing. *O*: Offset Array, *E*: Edge Array.

radix-4 source index modulo and radix-8 destination index modulo. The concrete graph partitioning scheme is in conjunction with the algorithmic design to ensure exclusive access to a memory channel from the corresponding AXI interface. In general, each Scatter PE in a BSP CU exclusively associates with a single memory channel that accommodates a partition of the entire graph.

Application-specific modification. The original CSR format merely captures the adjacency information of a graph. However, many real-world graph applications require associating extra properties with the connection itself. For example, **SSSP** requires extra weights (distance) on the edges, and **SpMV** requires extra vectors from source vertices. To this end, we employ application-specific modifications to the original CSR edge items. Specifically, an edge item is modified to the $(prop, idx)$ format to carry additional property values in the high significance bits, indicated by the dashed arrow in Fig. 3. Thanks to the HLS-based design flow, such bit-level manipulation and compression can be easily achieved through arbitrary precision data types (*ap_int*) from the HLS standard library.

Data packing. In practice, the CSR data structure must be aligned with the AXI DATA_WIDTH (W_{AXI_D}) to maximize the memory bandwidth utilization via vectorized accesses. Both the offset array and the edge array need to be reshaped to pack multiple data items to match W_{AXI_D}. The dashed rectangle in Fig. 3 shows an example where the packing factor is 4 (each packed data item contains four original items). To keep an aligned data layout and differentiate outgoing edges from different source vertices pointed by adjacent items in the CSR offset array, extra padding data is added to the packed CSR format, as shown in the shaded slots.

4.3 Customized Interconnection Network

GraFlex proposes a customized interconnection network to tackle the unsettled data dependencies among multiple BSP CUs. Consider an edge (src, dst) to be processed with a source-index partition scheme. It is fetched at a Scatter PE from the *src* side. Then, an update operation is required at a Gather PE on the *dst* side. However, the on-chip memory cores reserved for caching the updated vertex states are bound with the destination index (*dst*) and do not necessarily belong to the same BSP CU where the edge processing originates. Consequently, run-time data redirection is required to perform update operations in place.

GraFlex opts for the multi-stage butterfly network topology to derive customized interconnection networks for each design. This approach reduces hardware resource utilization and mitigates routing and timing closure difficulties by avoiding long wires. Moreover, the classic structure brings down the routing complexity and can be efficiently implemented with parameterized RTL designs.

Compared to ScalaBFS [26], GraFlex comprises considerable extensions to the multi-stage butterfly network, making it a customizable interconnection in terms of latency, resource usage, and network scale. First, the basic switch units can be configured to either 2×2 nodes or 4×4 nodes, enabling adjustable latency and flexible routing. Second, the memory primitives used to implement the FIFO buffers within each switch node are also configurable (either Distributed RAM or Block RAM) to meet the potential hardware resource constraints and to leave opportunities for physical design optimizations. Last but not least, GraFlex considers the interconnection network with memory interfaces and PE throughput to achieve throughput matching design (further elaborated in Section 4.5) and the balance between performance and resource usage.

Fig. 4 presents two design examples for the 16×16 interconnection network. In (b), the network uses 4×4 switch nodes in two stages to reduce latency. Routing bits are assigned in descending order along the network, and data packets are routed to the output port in ascending order of the remainder. In (c), the network uses 2×2 switch nodes in all four stages, resulting in relatively longer latency. The routing bits are rearranged to modify the routing scheme, and the remainder at each output port changes accordingly. For instance, to route an 8-src partitioned graph data, switches of the first three stages can be used, while the last-stage switches can be used for routing to multi-banked on-chip RAMs (banking factor = 2). In practice, the backend can automatically generate customized interconnection networks using user-defined configurations.

To maximize network bandwidth utilization and ensure worst-case packet traversing latency, GraFlex uses virtual channel (VC) flow control, which associates multiple virtual channels with a single Physical Channel (PC) [10]. Fig. 4(a) illustrates the control with a 2×2 switch node. Each VC has a FIFO buffer that temporarily stores data packets from the input PC. Incoming data packets are redirected to two VCs based on their respective destination PCs (PC_A or PC_B). When multiple VCs require simultaneous access to a single PC, the built-in matrix arbiter performs arbitration to fairly grant an outstanding request by turn. In order to prevent FIFO overflow and packet loss, a backpressure mechanism manages push/pull traffic at both the input and output sides, as illustrated by the dashed arrows. If a potential overflow is detected in the

