Rethinking Design Paradigm of Graph Processing System with a CXL-like Memory Semantic Fabric

Xu Zhang*†, Yisong Chang*†‡, Tianyue Lu*†, Ke Zhang*†, Mingyu Chen*†‡
*State Key Laboratory of Processor, Institute of Computing Technology, CAS, Beijing, China
Email: {zhangxu19s, changyisong, lutianyue, zhangke, cmy}@ict.ac.cn
†University of Chinese Academy of Sciences (UCAS), Beijing, China
‡Zhongguancun Laboratory, Beijing, China

***Abstract*—With the evolution of network fabrics, message-passing clusters have been promising solutions for large-scale graph processing. Alternatively, the shared-memory model is also introduced to avoid redundant copies and extra storage space of graph data. Compared to conventional network fabrics, with the capability of fine-grained, byte-addressable remote memory access, emerging memory semantic interconnects and fabrics, e.g., Intel's Compute Express Link (CXL), are intuitively more appropriate for adoption in shared-memory clusters. However, due to the latency gap between local and remote memory, it is still challenging to take advantage of the shared-memory graph processing with memory semantic fabrics.**

To tackle this problem, in this paper, we first investigate memory access characterizations of graph vertex propagation based on the shared-memory model. Then we propose GraCXL, a series of design paradigms to address high-frequency and long-latency of remote memory access potentially incurred in CXL-based clusters. For system adaptiveness, we elaborate GraCXL towards the general-purpose CPU cluster and the domain-specific FPGA accelerator array, respectively. We design a custom fabric with the CXL.mem protocol and leverage a couple of ARM SoC-equipped FPGAs to build an evaluation prototype in the absence of commodity CXL hardware and platforms. Experimental results show that the proposed GraCXL CPU and FPGA clusters achieve 1.33x-8.92x and 2.48x-5.01x performance improvement, respectively.

***Index Terms*—graph processing, memory semantic fabric**

I. Introduction

Motivation. As natural representations and abstractions of relationships between entities in real-world datasets, graphs have been widely adopted in many crucial application domains (e.g., social network [1], disease outbreak paths [2], machine learning [3], etc.). Thus, graph processing algorithms, e.g., breadth-first search (BFS), PageRank, single source shortest path, etc., have become an important kind of computing services in datacenters. With the exponential growth of data from real-world applications, it is strongly required to create a performant large-scale graph processing system.

This trend has inspired a large body of research in building clusters via conventional network fabrics (i.e., interface cards and switches supporting TCP/IP Ethernet or RDMA) [4]–[10]. With minimal setup overhead and simple API calls, the message-passing mechanism is widely leveraged in such scale-out clusters to interchange vertices' attribute data and to distribute operations among different computing nodes. Despite avoidance of cluster-scale cache coherency and atomicity, the message-passing clusters still suffer from huge deluge of inter-node messages along with redundant graph data copies that occupy a significant portion of network bandwidth and require extra storage space.

Limitations in state-of-the-art. To overcome the expense of inter-node communications, many graph frameworks have focused on scale-up shared-memory platforms [11]–[13]. Although the DRAM capacity is increasing, it is still difficult for a single machine to keep pace with the growing scale of graph data. Consequently, a couple of graph systems aim to extend the shared-memory model to a scale-out manner [14]–[16]. However, with commodity network fabrics, such shared-memory clusters still exhibit some drawbacks:

- Explicit initiation. With the existing DMA-based approach, graph engines have to explicitly initiate inter-node data movement with specific software threads or additional hardware units, blocking vertex or edge processing stages.

- Cost ineffectiveness. The coarse granularity in network fabrics always causes a large chunk of interchanged vertices data within a long period. Additionally, as some vertices may be inactive or useless, there exists a large portion of redundant data movement.

Based on these imposed communication overhead, *we argue that clusters based on existing network fabrics are inappropriate to exploit key benefits of the shared-memory model*.

Potential solution and challenges. Fortunately, with its fine-granularity, byte-addressable load-store (ld-st) semantics and native atomic support, the memory semantic interconnect, e.g., Intel's Compute Express Link (CXL) [17], is becoming a potential approach to enabling memory expansion for distributed computing platforms. Initially based on PCI-Express (PCIe), the scope of CXL is confined as an intra-chassis interconnect due to its limited cable length. But after recent absorption of Gen-Z standard [18], CXL is expected to harness Ethernet, breaking up the chassis boundary and extending its scope to a cluster-scale fabric. The same story has taken place in evolution of the widely leveraged NVMe standard [19] and the NVMe-over-Fabrics (NVMe-oF) protocol [20], which is originally designed for local access to non-volatile flash memory over PCIe and then extended to spread NVMe benefits over network fabrics, respectively. Therefore, a conceptual *memory semantic fabrics*, noted as CXL-over-Fabrics (CXL-oF), would raise a new opportunity to build efficient shared-memory graph

processing clusters, in which one node would directly initiate byte-addressable remote memory access (RMA) requests to other nodes and conduct in-place computing after receiving read-back data.

However, in order to implement a shared-memory graph processing cluster based on CXL-oF, we still meet four main system-level challenges:

- *Significant latency gap.* Compared with local DRAM access, it is intuitive to notice that fine-grained RMA over CXL-oF would introduce severe latency overhead. We need to describe impacts of such latency gap and explore proper computing paradigms to eliminate performance degradation of graph algorithms.

- *Unavailable hardware and platforms.* Even though CXL-oF exhibits a great potential, it has not yet been made for production, not to mention building a cluster. We need to investigate how to emulate the behavior of CXL-oF and how to optimize CXL-oF based clusters via hardware-software co-design methods in the absence of real CXL-oF hardware.

- *Divergence of graph engines.* To guarantee system performance, besides conventional general-purpose CPUs, a large number of heterogeneous computing resources (i.e., GPUs [5], [21], domain-specific accelerators based on FPGAs [8], [10], [14], [16] and ASICs [22], [23]) are involved in building graph processing clusters. We need to examine the adaptiveness and efficiency of CXL-oF to support various graph processing engines.

- *Semantic limitation in atomic operations.* In current CXL protocol, atomic write operations to one remote memory region would be allowed to initiate only when the exclusive ownership to that region is obtained and data blocks in that region are explicitly loaded from the remote node. Such semantic would lead to tremendous inter-cluster data movement in distributed graph processing for simultaneous attribute updates due to limited spatial locality of vertices and edges.

Our work. To tackle all challenges mentioned above, this paper describes GraCXL, a series of design paradigms and implementations to build a *feasible and adaptive* shared-memory cluster with the memory semantic fabric for competitive graph processing performance. It relies on our extended CXL standard over network fabrics (CXL-oF), which enables inter-node byte-addressable RMA via ld-st semantics with moderate latencies.

First, we qualitatively study RMA characterizations in both pull and push modes of vertex propagation via the shared-memory model. Based on our observations, we propose system-level approaches to 1) reducing access frequency and 2) overlapping RMA and computation, respectively, to overwhelm RMA latency overhead in CXL-oF clusters (§III).

Second, we design a custom hardware stack of CXL-oF based on the standard `CXL.mem` protocol with a lightweight extension of custom inter-node atomic operations. Such stack supports delivery of CXL commands over commodity network infrastructures and easy accommodation to existing graph engines and memory controllers. We synthesize and implement the custom CXL-oF on FPGA to provide a fundamental component for approximated real-world evaluations of GraCXL (§IV).

