Clementi: Efficient Load Balancing and Communication Overlap for Multi-FPGA Graph Processing

FENG YU, National University of Singapore, Singapore
HONGSHI TAN, National University of Singapore, Singapore
XINYU CHEN, The Hong Kong University of Science and Technology (Guangzhou), China
YAO CHEN*, National University of Singapore, Singapore
BINGSHENG HE, National University of Singapore, Singapore
WENG-FAI WONG, National University of Singapore, Singapore

Efficient graph processing is critical in various modern applications, such as social network analysis, recommendation systems, and large-scale data mining. Traditional single-FPGA systems struggle to handle the increasing size and complexity of real-world graphs due to limitations in memory and computational resources. Existing multi-FPGA solutions face significant challenges, including high communication overhead caused by irregular data transfer patterns and workload imbalances stemming from skewed graph distributions. These inefficiencies hinder scalability and performance, highlighting a critical research gap. To address these issues, we introduce Clementi, an efficient multi-FPGA graph processing framework that features customized fine-grained pipelines for computation and cross-FPGA communication. Clementi uniquely integrates an accurate performance model for execution time prediction, enabling a novel scheduling method that balances workload distribution and minimizes communication overhead by overlapping communication and computation stages. Experimental results demonstrate that Clementi achieves speedups of up to 8.75× compared to state-of-the-art multi-FPGA designs, indicating significant improvements in processing efficiency as the number of FPGAs increases. This near-linear scalability underscores the framework's potential to enhance graph processing capabilities in practical applications. Clementi is open-sourced at https://github.com/Xtra-Computing/Clementi.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; • **Computer systems organization → Data flow architectures**; • **Computing methodologies → Parallel algorithms**.

Additional Key Words and Phrases: Multi-FPGA graph processing, load balancing, communication overlapping.

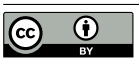
ACM Reference Format:
Feng Yu, Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, and Weng-fai Wong. 2025. Clementi: Efficient Load Balancing and Communication Overlap for Multi-FPGA Graph Processing. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 138 (June 2025), 27 pages. https://doi.org/10.1145/3725275

1 Introduction

Graph representations are pivotal for myriad applications, including data science, machine learning, social networks, roadmaps, and genomics, as they succinctly depict the inherent relationships between distinct entities [18]. This has prompted the development of various platforms for graph

*Corresponding author.

Authors' Contact Information: Feng Yu, National University of Singapore, Singapore, yuf@u.nus.edu; Hongshi Tan, National University of Singapore, Singapore, hongshi@u.nus.edu; Xinyu Chen, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, xinyuchen@hkust-gz.edu.cn; Yao Chen, National University of Singapore, Singapore, yaochen@nus.edu.sg; Bingsheng He, National University of Singapore, Singapore, hebs@comp.nus.edu.sg; Weng-fai Wong, National University of Singapore, Singapore, wongwf@comp.nus.edu.sg.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2836-6573/2025/6-ART138
https://doi.org/10.1145/3725275

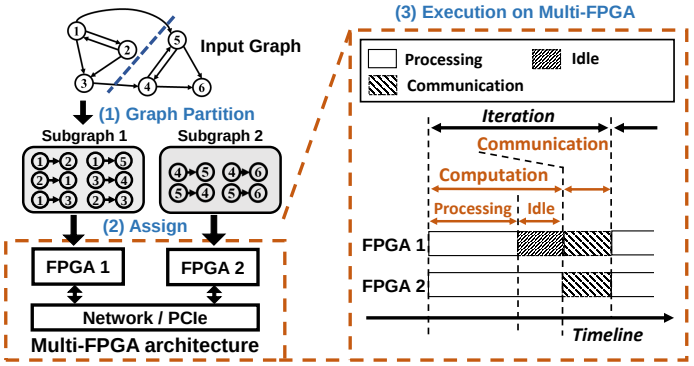


Fig. 1. Simplified graph processing workflow on a multi-FPGA platform.

processing and analysis, including CPU-based designs [8, 17, 29, 30, 47, 57], GPU-based designs [15, 23, 28, 43, 45, 55], and FPGA-based designs [3, 10–13, 19, 42, 56]. Owing to its inherent attributes such as reconfigurability, fine-grained parallelism, and energy efficiency, FPGA emerges as a highly promising option for accelerating graph analytics in datacenters [36, 46].