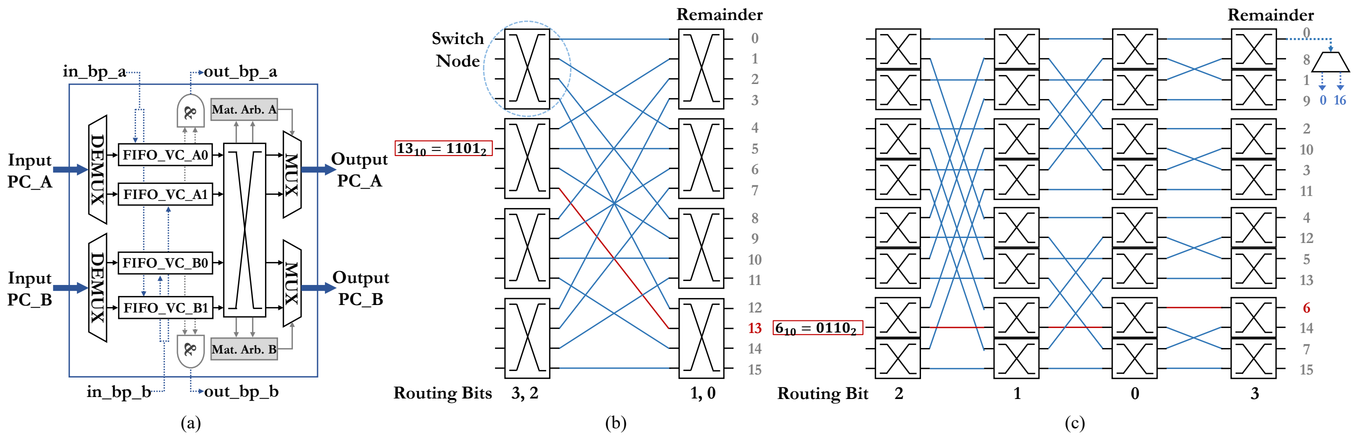

Figure 4: Butterfly Interconnection Network Design

next stage, the input signal *in_bp* is asserted, disabling the FIFO read enable signal until the warning is cleared. Alternatively, if a potential overflow is anticipated in the current stage, the output signal *out_bp* is asserted, informing the previous stage's switch node to stop sending data until the FIFOs have enough space. The Xilinx built-in FIFO programmable indicator (*prog_full*) enables the consideration of potentially unarrived data in flight.

4.4 Memory Access Optimizations

GraFlex proposes a novel coalesced memory access routine at the HLS level to overcome the inefficiency of random memory access in vertex-parallel graph application implementations. The major challenge here is how to merge multiple shattered read requests into coalesced memory accesses since the memory write unveils a consecutive access pattern. Nonetheless, the current HLS tools do not naturally support memory coalescing, as the generation of memory requests and actual memory accesses are inseparable. Furthermore, the lack of low-level control also hinders fine-grained memory manipulation. To address this issue, we propose to decouple the memory request generation and the actual data transfer.

Fig. 5 shows the fundamental procedure for coalesced memory read access. The Scatter PE first collects a sufficient number of memory request tuples in the form of $(arLen, ofstAddr)$, where $arLen$ represents the ARLEN signal from AXI protocol and $ofstAddr$ is the offset address aligned with the AXI DATA_WIDTH (W_{AXI_D}) settings. Afterward, up to 16 tuples are retrieved from the *memReqBuff* FIFO to the outstanding request FIFO (O/S Req. FIFO), and proceed to be filled in the Address Read Channel indicated by the *rd_start* signal. Through this implementation, coalesced accesses can be carried out as multiple outstanding AXI transactions in a

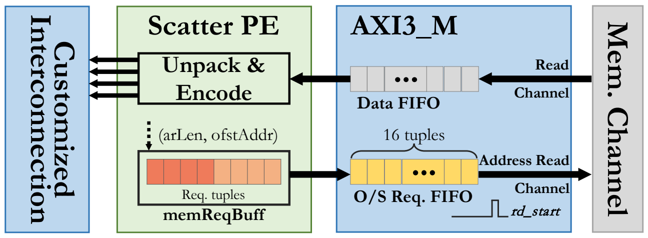

Figure 5: Coalesced Memory Read Access Routine

burst manner. This is in accordance with the AXI3 protocol, which permits up to 16 beats per transaction, thus ensuring compatibility with HBM devices. It is worth noting that the GraFlex memory interface design and optimization are not limited to HBM or any specific FPGA platform.

When the requested data gets ready, it is retrieved from the Read Channel and sent back to the Scatter PE via the stream interface. The data is then unpacked and encoded on demand as individual data packets traveling through the interconnection network.

We further illustrate the HLS implementation of memory coalescing at the Scatter PE side in Listing 1. Each round of coalesced memory access corresponds to one loop iteration in the *scatterMain* function (lines 21-25). The collection of awaiting memory requests starts with the *genMemTuple* function. It iterates over each source vertex within the $vertexBatch$ and forms the request tuples according to the CSR data structure. These tuples are then enqueued in the *memReqBuff* FIFO (lines 18). Next, the *feedNetwork* function is invoked to proceed with the unsettled memory requests. It first prepares the $tupleBatch$ that contains up to 16 unsettled tuples (line 3). Afterward, the request tuples are streamed out to the O/S Req. FIFO residing at the external AXI master (lines 4-5). After that, the *read_start* signal is asserted (line 6) to initiate the address transfer. Finally, the Scatter PE constantly detects incoming data through the *rd_port* interface until the last data is well received (lines 7-11).

Within the *scatterMain* function, the *genMemTuple* function and the *feedNetwork* function can be overlapped using the double buffering technique. It helps hide the latency for request tuple generation to improve bandwidth utilization.