Last but not least, to better understand how GraCXL could be effective to various types of graph engines, we adopt GraCXL paradigms and evaluate clusters' performance on both a representative von-Neumann architecture, i.e., general-purpose CPUs (GraCXL-CPU), and a domain-specific architecture, i.e., custom accelerators on FPGAs (GraCXL-FPGA), respectively. (§V).

System delivery. To prove the key concepts of GraCXL even in the absence of available commodity CXL-oF hardware, we build a prototype with ARM SoC-equipped FPGAs. By deploying the custom CXL-oF protocol stack, FPGAs are interconnected via a standard Ethernet switch. We select BFS and PageRank as our workloads on three real-world graphs as well as two scale-free synthetic graphs for both GraCXL-CPU and GraCXL-FPGA evaluations. Especially, we use the hard-core ARM SoCs in each FPGA node instead of commercial high-performance server-class CPUs to build a GraCXL-CPU cluster. We also extend Ligra [11], an open-source software framework for shared-memory graph processing, to a cluster scale for effectiveness evaluations of GraCXL-CPU (§VI).

In summary, our main contributions include:

- A comprehensive design paradigm, GraCXL, for CXL-based share-memory graph processing cluster.

- A custom design and implementation of a CXL-over-Fabric (CXL-oF) hardware protocol stack.

- Implementations and prototype of GraCXL towards the CPU cluster as well as the FPGA accelerator array.

- GraCXL-CPU cluster presents 1.33x-8.92x performance improvement compared to a simple distributed version of Ligra framework.

- GraCXL-FPGA array reaches up to 9.01 GTEPS (billions of traversed edges per second), which outperforms the state-of-the-art shared-memory and message-passing FPGA clusters by 2.48x-5.01x and 1.13x-3.73x, respectively.

II. Background

A. Basis of Graph Processing

Terminology. In one graph G, entities are denoted by a set of *vertices* V and relationships between entities are represented by a set of directed or undirected *edges* E. Each edge connects two individual vertices with an optional weight value. In this paper, we assume that each edge is directed (Figure 1(a)), and an undirected graph can be realized by adding an opposing edge to each directed edge. For one vertex v in a directed graph, *outgoing edges* of v are sourcing from v, while *incoming edges* of v are destined to v.

Synchronous execution model. Most graph processing algorithms are executed in the form of iterations, in each of which attribute values of each vertex (vertex-centric) or edge (edge-centric) are updated via a huge amount of vertex and edge traversals. In this paper, we choose to express an iteration using vertex-centric execution model. Gather-Apply-Scatter (GAS) model [4] is an easy-to-understand computing

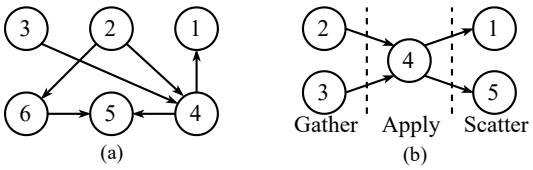

Fig. 1. We present an illustrative example graph in (a) and the three-phase GAS processing model (taking v_4 as an example) in (b) (§II-A).

paradigm in which such iterative kernel algorithm is specified as a *superstep* with three consecutive phases (Figure 1(b)). For a directed Graph *G = <V, E>*, in superstep *N*, new attribute values of the active vertex v_4 is first calculated in the *Gather* phase according to v_4's source vertices' (v_2, v_3) values updated in superstep *N-1*. Then attribute of v_4 is updated by such newly calculated value in the *Apply* phase. Finally, in the *Scatter* phase, v_4's updated value and active traversing status are propagated to all destination vertices (v_1, v_5) along v_4's outgoing edges for computing in superstep *N+1*. Thanks to inherent weak dependencies between vertices, the kernel algorithm is run concurrently for each active vertex in each superstep. In order to guarantee correctness of parallel algorithm execution, supersteps are serialized by strict global barrier synchronization, and any messages propagated to neighboring vertices in superstep *N* will only be leveraged in superstep *N+1*.

Propagation of vertex messages. In the GAS model, attribute updates are propagated between vertices along edges. Conventionally, a vertex is able to *pull* values updated in the previous superstep from active source vertices of all incoming edges in the Gather phase. However, such *pull* manner would result in severe unnecessary edge traversals to find active source vertices when only a subset of vertices needs to update its neighbors. To reduce edge traversing burden, each vertex is also allowed to *push* the updated value in the Scatter phase of current superstep along all outgoing edges. When multiple vertices *push* to the same destination vertex, data races would be incurred by simultaneous updates and could only be resolved by explicit atomic operations.

B. Limitations in Shared-Memory Clusters

With a unified abstraction to hide difference between local and remote memory, the shared-memory model is widely adopted in various kinds of graph processing clusters (e.g., SHMEMGraph [15] for general-purpose CPU cluster, and FPGP [16] as well as ForeGraph [14] for custom FPGA cluster). However, based on conventional network fabrics (i.e., TCP/IP network or RDMA), existing graph engines or software frameworks in the shared-memory clusters need to maintain extra local buffers for remote memory and copy data explicitly between iterations. For example, SHMEMGraph [15] builds a partitioned global address space with one-side RDMA, which provides a global address abstraction with explicit copy of vertices in a message-passing like manner. Similarly, FPGP [16] and ForeGraph [14] maintains a global shared vertex memory with per-node vertex caches and a distributed shared memory, respectively. In these clusters, a large chunks of vertices are read iteratively from remote memory without consideration of status of loaded vertices in the next iteration, leading to a large amount of redundant data movements.

C. Emerging Memory Semantic Interconnect

CXL [17], as well as its absorbed Gen-Z [18] and Open Coherent Accelerator Processor Interface (OpenCAPI) [24], offer high-performance connectivity among multiple processors, hardware accelerators, and memory devices in one chassis. Based on the CXL switch (similar to PCIe switch) introduced in the latest version of protocol, native load-store (ld-st) semantics would be expanded to a rack scale. With the further extension over network fabrics (i.e., CXL-oF mentioned in §I), such interconnect protocols are expected to help shared-memory distributed systems avoid explicit data copying. This is crucial for graph processing clusters as status of some vertices could be inactive in some iterations. In this situation, fabric bandwidth can be significantly saved with fine-grained ld-st semantics to avoid unnecessary data movements.

However, it is believed that latency of RMA via CXL is commonly among 300-500 ns [25]. Considering the inherent latency of Ethernet fabrics, the RMA latency of CXL-oF would be around 10x compared to the latency of local memory access. As a huge number of RMA would exist, how to optimize execution models of distributed graph processing to diminish the latency impact is a fundamental problem for performance improvement with the CXL-oF-based shared-memory clusters.

III. Design Paradigms in GraCXL

We first use PageRank as an example to qualitatively analyze the RMA characterizations as scaling out existing shared-memory single machine systems (§III-A). Based on our findings, we propose two optimization methods to reduce the frequency of RMA (§III-B) and hide RMA latency (§III-C), respectively.

A. Remote Memory Access Characterizations

In order to accurately describe behaviors of remote memory access in the CXL-oF based shared-memory clusters, we investigate both pull and push modes of vertex messages' propagation.