As the size of real-world graphs continues to grow at an unprecedented rate, traditional graph processing techniques are insufficient to handle the scale of data [16, 27, 40]. Specifically, the resource limitations of a single FPGA board become apparent, such as the on-chip block RAM and the on-board DDR memory. As a result, current approaches explore distributed designs that leverage multiple FPGAs with graph partition methods, demonstrating impressive performance in graph processing with multiple FPGAs [13, 14, 39, 48, 53].

Graph processing on multi-FPGA commonly (1) partitions input graph data into several subgraphs, (2) assigns partitioned subgraphs to each FPGA, and (3) executes graph analysis over the multi-FPGA architecture, which includes both computation and communication stages. As shown in Figure 1, using the widely-used graph processing application PageRank as an example, each FPGA retains its assigned subgraphs in its own memory. During processing iterations, each FPGA processes its assigned edge list, computing the sum of vertex properties for each destination vertex to derive new properties. Subsequently, a communication stage updates these new vertex properties across all FPGAs. A global synchronization barrier at the end of each iteration ensures data consistency in a multi-FPGA scenario.

However, current multi-FPGA designs suffer from poor scale-out performance due to the following problems: Firstly, the synchronization barrier caused by the global update prevents the overlapping of the communication and computation process in different FPGAs. This can be observed in designs such as ForeGraph [13] and GraVF-M [14], where the FPGAs require synchronization after the assigned subgraphs have been processed. Secondly, skewed graph data distribution leads to an imbalanced workload, resulting in FPGA under-utilization and thus undermining the overall scalability in a multi-FPGA environment.

For a detailed analysis, we profiled ForeGraph [13], a state-of-the-art example. In Figure 2, we define the Normalized System Runtime as the sum of normalized computation time and communication time. To provide a more detailed breakdown, we introduce Total Active PE Time, which represents the cumulative processing time required if all processing elements (PEs) were fully utilized. The Idle Time represents the periods when PEs are underutilized due to workload imbalance, and Communication Time represents the overhead caused by data transfers between FPGAs. The results reveal that when using four FPGAs for the TW dataset [24] (41.6 million vertices and 1.47 billion edges), ForeGraph incurs a significant communication overhead, which accounts for up to

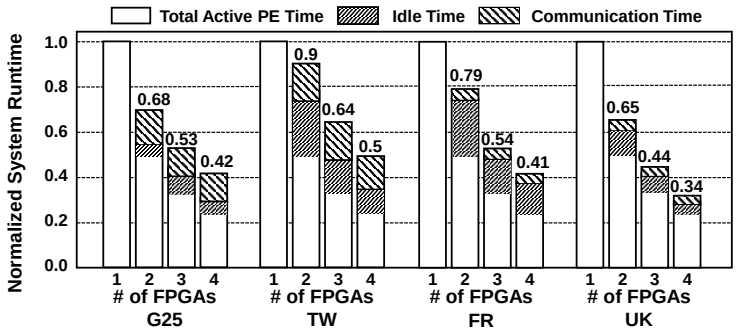


Fig. 2. System runtime profiling of ForeGraph [13] on varying number of FPGAs.

28% of the total execution time, while there is a 22% increase in computation time due to idle PEs across FPGAs, resulting in a 50% overall performance degradation.

In this paper, we propose Clementi, a multi-FPGA graph processing framework to address these two critical issues. We first adopt the GAS graph processing model over a multi-FPGA architecture, selecting the remote-apply pattern to better overlap communication and computation. Based on this pattern, we customize fine-grained hardware pipelines in each FPGA for the graph computation and communication stages. Furthermore, we propose an architecture-oriented performance model that captures the execution behavior of these pipelines for partitioned subgraphs, enabling an effective scheduling method that assigns partitioned subgraphs to different FPGAs. This method not only balances execution time across all FPGAs, but also overlaps communication stages with computation stages, effectively reducing communication overhead. The major contributions of our proposed method are as follows:

- We build Clementi, a multi-FPGA graph processing framework that achieves efficient load balancing and communication overlapping, demonstrating significant performance improvements when scaling out.
- We design fine-grained, customized pipelines that maximize the utilization of off-chip memory and network bandwidth while enabling communication-computation overlap.
- We develop an effective workload scheduling method to address the load balancing issues of graph processing among multiple FPGAs. The scheduling is based on a novel architecture-oriented performance model to capture the execution behavior of different stages in graph processing.
- Clementi delivers the state-of-the-art performance, achieving maximum 19.7 GTEPS, and a speedup of up to 8.75× when compared to ForeGraph [13] and GraVF-M [14] with the same number of FPGAs. Clementi also delivers comparable throughput to multi-CPU design Gemini [57] and multi-GPU design Lux [21], along with superior energy efficiency when scaling out.