By first collecting the memory request tuples on a vertex-batch basis and grouping them into iterations, the proposed routine is able to decompose heavy requests into multiple ones and accumulate light requests. This way, the bandwidth utilization fluctuation due to degree skew can be mitigated significantly.

4.5 Throughput-Matching Design

Thus far, we have discussed the techniques in GraFlex for addressing flexibility and scalability challenges. With the CSR data structure, GraFlex can partition input graphs and distribute the partitions among multiple memory channels. Moreover, the multi-stage interconnection network can be configured in terms of network scale and latency.

Listing 1: Coalesced Memory Access

```
Function feedNetwork($rdPort, netPort, memReqBuff$):
    for $k$ from 1 to $\lceil memReqBuff.size() / 16 \rceil$ do
        Prepare $tupleBatch$ from $memReqBuff$;
        foreach $reqTuple \in tupleBatch$ do
            $rdPort$.writeReq($reqTuple$);
        $rdPort.rd\_start \leftarrow$ True;
        do
            $tmpData \leftarrow rdPort$.read();
            $outPackets \leftarrow$ UnpackEncode($tmpData$);
            $netPort$.write($outPackets$);
        while $tmpData$.isLast()=False;
        $rdPort.rd\_start \leftarrow$ False;

Function genMemTuple($CSR\_O, CSR\_E, memReqBuff$):
    Prepare $vertexBatch$;
    foreach vertex $v \in vertexBatch$ do
        $arLen \leftarrow$ getArLen($CSR\_O$);
        $ofstAddr \leftarrow$ getOfstAddr($CSR\_O, CSR\_E$);
        $memReqBuff$.enQueue(($arLen, ofstAddr$));

Function scatterMain($CSR\_O, CSR\_E, rdPort, netPort$):
    Initialize $memReqBuff_1$ and $memReqBuff_2$;
    for $n$ from 1 to $N_{batch}$ do
        do in parallel
            feedNetwork($rdPort, netPort, memReqBuff_1$);
            genMemTuple($CSR\_O, CSR\_E, memReqBuff_2$);
        Swap $memReqBuff_1$ and $memReqBuff_2$;
```

GraFlex balances performance and hardware resource utilization under the guidance of throughput-matching design. In order to estimate the theoretical system throughput, three data links need to be considered, as labeled in Fig. 2.

First, the data transfer rate of the main memory (labeled as ①) is determined by the specification of the memory access interface (AXI3_M). If multiple CSR edge items are packed into a single AXI beat, the number of edges that can be accessed per clock cycle can be calculated as $(W_{AXI\_D}/W_e) \times N_C$, where W_e represents the bit width of the CSR edge item and N_C denotes the number of utilized memory channels.

Second, the customized interconnection network's scale (number of I/O ports, denoted as $N_{I/O}$) decides the network's theoretical maximum bandwidth. Therefore, the number of edges that can be transferred through link ② per cycle equals $N_{I/O}$.

Third, at the output side of the network (③), each Gather PE starts to process a data packet upon its arrival. Nonetheless, the processing cannot be finished within one clock cycle, resulting in an initiation interval (II) greater than one. To match the data consuming rate with the network's throughput, a multi-banked design is adopted to amortize the traffic across multiple Gather PEs within one BSP CU (each CU has N_G Gather PEs). This can be implemented by attaching distributors [10] to the last-stage switch nodes as part of the interconnection network. For example, in Fig. 4, the data packets received at the top-most port can be categorized further according to their remainders divided by 32 (either 0 or 16) to fit a Gather PE with $II_{Gather} = 2$. Assuming an ideal traffic pattern with a balanced payload, a maximum number of N_G/II_{Gather} edges can be handled per clock cycle. In real practice, the run-time traffic pattern heavily influences the data consumption rate. However, the law of large numbers [14] guarantees that the approximations are proper with sufficient traffic volumes.

In summary, a throughput-matching design in GraFlex should adhere to the following inequations:

$$\frac{W_{AXI\_D}}{W_e} \times N_C \le N_{I/O} \le \frac{N_G}{II_{Gather}} \qquad (1)$$

Such that the data consuming rate is greater than or equal to the producing rate, avoiding the accumulation of data packets residing in the FIFO buffers halfway.

Table 2: Graph Datasets for Evaluation

Graphs	Type	$\|V\|$	$\|E\|$	Density
mouse-gene (MG)	Undirected	45.1K	14.5M	7.13 E−3
ca-hollywood-2009 (HW)	Undirected	1.1M	56.3M	4.65 E−5
com-orkut (OR)	Undirected	3.1M	117.2M	1.22 E−5
pokec-relationships (PK)	Directed	1.6M	30.6M	1.20 E−5
amazon-2008 (AM)	Directed	735.3K	5.2M	9.62 E−6
wiki-topcats (TC)	Directed	1.8M	28.5M	8.80 E−6
web-google (GG)	Directed	875.7K	5.1M	6.65 E−6
web-hudong (HD)	Directed	2.0M	14.9M	3.73 E−6

5 EVALUATION

5.1 Experimental Setup

System Settings. We evaluate GraFlex with an AMD Xilinx Alveo U280 FPGA board, equipped with 8 GiB of HBM2 memory structured with 32 HBM pseudo channels accessible through the AXI3 interfaces. The development of the whole framework and the compilation of bitstreams are accomplished with AMD Xilinx Vitis & Vivado 2020.2 toolkits on a desktop with an AMD 5900X CPU and 128 GiB DDR4 memory. The running OS is Ubuntu 18.04.5 LTS.