Pull mode. As shown in Figure 2 (a), the Gather stage in one node iteratively traverse local vertices v_i and their incoming edges. For each edge, the graph engine loads the source vertex v_j's value $attr\_current[v_j]$ updated in the previous superstep from remote or local memory. These loaded values are consumed to update destination vertices. This method introduces random access to remote memory and wastes inter-node communication bandwidth as the frequent thrashing incurred by limited capacity of data caches in each node.

Push mode. As shown in Figure 3 (a), in order to proactively store updated vertex attribute values among nodes, one node loads remote vertices attributes $attr\_next[v_j]$ from other

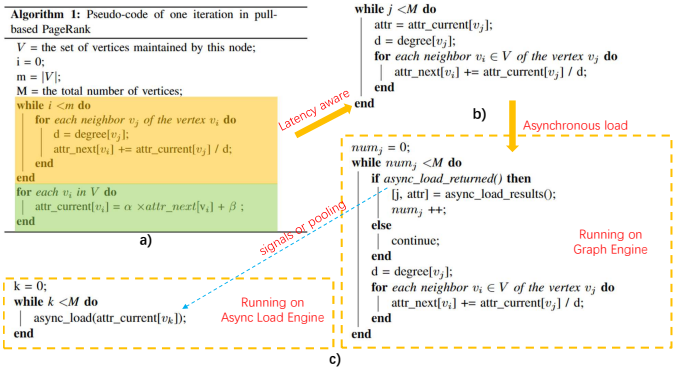

Fig. 2. Pseudocode of pull-based PageRank by scaling-out existing single machine shared-memory system Ligra [11] in (a). Yellow and green block represents Gather, Apply kernels respectively. The latency-aware pull method and asynchronous load are showed in (b) and (c) (§III).

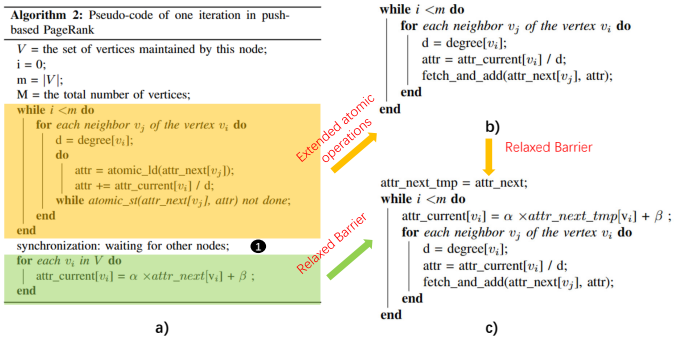

Fig. 3. Pseudocode of push-based PageRank by scaling-out existing single machine shared-memory system Ligra [11] in (a). Yellow and green block represent Gather, Apply kernels respectively. The fetch-and-ops atomic operations and relaxed barrier are showed in (b) and (c) (§III).

nodes when traversing outgoing edges of local vertices v_i, and atomically stores updated value $attr$ to remote vertices attributes. However, when multiple nodes simultaneously try to store to the same vertex in an atom manner, only one atomic store would be exclusively guaranteed at a time, leading to other nodes re-reading new value and retrying atomic stores. Such exclusive manner would significantly damage performance if competing stores are severe among different nodes.

In summary, both pull and push modes for vertex messages' propagation suffer from the large number of random RMAs. With such characterization, graph engines would be frequently stalled to wait for return load values from remote nodes.

B. Paradigm 1: Frequency Reduction

An intuitive method to reduce overhead of remote memory access is to eliminate the frequency to initiate remote vertex data traversals. In order to achieve this goal, adjustments are required in execution models.

Pull mode. As the cache-miss penalty of data located in remote memory is more expensive than that of data in the local memory, we introduce a latency-aware pull method as shown in Fig 2 (b), which iteratively traverses remote vertices v_j, and their outgoing edges to randomly update local vertices v_i. The latency-aware pull method takes full use of cache locality, as after one cache miss for remote vertices, the succeeding loads will get cache hits.

Push mode. To reduce the number of atomic stores, we extend `CXL.mem` protocol with a custom atomic command named *fetch-and-ops*. The client node initiates the special atomic operations to store the source vertex attributes $attr$ to the address of destination vertex. The master node loads the destination vertex attributes $attr_next[v_j]$, process them with specific operations add, and stores updated values. As the operations in graph processing follow the commutative property, the master node can handle the requests based on their arrival order. For PageRank, the required atomic operation is fetch-and-add for double floating-point which is leveraged as the data format of vertex attribute (Fig 3 (b)).

C. Paradigm 2: Latency Hiding

Parallelism exploitation is a common method to hide latency overhead. By overlapping remote memory access with processing in the same or other essential phases in the graph processing execution model, the latency overhead of remote vertex traversing would also be diminished or even eliminated.

Pull mode. Although we can simply double the number of issued load instructions with a deeper queue to hide latency, the graph engine still needs to wait for return load values in a long period as the huge number of RMAs initiated in distributed graph processing would easily saturate the load instruction queue. To solve such problem, we propose an asynchronous load trying to overlap loads with processing in the Gather phase, as shown in Fig 2 (c). The graph engine explicitly invokes a large number of concurrent load operations ($async\_load$) to a specific hardware unit (Async Load Engine) to pull remote vertices, and continues processing without waiting for return values. Thus, the long-latency RMAs cannot block graph engine for a long time. The graph engine resumes processing on returned remote vertices ($async\_load\_results$) once the specific hardware unit informs it with polling or signals ($async\_load\_returned$).

As pull method traversing local vertices randomly, it starts the Gather phase only after all local vertices' attributes $attr\_current$ are prepared, and proceeds to the Apply phases when the Gather phase is done. Such strict data dependency restrains us to hide latency between computing phases or superstpes.

Push mode. To hide latency among the phases between supersteps, we propose relaxed barrier. In Fig 3 (a), the final values of vertex attribute $attr\_next$ are only fixed after the Gather phase, which traversing outgoing edges of local vertices v_i in push method. Due to dependency on new vertex attribute values $attr\_next$, the Apply phase is strictly serialized after the Gather phase ❶. However, there is no such strict data dependency between the Apply phase of superstep *N-1* and the Gather phase of superstep *N*, as the traversing operations of vertex v_i can be processed right after the update of $attr\_current[v_i]$ in the Apply phase of superstep *N-1*.

But the strict barrier between superstps stop us from fully interleaving two phases. We propose relaxed barrier, which replaces barrier between the Gather and Apply phases, so that fine-grained RMA in the Gather phases of superstep *N* are fully interleaved with processing of the Apply phase of superstep *N-1*, without damaging correctness of parallel algorithm execution, as shown in Fig 3 (c).

IV. Custom Design of CXL-oF

As there have not been yet available productions or open-sourced implementations, not to mention existing platforms to provide memory semantic interconnect, we need to design a lightweight hardware protocol stack to emulate basic functionalities and performance of CXL-over-fabrics (CXL-oF). We select `CXL.mem` protocol as the basis of our memory semantic fabric.