Benchmark Settings. We experiment with a wide range of graph applications on real-world graphs. In selecting benchmarks, one principle considered is the availability of fair comparisons with previous works. Table 2 lists the details of the graph datasets used for evaluation in the descending order of density. All of them are accessible to the public online [25, 29].

Six graph applications of practical importance and popularity are examined: breadth-first search **(BFS)**, single-source shortest path **(SSSP)**, weakly-connected components **(WCC)**, closeness centrality **(CC)**, sparse matrix-vector multiplication **(SpMV)**, and PageRank **(PR)**. The vertex set size is limited to $2M$ for **SpMV** and **PR** due to the on-chip buffering scheme for CSR offset, temporal results, and input vector (if any), resulting in the absence of results for the **OR** dataset. For the remaining problems, the limit is relaxed to $4M$. For the graph feature data associated with edges, we apply 16-bit unsigned integer distances to **SSSP** and 32-bit fixed-point operands to **SpMV** and **PR**. The **PR** application is executed for ten iterations to obtain the average throughput. We omit the article rank **(AR)** application evaluated in ThunderGP [6, 7] since it is derived from **PR** and shares extremely high similarity.

Table 3 presents the GraFlex system configurations for each application, including the graph partition scheme denoted as *(m-src, n-dst)*, the bit width of the CSR edge item ($W_e = W_{prop} + W_{idx}$), the interconnection network configurations ($N_{net}, N_{I/O}, W_{pkt}$) and

Table 3: System Configurations per Application

App.	Partition	W_{AXI_D}	$W_{prop} + W_{idx}$ [1]	$(N_{net}, N_{I/O}, W_{pkt})$ [2]	II_{Gather}
BFS	32-src	128b	0b + 32b	(1, 128, 32b)	2
SSSP	32-src	128b	8b + 24b	(1, 128, 48b)	2
WCC	32-src	128b	0b + 32b	(1, 128, 48b)	2
CC	32-src	128b	0b + 32b	(1, 128, 32b)	2
SpMV	4-src, 8-dst	256b	32b + 32b	(8, 16, 66b)	3
PR	4-src, 8-dst	256b	32b + 32b	(8, 16, 66b)	3

[1] W_{prop}: bit width of the property; W_{idx}: bit width of the index
[2] N_{net}: number of networks, W_{pkt}: bit width of the network packets

Table 4: Traversal Throughput (MTEPS)

🟧 Best Case 🟦 Worst Case

App.	BFS	SSSP	WCC	CC	SpMV	PR
MG	8,731	8,459	11,673	8,477	12,310	11,240
HW	11,013	11,060	13,278	10,681	12,018	9,260
OR	9,876	10,137	11,881	9,574	-	-
PK	7,824	7,916	10,202	7,542	6,266	5,863
AM	5,048	5,126	6,909	4,902	2,535	2,420
TC	3,376	3,410	6,072	3,278	4,519	4,302
GG	4,429	4,511	5,805	4,275	2,117	2,014
HD	5,160	5,265	6,890	5,005	2,658	2,382
Geo. Mean[1]	**6,039**	**6,081**	**8,273**	**5,851**	**4,812**	**4,371**
ThunderGP [6]	4,251	3,689	3,959	3,808	5,308	3,878
Speedup	**1.42×**	**1.65×**	**2.09×**	**1.54×**	**0.91×**	**1.13×**

[1] Excluding OR dataset for fair comparison with ThunderGP [6].

the II for the Gather PEs (II_{Gather}). We utilize all 32 HBM (pseudo) channels for all implementations. For **SpMV** and **PR**, W_{AXI_D} = 256b, and W_e = 64b. In addition, the interconnect is configured as eight separate 16×16 networks, which cooperate with the two-dimensional partition scheme to overcome the routing difficulties. For the remaining applications, W_{AXI_D} = 128b, and W_e = 32b, with a single 128×128 network adopted. Note that the configurations listed here are set for best-effort designs. A systematic design space exploration could be an interesting direction for future work.

Baselines. We mainly compare our results with ThunderGP Alveo U280 implementation [6] since it represents the most comprehensive and state-of-the-art performance for fair comparison. The *traversal throughput* (Traversed Edges Per Second (TEPS)) is used as the consistent metric for performance evaluation.