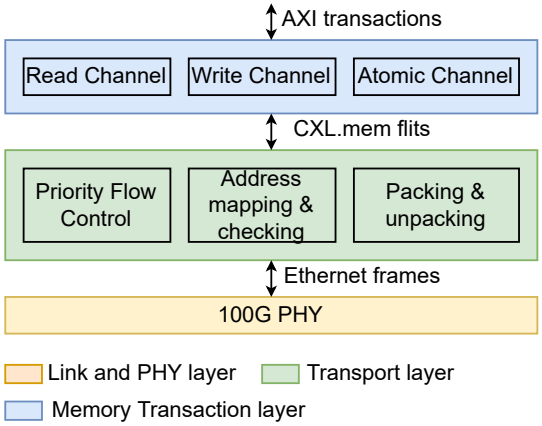

Fig. 4. The organization of functional units in the CXL-oF hardware stack (§IV).

As shown in Fig. 4, our custom CXL-oF protocol stack directly translates all relevant signals of an on-chip interconnect (e.g., ARM AMBA AXI protocol) to a `CXL.mem` flit and further encapsulates such flit as an Ethernet frame. Based on standard Ethernet infrastructure, such frame is delivered to adjacent computing nodes via either a direct back-to-back connection or a point-to-point connection over switch. With a standard on-chip interconnect interface, our proposed CXL-oF stack is easy to adopt with existing graph engines and memory controllers. This is especially crucial for FPGA system prototyping as the standard ARM AMBA AXI on-chip interconnect interface has been a widely used wrapper of various hard-core processors and soft-core intellegent properties in commodity FPGA chips. The custom CXL-oF hardware stack is fully pipelined to support high concurrency of on-chip interconnect transactions.

Memory Transaction layer. We arrange a memory address space that is configurable and visible to the graph engine, backed up by off-chip memory on each FPGA chip. The graph engine is allowed to directly initiate a write/read AXI transaction to CXL-oF. The extended atomic operations use AXI write transactions with user defined signals (awuser) to identify different operation types. In this layer, AXI write/read transactions are translated to MemRd/MemWr transactions of `CXL.mem`. As discussed in §III, we implement two atomic commands by extending the command field of `memory opcodes` as declared in `CXL.mem` protocol, *fetch-and-add* and *fetch-and-minimal*, respectively. In specific, *fetch-and-add* fetches a double data, adds it with new value, and writes the result back. Meanwhile, *fetch-and-minimal* fetches an unsigned data, and writes the new value when it less than fetched data. To match the state-of-the-art network speed (100Gbps throughput), we eliminate all states in transaction layer. To ensure correctness, the graph engine is responsible to maintain states of AXI transactions.

Transport layer. First, the address mapping and checking are applied to each flit. The continuous physical address space of each node are mapped to a continuous virtual address space. A series of virtual address spaces are combined as the address space exposed to the graph engine. The mapping relations need to be configured at the start time and stored in an on-chip buffer. For each flit, all mapping relations are checked simultaneously, and the destination address is mapped to destination IP. We return signals to indicate wrong transactions if the address checking fails. Then, We encapsulate flits into a UDP network packet with a couple of specified control fields (e.g., IP address field) to guarantee such packet is routable in standard Ethernet infrastructure. We handle network congestion caused packet loss and retransmission with priority flow control, which may has issues like head-of-line blocking [26]. We further rely on high-level flow control in graph engines to avoid triggering PFC as much as possible.

Link and PHY layer. In order to improve inherent fabric performance, We consolidate high-speed network (e.g., 100Gbps) into CXL-oF. We assign a fixed MAC address to each computing node to generate frames that are deliverable via commodity Ethernet switches.

	Frequency Reduction	Latency Hiding	Implementation challenges
CPU pull method	Modifications on algorithms	Independent pulling threads	☺
CPU push method	Atomic instructions added to ISA		☹
FPGA pull method	Hash-indexed requests table		☹
FPGA push method	Flow control	Adjusted sequence of Apply-Scatter-Gather kernels	☺

Fig. 5. A summary of implementation challenges of four reference architectures (§V).

V. Cluster Implementations based on GraCXL

In this section, we aim to integrate our custom CXL-oF hardware protocol stack with existing graph processing engines to build prototyping clusters to examine the effectiveness of our proposed GraCXL design paradigms. We select the general-purpose CPU cluster and the domain-specific FPGA accelerator array as two representative graph processing clusters to adopt GraCXL optimizations. We introduce four ref-

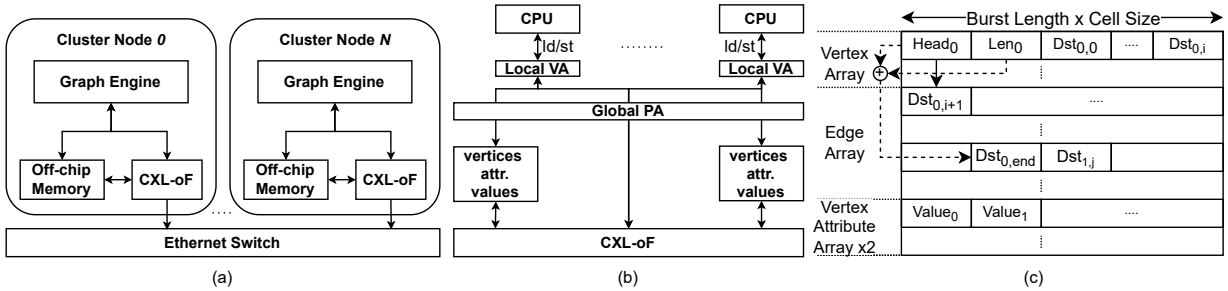

Fig. 6. A comprehensive system architecture of GraCXL is presented in (a). The data abstraction viewed by CPU in GraCXL-CPU is shown in (b). PA and VA refer to physical address and virtual address. We simplify and optimize graph data structure in Ligra framework for GraCXL-FPGA to improve local off-chip memory accessing performance, depicted in (c) (§V).

erence architecture designs with optimized execution models for each kind of cluster (summarized in Fig. 5).

In our proposed cluster, as shown in Fig. 6.(a), a couple of computing nodes are interconnected via our custom CXL-oF hardware protocol atop of a standard Ethernet switch. Each node is equipped with a custom *graph engine* accommodated to our optimized execution models as discussed in Section III. A *graph engine* is either a software thread running atop one CPU core or a hardware block that can be implemented in the FPGA logic. We use a straightforward, vertex index-based strategy leveraged in [14] to partition vertices and edges among computing nodes.

A. CPU cluster optimizations

We port the state-of-the-art lightweight graph processing framework for shared memory, Ligra to our platform. In Ligra, the graph is represented as adjacency list, and the vertices attributes values are stored in arrays. We store vertices attributes values in global shared memory region, so the CPUs can directly load or store remote vertices, as shown in Fig 6.b.

1) pull mode:

Frequency Reduction. The modifications on algorithms are only needed under the assumption that cache hierarchy will not evict the data until all remote vertices in the same cache line being processed. In the optimal scenario, the CPU will have a cache miss for several remote vertices. However, the cache mapping conflict with local data may evict prefetched remote vertices, especially when traversing outgoing edges of fat vertices. Considering there is no technology allocating cache space for specific data in ARM CPUs, we will see minor performance improvement on some graphs in Section VII-B.