5.2 Throughput Results

Table 4 shows the traversal throughput of the specified applications and datasets. Thanks to the SW-HW co-optimization and throughput-matching design, we achieve noticeable speedup over ThunderGP in general [6]. Specifically, it is evident from the results that GraFlex performs best on the **WCC** application, achieving a **2.09×** speedup in geometric mean. This is partially because the **WCC** application requires an undirected representation of the input graph, regardless of its original type. Consequently, the random access issue is mitigated. For **BFS**, **SSSP**, and **CC**, GraFlex also achieves significant speedup on average.

As for **SpMV**, GraFlex maintains high throughput for denser datasets (MG and HW) while suffering from more severe performance degradation for sparser graphs compared to other applications. This is mainly because the **SpMV** implementation requires a global reduction of partial sums for each batch of source vertices. Despite using a global double buffering design to overlap the reduction with partial sum calculation, the reduction process still dominates the total elapsed time due to a much lower edge-to-vertex ratio in sparser graphs. Moreover, the limited reduction concurrency exaggerates the load imbalance issue and the tail effect. Compared to ThunderGP [6], our performance is partially recovered in **PR** since we maintain a sole copy of the PageRank vectors throughout the entire execution. After each PageRank iteration, the vectors are updated at the original location, avoiding extra vector movement and the induced overhead.

5.3 Frequency, Resource and Power Statistics

Table 5 presents the resource utilization numbers of GraFlex (with a breakdown for the interconnection network) and the corresponding numbers of ThunderGP [6]. Benefiting from resource-efficient interconnect design and the throughput-matching design methodology, GraFlex can achieve high performance while maintaining a medium level of resource usage. Compared to ThunderGP [6], GraFlex achieves ~27% CLB saving on average. This is primarily due to the highly efficient interconnection network design in GraFlex compared to the data shuffling logic of ThunderGP [6]. Since CLB tiles dominate the area utilization, GraFlex has the potential to accommodate a larger design more compactly.

Table 6 shows the kernel frequency and power statistics of GraFlex compared with ThunderGP. GraFlex achieves 160~190 MHz among all six applications implemented. Although timing closure techniques such as inserting extra flip-flops on latency-insensitive paths have been employed, further frequency improvement requires non-trivial systematic physical design optimizations, which are left for future work. In terms of power efficiency, GraFlex achieves a ~12% average reduction. Although the credit can be attributed to the lower clock rate, this frequency difference does not prevent us from achieving superior throughput.

5.4 Analysis and Discussion

To study the throughput fluctuation across different graph datasets, we create an analytical bandwidth efficiency model based on the coalesced memory access routine discussed in Section 4.4. Fig. 6 shows a basic timing diagram illustrating our model. Assuming that W_e = 32b, W_{AXI_D} = 128b and the size of *vertexBatch* is 64. The idle latency between address and data transfer is set to N_{Idle} = 20

Table 5: Resource Utilization

App.	BRAM		URAM		CLB		DSP	
	G	T	G	T	G	T	G	T
BFS	26.79% (6.34%)	61%	66.67%	60%	55.62% (12.96%)	94%	0.04%	0.04%
SSSP	39.48% (12.68%)	65%	66.67%	53%	57.68% (17.95%)	88%	0.04%	0.04%
WCC	33.13% (12.68%)	62%	93.33%	62%	63.31% (17.85%)	90%	0.04%	0.04%
CC	23.61% (6.34%)	62%	66.67%	60%	57.19% (13.06%)	91%	0.04%	0.04%
SpMV	71.23% (0%)	66%	43.33%	68%	77.59% (14.12%)	98%	22.74%	2.44%
PR	71.23% (0%)	63%	43.33%	60%	76.75% (14.64%)	92%	22.74%	0.4%

Note : **G**: GraFlex, with interconnect resource breakdown in brackets (w.r.t hardware budget, if any); **T**: ThunderGP [6].

Table 6: Kernel Frequency and Power Consumption

	BFS		SSSP		WCC		CC		SpMV		PR	
	G	T	G	T	G	T	G	T	G	T	G	T
Freq. (MHz)	175	250	180	243	160	258	170	251	190	250	180	267
Power (W)	42	52	44	52	43	49	42	48	51	58	50	50

Note : **G**: GraFlex (Ours); **T**: ThunderGP [6].

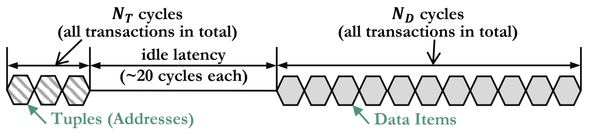

Figure 6: Timing Diagram of Coalesced Memory Access Model

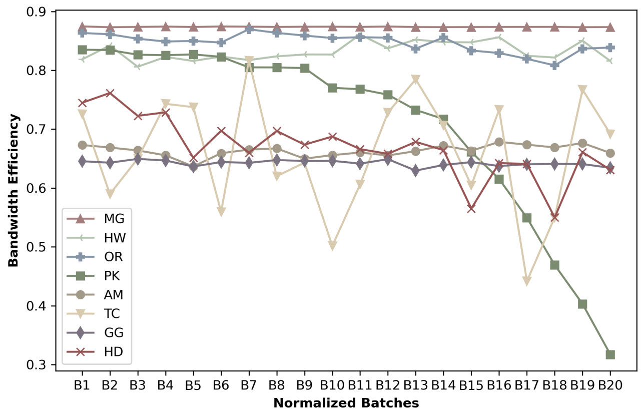

Figure 7: Theoretical bandwidth efficiency calculated by batches sequentially. The batch number of different datasets is unified to 20 with the nearest interpolation to show the comparison.

cycles according to the results from Shuhai [33] and our kernel frequency. Since each AXI3 transaction holds up to 16 beats, and a coalesced memory access handles up to 16 memory requests, the theoretical bandwidth efficiency (BW_{opt}) for each vertex batch can be defined as:

$$N_D = \sum_{v=0}^{64-1} \lceil \frac{\text{out\_degree}(v)}{W_{AXI\_D}/W_e} \rceil, \; N_T = \lceil \frac{N_D}{16} \rceil$$
$$BW_{opt} = \frac{N_D}{N_{Idle} \times \lceil \frac{N_T}{16} \rceil + N_T + N_D} \tag{2}$$

If no coalesced access is performed, each vertex in the batch introduces N_{Idle} extra cycles to establish a transfer handshake. Thus, the bandwidth efficiency (BW_{ori}) is expressed as:

$$BW_{ori} = \frac{N_D}{N_{Idle} \times 64 + N_T + N_D} \tag{3}$$

Based on this model, we profile the datasets to obtain their bandwidth efficiency for one-pass source vertex ascending traversal from 32 parallel Scatter PEs. We set a 32-src partition scheme with 32 memory channels. Due to the different dataset sizes, the number of batches ranges from 22 to 1501. To visually compare the bandwidth efficiency, we take the average bandwidth among all 32 partitions per batch for each dataset. Then, we apply the nearest interpolation and normalize the results to 20 data points, as Fig. 7 shows.

The mean and variance of memory bandwidth utilization are shown in Table 7. Several interesting findings arise when jointly analyzing it with the throughput (Table 4) and the graph density numbers (Table 2).

- Coalesced memory accesses bring 1.04×~9.69×improvement of average bandwidth utilization. Compared to denser graphs, the optimization is more effective for sparser graphs, contributing to the overall performance of GraFlex.

Table 7: Mean and Variance of Memory Bandwidth Utilization

Graph	MG	HW	OR	PK	AM	TC	GG	HD
$\overline{BW_{ori}}$	83.67%	40.19%	43.26%	16.77%	7.85%	12.89%	6.63%	8.22%
$\overline{BW_{opt}}$	87.38%	83.94%	84.57%	70.50%	66.51%	66.78%	64.27%	66.53%
$\overline{BW_{opt}}/\overline{BW_{ori}}$	1.04×	2.09×	1.95×	4.20×	8.47×	5.18×	9.69×	8.09×
$\sigma^2_{BW_{opt}}$	1.82E−7	3.36E−4	2.28E−4	2.43E−2	1.04E−4	1.27E−2	3.25E−5	4.64E−3

- Generally, the traversal throughput goes down with graph density within the same application, which entails that the upstream data transfer rate in link ① gets lower with sparser graphs and becomes the performance bottleneck.
- Despite sharing similar density and average bandwidth efficiency, **AM**, **TC**, and **GG** datasets display considerable performance differences. **AM** and **GG** show noticeably higher throughput than **TC** when executing **BFS**, **SSSP**, and **CC** due to the much lower variance. Please note that the dataset profiling results do not apply to **WCC** as it takes an undirected graph as input.

The above results and findings are less meaningful for **SpMV** and **PR** since these two implementations necessitate global synchronization and reduction per processing tile.

5.5 Case Study: Implementation Choices of BFS

The GAS model bypasses the load imbalance issue induced by vertex degree skew with edge-level parallelism [22]. However, our observation in practice is that the preference for edge parallelism might restrict the implementation choices of certain graph applications. For example, the BFS implemented with Dijkstra's algorithm is much more work-efficient than using the Bellman-Ford algorithm since it avoids redundant traversals. Nevertheless, the edge-parallel GAS model cannot implement the Dijkstra's version efficiently due to the vertex-centric nature of this implementation [8, 22]. With GraFlex, we are capable of implementing both of them efficiently.

In real-world scenarios, pursuing pure high *traversal throughput* can be less attractive than it appears as it merely reports the raw throughput of the system, regardless of the exact algorithm performance. To evaluate the actual algorithmic efficiency, we adopt the *algorithm throughput* [8] (the number of undirected edges in the connected component divided by the execution time) metric instead and compare the two BFS algorithm variants implemented by GraFlex in Table 8.

Compared to the Bellman-Ford version, the Dijkstra's achieves **6.58**× speedup in *algorithm throughput* on geometric average even at a lower kernel frequency. Since for the Dijkstra's, the *algorithm throughput* equals to the *traversal throughput*, we also find that its vertex-centric nature does not prevent GraFlex from achieving a relatively high *traversal throughput.*

5.6 Scalability Evaluation

In order to evaluate GraFlex's scalability, we explore the throughput scaling of the generated designs with varying numbers of PEs. Fig. 8 shows the experimental results with BFS as an example.