Latency Hiding. The pre-prefetch instructions in current modern ARM CPU can fetch data into cache hierarchy, which can be regard as a kinds of asynchronous load. But same eviction problem remains as mentioned before, if we prefetch data into caches too early. In our design, we use pulling threads to load data from remote memory and store it in a FIFO, whose size is less than last-level-cache. In this way, the remote data will be in caches or in local memory when evicted. The execution thread stores remote data load requests to the tail of *Reqs* FIFO during graph processing. Then pulling thread handles requests at the head of the *Reqs* FIFO, loads remote data, and stores them to the tail of *resp* FIFO. The execution thread gets data from the head of *resp* FIFO and resumes processing immediately. As operations on shared FIFOs should require/release locks to guarantee correctness of multi-threads program, the overhead of lock/unlock will damage the potential preformance gains, as discussed in Section VII-B.

2) push mode:

Although we adds atomic operations to our custom CXL-oF hardware stack, the modifications on CPU and memory controller are also needed to generate such instructions and conduct required operations respectively. Modifications on CPU is time-consuming to build real system, as we prefer to conduct evaluations on the same testbed for CPU and FPGA. If such CPU is available, as the execution thread do not need to wait for response of atomic operations, it can generate much more concurrent RMA requests than pull model, whose concurrency is determined by the number of miss status handling registers (MSHR) in cache hierarchy. Thus, CPU-push will show better tolerance on memory latency than CPU-pull theoretically.

B. FPGA cluster design

As shown in Fig. 6(c), we leverage and optimize graph data used in Ligra framework, which is organized as three arrays: an *edge array* to sequentially store destination vertex indices of outgoing edges initiated from each source vertex, a *vertex array* maintaining the first index in the edge array as well as the number of outgoing edges for each vertex, and two *vertex attribute array* to record attribute values of each vertex updated at the end of each superstep. The *vertex attribute array* is stored in global shared memory.

1) pull method:

For eviction problem encountered in pull method of GraCXL-CPU, we use seperate on-chip memory as caches of local and remote data. To support asynchronous load and high currency of RMA, a hash-indexed *requests table* to store status of load requests is needed. Such data structure should support quick lookup when data returned out of order, as remote data may belong to different FPGAs. A pulling hardware unit atop CXL-oF unit receives load requests from accelerators, and it checks *requests table* with memory address to merge same requests. Then it inserts new

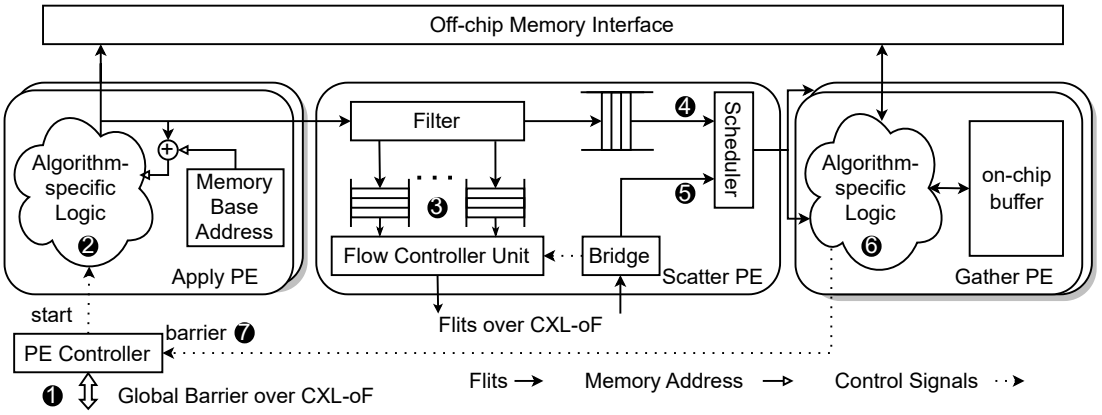

Fig. 7. Details of PEs in one accelerator. We implement a configurable hardware template in Apply PEs and Gather PEs, respectively, for flexible adoption of various graph algorithms. The *PE Controller* (PEC) initiates the `start` signal to trigger processing in one superstep. When capturing all `barrier` signals from PEs in current supersteps, PEC sends out one *global barrier* signal over CXL-oF and waits for other remote FPGA nodes' global barrier signals indicating the termination of current superstep (§V-B).

entry $< address, ID, accelerator\_ID >$ into *requests table*, and the hash collision can be resolved by hardware friendly open-addressing hash scheme [27]. When data returned, the *requests table* is checked to get accelerator ID, and the entry is deleted. As handling hash collision will incur long latency and a large *requests table* will occupy huge amount of on-chip storage resources to support thousands of concurrent RMAs, we category pull method of GraCXL-FPGA as hard to implement.

2) push method:

Frequency Reduction. As large number of atomic operations can be issued simultaneously, the saturated computing capability in remote nodes will cause end-to-end packets loss. Considering retransmission will double the RMA latency, a *flow control unit* (FCU) atop CXL-oF is needed to avoid end-to-end packets loss. In FCU, egress bandwidth of CXL-oF is equally allocated to each dispatch queue of remote FPGA node. A dynamic interval is inserted after one atomic operation has been drained by CXL-oF to mitigate egress flow to one FPGA node. A notification signals indicating the end of propagation of atomic operations in current FPGA node would be sent to other remote FPGA nodes. When the FCU in one remote node receives such signals, egress bandwidth allocated to the notifying node is rearranged to other FPGAs that still have atomic operations to propagate.

Latency Hiding. We implement three kinds of PEs (Fig. 7) to perform the relaxed barrier paradigm as discussed in Section III-C in a fully pipelined manner. All PEs in one graph engine are managed by a *PE Controller* (PEC) via a global barrier among FPGAs ❶ to indicate strict control boundary of each superstep. A set of *Apply PEs* (APEs) are designed to read attribute values from the *vertex attribute array 1* and execute the apply kernel ❷ of the previous superstep. APEs inject flits ($< dest\_vertex\_index, src\_attr\_value, OpCode >$) to the Scatter phase initiated from such active vertices. One *Scatter PE* (SPE) filters local flits from all APEs out to remote SPEs ❸. The SPE also schedules the unfiltered local flits ❹ and received remote flits ❺ to the local Gather phase. A couple of *Gather PEs* (GPEs) conduct the atomic operations ❻ to get new vertices attributes in current superstep with flits delivered from local SPE. Each GPE evokes a barrier signal ❼ to PEC when all flits in this GPE have been consumed and all vertex attribute values are updated to *vertex attribute array 2*. Details of logical structures of various PEs are discussed:

- *Apply PE (APE)*. In order to fully utilize off-chip memory bandwidth, we assign each APE directly to one port of the off-chip memory interface. Each APE deals with a portion of vertices in one superstep. As shown in Fig. 7, when receives a `start` signal from PEC, each APE is triggered to load assigned vertex with the base memory address of the *vertex attribute array 1*. Then, APE traverses outgoing edges of each active vertex. Finally, APE encapsulates index of one destination vertex, attribute values of the active source vertex and OpCode of the atomic operation into one flit. In one clock cycle, all flits encapsulated in each APE are simultaneously processed in the Scatter PE.