Table 8: Algorithm Throughput of BFS with Two Implementations

Throughput (MTEPS)	MG	HW	OR	PK	AM	TC	GG	HD
Bellman-Ford[1] (T_B)	1,449	1,375	1,235	1,565	240	25	261	369
Dijkstra's[2] (T_D)	7,689	8,180	7,624	5,528	1,377	1,166	1,349	1,369
Speedup (T_D/T_B)	5.31	5.95	6.17	3.53	5.74	46.64	5.17	3.71

[1]Kernel frequency = 175 MHz. [2]Kernel frequency = 160 MHz.

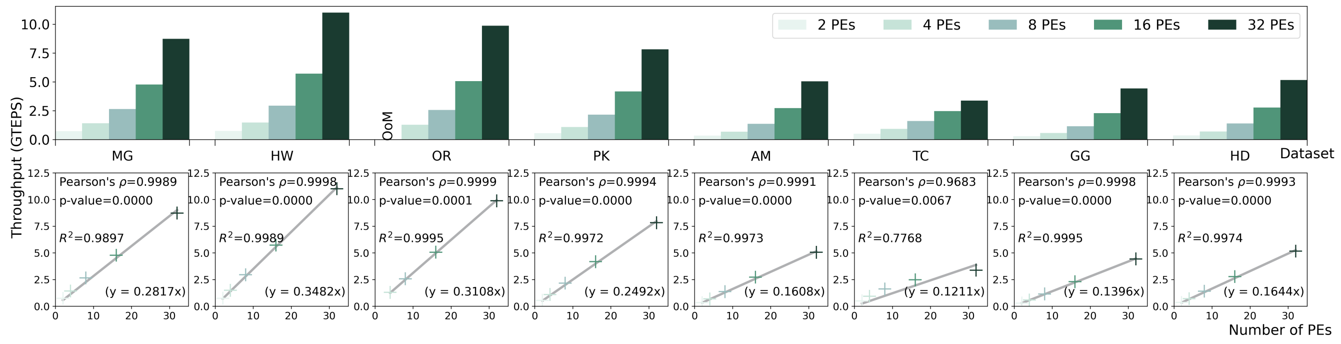

Figure 8: Throughput of the BFS Application on Different Datasets, and Almost-Linearity of Throughput v.s. the Number of PEs

Following the proposed throughput-matching design methodology, each PE corresponds with one memory channel. We adjust the graph partition scheme, network scale, and the number of adopted HBM channels accordingly. To ensure fairness in comparison, the AXI DATA_WIDTH remains at 128 bits for all configurations, and the operating frequency is uniformly set at 175 MHz. Please note that due to the memory capacity overflow, we cannot obtain results for 2 PEs evaluated on the **OR** dataset.

The above results show that seven of the eight dataset groups demonstrate a strong linear correlation between system throughput and PE number. Nevertheless, according to the Pearson correlation coefficient, the **TC** dataset group exhibits the weakest correlation, which is in line with findings from Section 5.4 that attribute the low performance to the skewed topology of the **TC** dataset. Overall, the strong linearity also confirms the efficacy of the proposed design methodology outlined in Section 4.5, proving that run-time traffic can be evenly distributed regardless of the graph partition scheme or the network scale.

6 RELATED WORKS

GraphLily [18] proposes an FPGA-based hardware overlay of GraphBLAS [23]. It expresses several typical graph applications in linear algebra. While the Alveo U280 platform with HBM is deployed to overcome memory bandwidth limitations, there remains a notable performance gap due to architectural inefficiencies. Moreover, the linear algebra model restricts efficient implementations of some graph applications [22]. Additionally, GraphLily requires preserved HBM channels for the exclusive use of data movement, and the arbitrated crossbar logic is less adaptable for scalable designs.

GraphScale [12] proposes a scalable graph processing framework with a two-level vertex-labeled crossbar. It adopts a vertex-centric iteration scheme and pull-based data flow to perform graph applications with Graph Cores. Despite the scalability advantage, the HDL-based development flow requires higher efforts for accommodating new applications.

ThunderGP [7] is the first HLS-based high-performance graph processing framework targeting modern multi-die FPGAs. It adopts the Gather-Apply-Scatter (GAS) model to map the three processing phases into heterogeneous pipelined PEs and exploits a caching mechanism to enhance data locality. The original implementation with multiple DDR banks suffers from the memory bottleneck [7], which is mitigated by migrating to the Alveo U280 HBM-enabled FPGA in their follow-up work [6]. Nonetheless, the GAS model adopted is restricted in graph problem expressiveness [22]. Besides, their design methodology requires significant effort to derive scalable designs. Specifically, directly fitting an appropriate number of kernels to saturate the HBM memory bandwidth is hindered by the resource-consuming shuffling logic [6].

ACTS [21] proposes to replace offline graph slicing with its run-time partition phase to enhance the scalability. It also presents a novel edge-packing format (ACTPACK) to overcome the concurrency limitations during memory access. However, it heavily relies on scratchpad memory (BRAMs and URAMs) resources as property buffers, while the target Alveo U280 platform can only accommodate up to 24 PEs. Additionally, the one-fit-all design style cannot be optimized per application.