- *Scatter PE (SPE)*. When one flit is delivered from the previous Apply phase, the *filter unit* in SPE injects the flit into one of the *flit dispatch queues* according to the flit's destination vertex index field. Flits in dispatch queues are drained to either CXL-oF interface for inter-FPGA passing or local Gather PEs. SPE is also responsible for inter-FPGA flits receiving via the *bridge unit*, which translates AXI transactions into flits. All received inter-FPGA flits are scheduled to Gather PEs together with unfiltered local flits in a round-robin manner.

- *Gather PE (GPE)*. To guarantee sufficient computing capability for flit consumption, a couple of GPEs are adopted to update attributes of destination vertices in parallel. In each GPE, we setup an on-chip buffer to temporarily store updated attributes of vertices dominated by current FPGA node, avoiding random access to local memory. However, due to the limited capacity of on-chip memory resources, we divide vertices in each FPGA node into the same number of segments [14] and correspondingly decouple one superstep

into independent substeps. In one substep ss_i, vertex segment seg_i of each FPGA node is processed. At the end of ss_i, all on-chip updated values of seg_i are written back to the off-chip *vertex attribute array 2*. In order to avoid unnecessary data movement between two *vertex attribute array* at the end of each superstep, we further propose a ping-pong method to flexibly use these two arrays between consecutive supersteps.

VI. Experimental Setup

We build a consistent evaluation testbed for the different approaches with commercial FPGAs, despite that lots of hardware work are needed to make systems run correctly. Considering implementation challenges of reference architectures proposed in Section V, we choose to implement prototypes of GraCXL-CPU and GraCXL-FPGA to evaluate pull method and push method respectively.

Platform setup. As shown in Fig. 8, we build a prototype system with four custom FPGA nodes based on Xilinx ZynqMP chips. Each node is attached to the same Ethernet Switch by fiber and optical modules via the FPGA's 100Gbps transceivers. Each node is also equipped with a 16GiB off-chip DDR4 SODIMM. We leverage Xilinx Vivado 2020.2 for logic synthesis and implementation.

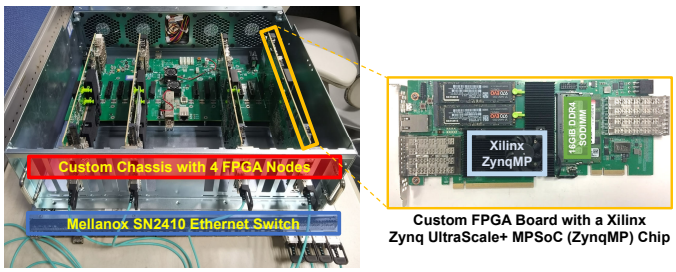

Fig. 8. The prototype with 4 individual custom nodes based on manufactured circuit boards with the largest chip in Xilinx's ZynqMP series, XCZU19EG (tightly-coupled FPGA logic with a hard ARM Cortex-A53 quad-core CPU) in a custom chassis. Each FPGA node is attached to the same Mellanox SN2410 Ethernet switch. The chassis is able to afford 16 custom nodes at maximum. (§VI)

Graph workload. We select three real-world graphs [28] as well as two scale-free synthetic graphs with different sizes. The synthetic graphs are obtained from RMAT generator [29]. Properties of each selected graph are summarized in Table I. We conduct BFS and PageRank, two representative graph algorithms, in performance evaluations. We leverage the performance metric *traversed edges per second* (TEPS) which depicts system capability to fetch and process edge data from off-chip memory.

Graph engine configurations. For GraCXL-FPGA, the graph engine is described in high-level Chisel language [30]. As listed in Table II, we adopt four APEs in our proposed graph engine, each of which is attached to one 128-bit AXI port exposed by the off-chip memory controller. Correspondingly, 16 GPEs are instantiated to simultaneously consume the maximum number of flits directly delivered from local APEs. We leverage Block RAM to construct the configurable on-chip buffer in each GPE. Note that dispatch queues in the SPE are organized by flip-flops to reduce on-chip memory occupations. For GraCXL-CPU, we use two pulling threads and one execution thread, and each thread occupy an independent core.

TABLE I
Properties of workloads in GraFF evaluation (§VI)

Workload	# Vertices	# Edges	Average Degree
Real-world Graphs [28]			
com-orkut (OR)	3.07 M	234 M	76.2
soc-LiveJournal1 (LJ)	4.84 M	69 M	14.3
twitter-rv (TW)	41.65 M	1468.37 M	35.3
Synthetic Graphs			
graph500-26 (G26)	64 M	1024 M	16
RMAT-24 (R24)	16 M	512 M	32

TABLE II
Configurations of one graph engine (§VI)

Parameters	Value
GraCXL-FPGA	
frequency	200 MHz
# APEs	4
# GPEs	16
# SPEs	1
on-chip buffer in GPEs	16x 1.02Mib
GraCXL-CPU	
frequency	1.33 GHz
# execution threads	1
# pulling threads	2
CXL-oF	
frequency	300 MHz

We evaluate our prototype to answer the following research concerns:

- **C1.** What are the benefits of our custom CXL-oF implementation (§VII-A) ?
- **C2.** What is the effectiveness of GraCXL introduced to a pull-mode CPU cluster (§VII-B) ?
- **C3.** What is the advantage in a GraCXL-based multi-FPGA system compared to existing clusters (§VII-C) ?

VII. Evaluations

A. Performance of custom CXL-oF protocol stack

CXL-oF uses less than 5% of logic, registers and memory resources thanks to its stateless design. We conduct and record latency tests for 50 times to estimate the performance of CXL-oF. As shown in Fig. 9, the round trip time (RTT) of a cache-line size (64 byte) remote load access exhibits stability for both direct back-to-back and indirect point-to-point via Ethernet switch. The average latency is 0.922μs and 1.543μs in back-to-back and point-to-point configurations, respectively.

B. Effectiveness of GraCXL in CPU cluster

To show performance improvement of each optimization, we compare performance of two computing nodes running both algorithms without optimization (normalization baseline),

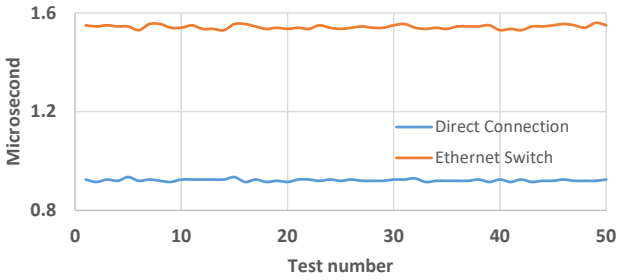

Fig. 9. The latency measurements of CXL-oF with a dual-node system in either a point-to-point connection via Ethernet switch or a direct back-to-back connection. The packet generator sends one load request at a time. The packet consumer return random data right after the request been received (§VII-A).

with latency aware optimization (OPT-1), with asynchronous load optimization (OPT-2), and with all optimizations (OPT-all) in Fig. 10.

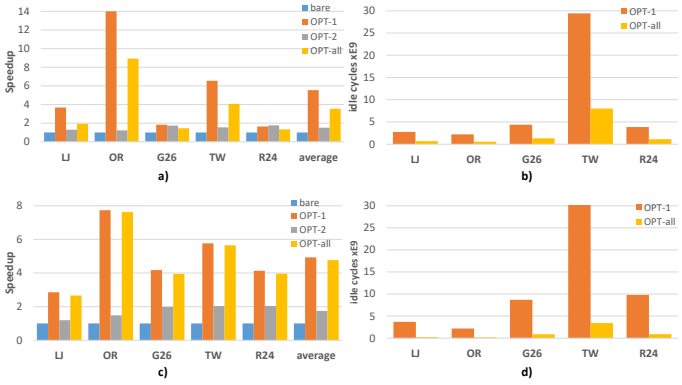

Fig. 10. GraCXL-CPU throughput (MTEPS) running on five graphs with different optimizations. The results of BFS and PageRank is shown on (a) and (c) respectively. The idle time is measured by the CPU cycles, which we count it with the cycle counter register in the performance monitor unit ($PMCCNTR\_EL0$), and present it in (b) and (d). (§VII-B).

Our proposed latency-aware optimization and asynchronous load optimization brings 1.83x-14x and 1.19x-1.98x improving in throughput, respectively. However, *OPT-all* shows almost none performance improvement than *OPT-1*, and even decrease throughput in some test cases. To find out performance bottleneck of *OPT-all*, we split running time of the execution thread into idle time and busy time. The idle time refers to execution thread polling returned data from *resp* FIFO. As shown in Figure10, a huge portion of the idle time has been removed after applying *OPT-2* to *OPT-1*. Thus, the overhead of lock/unlock operations in threads communication leads to damaging performance of GraCXL-CPU in some scenario.

The above results show we can shift system bottleneck from long latency remote memory accessing to randomly local memory accessing with proposed optimizations. We can further improve system performance if a light-wight threads communication software stack is adopted.

C. Advantages of GraCXL in FPGA cluster

Resource Occupations. GraCXL-FPGA achieves high frequency and uses about 36% of logic resources and half of memory resources. We also notice that the SPE only occupies about 27% LUTs for flit passing among four FPGAs, avoiding significant occupations of on-chip memory resources to hold messages in message-passing model. Such resource efficiency mainly lies in the simplicity of flip-flop-based dispatch queues as well as a couple of cycle counters introduced by our fully pipelined architecture and benefits of CXL-oF.

Performance Analysis. To show sever RMA latency overhead in the GraCXL-FPGA, we measure the maximum total inter-FPGA flits size which ranges 68.5-2163.5 MiB, and accumulated RMA latency overhead which occupies a significant portion (17.9%-37.5%) of the total execution time. Although the fine-grained asynchronous atomic operations help to save CXL-oF bandwidth and overlap RMA latency with execution time. When execution time within Gather phases is less than synchronization time, the relaxed barrier is needed to further improve performance.

To show the effectiveness of relaxed barrier, we measure throughput of a system with asynchronous atomic operations (*OPT-1*). We make APEs write vertices attributes values of the last superstep to on-chip buffer (BRAM), then send global barrier. Due to capacity limitation on BRAM, we only run small graphs whose number of vertices is less than 4096. As shown in Figure 11, GraCXL-FPGA with relaxed barrier (*OPT-all*) further improves 1.05x-1.66x throughput compared with *OPT-1*.

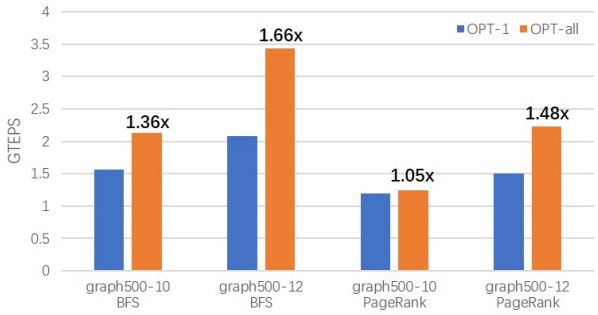

Fig. 11. The throughput (GTEPS) for the system with asynchronous atomic operations on four combinations of graph workloads and algorithms. The speedup of *OPT-all* compared to *OPT-1* is shown on top of bars. (§VII-C).

With the help of asynchronous atomic operations and relaxed barrier, such latency overhead has been greatly relieved. Table III shows that throughput of GraCXL-FPGA with 4 FPGAs reaches up to 7.28 and 9.01 GTEPS in BFS and Pagerank, respectively, for graph workloads with high average degree. Such workloads always exhibit less randomness to access off-chip memory for edge traversing due to high density in the vertex array and large consecutiveness in the edge array.

To find out the performance bottleneck of GraCXL-FPGA, We further investigate each PE in BFS via a series of performance counters to measure status of pipeline stall due to flit congestion in the graph engine. In each graph workload, pipeline stall cycles caused by GPEs occupy negligible 1% of total execution cycles. We also observe only 6.45GB/s peak fabric bandwidth via Ethernet switch, indicating that the

TABLE III
SYSTEM THROUGHPUT OF FPGA-PUSH WITH VARIOUS NUMBERS OF FPGA NODES RUNNING DIFFERENT GRAPH WORKLOADS (§VII-C).

Workload	BFS		PageRank	
	Throughput (GTEPS)	Time (ms)	Throughput (GTEPS)	Time (ms)
com-orkut	7.28	32.21	9.01	312.19
soc-LiveJournal1	2.65	25.69	5.96	138.47
twitter-rv	5.02	282.31	7.56	2331.88
RMAT-24	6.23	172.32	8.40	1534.51
graph500-26	5.58	384.77	7.28	3537.74

SPE is unsaturated by inter-FPGA flit passing within current workload. Finally, we measure as low as 0.3% portion of idle cycles in which no read transaction exists in four AMBA AXI channels to off-chip DRAM during APEs' edge traversing, which means that off-chip memory bandwidth is fully utilized by APEs in our current design.

The above analysis has show we have shifted system bottleneck from significant latency gap of RMA to randomly local memory accessing using asynchronous atomic semantic operations and relaxed barrier.

Performance comparison. We compare performance of GraCXL-FPGA with existing multi-FPGA systems using results on the same datasets (twitter-rv, soc-LiveJournal1 and RMAT-24). For fairness, we calculate the resource utilization rate of PEs on chips adopted by existing systems, and GraCXL-FPGA can fit into their chips. Besides, as these systems all work at the same clock frequency (200MHz), different FPGA chips has little impact on the performance improvement. Figure 12 shows that GraCXL-FPGA obtains 2.48x-5.01x progress in throughput against the state-of-the-art shared-memory system, and 1.13x-3.73x against message-based systems.

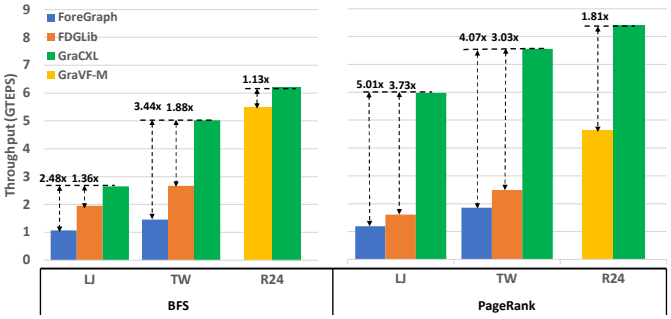

Fig. 12. The throughput (GTEPS) comparison with the state-of-the-art systems based on peak performance reported in their papers, since most of the related works have not been open-sourced (just as previous works have also done). S and M refer to shared-memory and message-passing respectively. The speedup against GraCXL-FPGA is shown on top of their bars (§VII-C).