ReGraph [5] proposes the dense-sparse heterogeneous pipelines for resource efficiency and high performance. Nevertheless, it requires non-negligible pre-processing of the input graph compared to GraFlex, and the limitations brought by the GAS model remain.

7 CONCLUSION AND FUTURE WORK

This work presents GraFlex, a flexible scatter-gather graph processing framework on FPGAs with scalable interconnect for memory. GraFlex adopts the BSP paradigm to achieve global control and synchronization. The HLS-based design flow enables rapid deployment of high-performance graph processing systems. Benefiting from the software-hardware co-optimizations, GraFlex achieves up to **2.09**× speedup in average throughput over the existing state-of-the-art solution, ThunderGP [6]. It also brings non-negligible reduction in power and resource consumption. A case study on BFS using GraFlex reveals that the enabled design flexibility can bring an average algorithm throughput speedup of **6.58**×. The study also reports an almost linear throughput scaling versus the PE number and memory channels. In future work, physical design optimizations can be further applied to boost the kernel frequency and the overall performance. A systematic study on design space exploration could also help locate the optimal design point per target problem.

ACKNOWLEDGMENT

We thank Kang Zhao from Beijing University of Posts and Telecommunications for his advice on HLS-based design and the anonymous reviewers for their insightful comments and remarks. This research was partially supported by AI Chip Center for Emerging Smart Systems (ACCESS), Hong Kong SAR and the RGC GRF grant 16213422.

REFERENCES

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

[2] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. 2019. Graph processing on fpgas: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697* (2019).

[3] Brahim Betkaoui, Yu Wang, David B Thomas, and Wayne Luk. 2012. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 8–15.

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

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

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

[7] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.

[8] Yuze Chi, Licheng Guo, and Jason Cong. 2022. Accelerating SSSP for power-law graphs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 190–200.

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

[10] William James Dally and Brian Patrick Towles. 2004. *Principles and practices of interconnection networks*. Elsevier.

[11] Jonas Dann, Daniel Ritter, and Holger Fröning. 2022. GraphScale: Scalable bandwidth-efficient graph processing on FPGAs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 24–32.

[12] Jonas Dann, Daniel Ritter, and Holger Fröning. 2023. GraphScale: Scalable Processing on FPGAs for HBM and Large Graphs. *ACM Transactions on Reconfigurable Technology and Systems* (2023).

[13] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. 2014. Work-efficient parallel GPU methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 349–359.

[14] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. 2005. *A Modern Introduction to Probability and Statistics: Understanding why and how*. Vol. 488. Springer.

[15] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021), 1–70.

[16] Eric Finnerty, Zachary Sherer, Hang Liu, and Yan Luo. 2019. Dr. BFS: Data centric breadth-first search on FPGAs. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[17] Paul Grigoraş, Pavel Burovskiy, Wayne Luk, and Spencer Sherwin. 2016. Optimising Sparse Matrix Vector multiplication for large scale FEM problems on FPGA. In *2016 26th international conference on field programmable logic and applications (FPL)*. IEEE, 1–9.

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

[19] Xilinx Inc. 2020. Vitis High-Level Synthesis User Guide. https://docs.xilinx.com/r/2020.2-English/ug1399-vitis-hls/

[20] Abhishek Kumar Jain, Hossein Omidian, Henri Fraisse, Mansimran Benipal, Lisa Liu, and Dinesh Gaitonde. 2020. A domain-specific architecture for accelerating sparse matrix vector multiplication on fpgas. In *2020 30th International conference on field-programmable logic and applications (FPL)*. IEEE, 127–132.

[21] Wole Jaiyeoba, Nima Elyasi, Changho Choi, and Kevin Skadron. 2023. ACTS: A Near-Memory FPGA Graph Processing Framework. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 79–89.

[22] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2017. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering* 30, 2 (2017), 305–324.

[23] 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 *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[24] Guoqing Lei, Yong Dou, Rongchun Li, and Fei Xia. 2015. An FPGA implementation for solving the large single-source-shortest-path problem. *IEEE Transactions on Circuits and Systems II: Express Briefs* 63, 5 (2015), 473–477.

[25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[26] Kexin Li, Chenhao Liu, Zhiyuan Shao, Zeke Wang, Minkang Wu, Jiajie Chen, Xiaofei Liao, and Hai Jin. 2021. ScalaBFS: A Scalable BFS Accelerator on HBM-Enhanced FPGAs. *arXiv preprint arXiv:2105.11754* (2021).

[27] Hang Liu and H Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[28] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

[29] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 29.

[30] Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai Jin. 2019. Improving performance of graph processing on FPGA-DRAM platform by two-level vertex caching. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 320–329.

[31] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From" think like a vertex" to" think like a graph". *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[32] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[33] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. 2020. Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 111–119.

[34] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. 2015. Accelerating large-scale single-source shortest path on FPGA. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 129–136.

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