VIII. RELATED WORK

Distributed shared memory system. Distributed shared memory systems, such as FaRM [31] and FaSST [32], utilize a cluster of nodes that pool memory into a global address space. FaRM features a cluster of symmetric compute nodes that can perform collective communication and two sided RPC via RDMA. These systems are not designed for characters of graph applications, and builds on traditional net fabrics. GraCXL is designed to tackle challenges met by graph applications under the new memory semantic interconnections circumstance.

Memory semantic fabric. A couple of academia studies have recently focus on using CXL for memory disaggregation. DirectCXL [33] implemented preliminary CXL stack on FPGAs. TPP [25] and Pond [34] simulates the latency of CXL on a dual-socket system. GraCXL tries to leverage CXL in graph processing and to break boundry of chasis by extending CXL.mem protocol over net fabric. Other emerging industrial open standards are also implemented for memory disaggregation and CPU-FPGA coordination [35]–[38], without customized for graph processing.

Graph processing multi-FPGA system. ForeGraph [14] simulates a 4-FPGA platform on one FPGA board. ForeGraph loads source vertices from other boards to local memory in turns and computes updating of local destination vertices. GraVF-M [10] implements a 4-FPGA platform on real machines, Micron AC-510 Modules with HMC. GraVF-M uses a large update queue stores the messages needs to be sent in the next superstep, and broadcasts them at the very beginning of each superstep. FDGLib [8] builds a 32-FPGA Microsoft Catapult-like cluster. FDGLib introduces a easy-to-use communication library, aiming to scale out any single FPGA-based graph accelerator to a distributed version.

IX. CONCLUSION

In this work, we propose GraCXL, a series of design paradigms to address high-frequency and long-latency of remote memory access potentially incurred in the CXL-based shared-memory graph processing clusters. We design and implement a custom fabric with the CXL.mem protocol and leverage the ARM SoC-equipped FPGAs to build an evaluation prototype for CPU and FPGA. Experimental results show that the GraCXL CPU and FPGA clusters achieve 1.33x-8.92x and 2.48x-5.01x performance improvement, respectively. We would like to open source of GraCXL described in Chisel to the community.

ACKNOWLEDGMENTS

This work is supported by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant No. XDA0320000 and XDA0320300, by the CCF-Huawei Populus Grove Fund under Grant No. CCF-HuaweiTC2022001, by the Virtual Teaching and Research Section of the Ministry of Education of P. R. China for the Computer System and Processor Courses, by the Beijing Municipal Education Commission under Grant No. 201914430004, by the National Natural Science Foundation of China under Grant No. 62090023 and 61702485, by the Youth Innovation Promotion Association of CAS under Grant No. 2017143. We appreciate anonymous reviewers for their valuable feedbacks.

REFERENCES

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010, pp. 135–146.

[2] S. Eubank, H. Guclu, V. Anil Kumar, M. V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang, "Modelling disease outbreaks in realistic urban social networks," *Nature*, vol. 429, no. 6988, pp. 180–184, 2004.

[3] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *Proc. International Conference on Learning Representations (ICLR) 2019*, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. of USENIX OSDI*, 2012, pp. 17–30.

[5] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*, 2018, pp. 752–768.

[6] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "Gram: Scaling graph computation to the trillions," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 408–421.

[7] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 3, pp. 1–39, 2019.

[8] Y.-W. Wu, Q.-G. Wang, L. Zheng, X.-F. Liao, H. Jin, W.-B. Jiang, R. Zheng, and K. Hu, "FDGLib: A communication library for efficient large-scale graph processing in FPGA-accelerated data centers," *Journal of Computer Science and Technology (JCST)*, vol. 36, no. 5, pp. 1051–1070, 2021.

[9] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A Computation-Centric Distributed Graph Processing System," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, 2016, p. 301–316.

[10] N. Engelhardt and H. K.-H. So, "GraVF-M: Graph processing system generation for multi-FPGA platforms," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 4, pp. 1–28, 2019.

[11] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 135–146.

[12] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A High-Performance Graph DSL," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018.

[13] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory," *Proc. VLDB Endow.*, vol. 13, no. 8, p. 1304–1318, may 2020.

[14] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proc. of ACM/SIGDA FPGA*, 2017, pp. 217–226.

[15] H. Fu, M. Gorentla Venkata, S. Salman, N. Imam, and W. Yu, "SHMEMGraph: Efficient and balanced graph processing using one-sided communication," in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 513–522.

[16] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16, 2016, p. 105–110.

[17] D. D. Sharma, "Compute express link," *CXL Consortium White Paper.[Online]. Available: https://docs. wixstatic. com/ugd/0c1418 d9878707bbb7427786b70c3c91d5fbd1. pdf*, 2019.

[18] Gen-Z Consortium, "Gen-Z final specifications," https://genzconsortium.org/specifications.

[19] I. NVM Express, "Nvm express base specification," https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2_0-2021.06.02-Ratified-5.pdf, Jun. 2021.

[20] ——, "Nvm express over fabrics," https://nvmexpress.org/wp-content/uploads/NVMe_over_Fabrics_1_0_Gold_20160605.pdf, Jun. 2016.

[21] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on gpus," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 1–19.

[22] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng *et al.*, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 615–628.

[23] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[24] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison, "Ibm power9 opens up a new era of acceleration enablement: Opencapi," *IBM Journal of Research and Development*, vol. 62, no. 4/5, pp. 8–1, 2018.

[25] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: Transparent Page Placement for CXL-Enabled Tiered Memory," 2022.

[26] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over Commodity Ethernet at Scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, 2016, p. 202–215. [Online]. Available: https://doi.org/10.1145/2934872.2934908

[27] I. Yaniv and D. Tsafrir, "Hash, don't cache (the page table)," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 337–350, 2016.

[28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[29] D. Chakrabarti and C. Faloutsos, "The (recursive matrix) graph generator," in *Graph Mining*. Springer, 2012, pp. 81–86.

[30] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proc. Design Automation Conference (DAC)*, 2012, pp. 1212–1221.

[31] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast Remote Memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14, 2014, p. 401–414.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, 2016, p. 185–201.

[33] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, High-Performance memory disaggregation with DirectCXL," in *Proc. USENIX Annual Technical Conference (ATC)*, 2022, pp. 287–294.

[34] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2023.

[35] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, 2022, p. 434–451.

[36] S. Tamimi, F. Stock, A. Koch, A. Bernhardt, and I. Petrov, "An evaluation of using CCIX for cache-coherent host-FPGA interfacing," in *Proc. IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–9.

[37] C. Pinto, D. Syrivelis, M. Gazzetti, P. K. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "ThymesisFlow: A software-defined, HW/SW co-designed interconnect stack for rack-scale memory disaggregation," in *Proc. IEEE/ACM International Symposium on Microarchitecture, (MICRO)*, 2020, pp. 868–880.

[38] W.-o. Kwon, S.-W. Sok, C.-h. Park, M.-H. Oh, and S. Hong, "Gen-Z memory pool system implementation and performance measurement," *ETRI Journal*, vol. 44, no. 3, pp. 450–461, 2022.