2 Preliminary

2.1 GAS Execution Model

The Gather-Apply-Scatter (GAS) model [17, 38] is a widely used framework in graph processing. It defines three conceptual stages: (1) Gather: properties of adjacent vertices and edges are collected. (2) Apply: a user-defined function (UDF) computes the new vertex property based on the collected values from the gather stage. (3) Scatter: the new vertex values are propagated to update the properties of adjacent vertices. In vertex-centric processing, the GAS model operates by iterating over all graph vertices and traversing their adjacent edges. In contrast, edge-centric processing sequentially processes all edges, naturally transitioning into a 'scatter-gather-apply' (SGA) sequence. This reordering optimizes memory access patterns by first scattering edge updates to minimize

random memory accesses, followed by gathering vertex updates locally for efficient processing. Following previous works [17, 38], we still name the edge-centric execution model as GAS model, with the pseudo code in Algorithm 1.

Multi-FPGA execution: When extending the edge-centric GAS model over the multi-FPGA architecture, two practical processing patterns are identified. ① **Remote-scatter Pattern:** The current FPGA implements remote accesses to vertex data during the scatter stage, transmitting only vertex properties not present on the current FPGA while calculating and updating vertex data locally. ② **Remote-apply Pattern:** Each FPGA stores replicas of the vertex set along with their associated properties in local memory during the scatter and gather stages and synchronizes updates to these vertex replicas across FPGAs during the apply stage of each iteration.

Algorithm 1 Edge-centric GAS Model

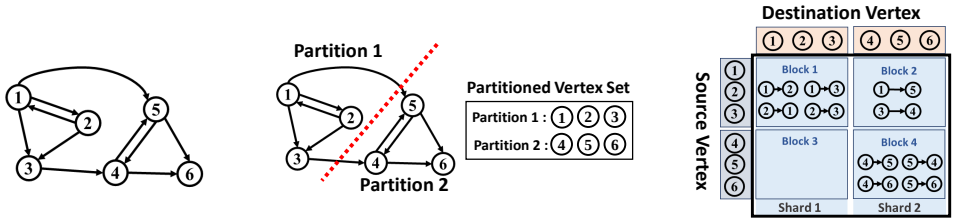
Input: **Edges**: edge list. **Vertices**: vertex set.
Output: **NewVertices**: calculated vertex set.
Notations: **dst**: destination vertex of an edge. **src**: source vertex of an edge. **w**: weight of the edge. **prop**: vertex property. **accum**: accumulated propagated property for each vertex. **value**: propagated vertex property of an update item.

```
1:  while not done do
2:      for all e ∈ Edges do          /* Scatter Stage: Propagate property from src to dst vertices.*/
3:          u ← new update item
4:          u.dst ← e.dst
5:          u.value ← Scatter(e.weight, e.src.prop)
6:      end for
7:      for all u ∈ Updates do        /* Gather Stage: Aggregate updates at each dst vertex.*/
8:          u.dst.accum ← Gather(u.dst.accum, u.value)
9:      end for
10:     for all v ∈ Vertices do       /* Apply Stage: Update vertex prop based on accum.*/
11:         v.prop ← Apply(v.prop, v.accum)
12:     end for
13: end while
14: NewVertices ← Vertices
15: return NewVertices
```

Both patterns ensure correct computation results, but differ in how remote vertex access and data updates are handled. The remote-scatter pattern performs random remote vertex access during the scatter stage and updates only local vertices during the gather and apply stages. In contrast, the remote-apply pattern features sequential remote vertex access during the apply stage, while scatter and gather operations are performed locally.

2.2 Graph Partition

Graph partition plays an important role in FPGA-based graph processing designs [3, 9–13, 19, 34, 42, 56] to realize efficient on-chip memory utilization and massive parallelism. Current graph partition methods used in FPGA-based graph processing mainly consist of Metis [22] and 2D partition method [6, 54]. Metis [22] is a widely used graph partition method by reducing cross-cutting edges, shown in Figure 3(b). It is important to note that Metis relies on heuristic algorithms that may not always guarantee optimal workload balance. Compared to Metis, the 2D partition method, also known as edge partition, partitions a large graph into several subgraphs in a two-dimensional



(a) Original graph (b) Metis partition method (c) 2D partition method

Fig. 3. Graph partition.

manner. It initially divides the original edge list into `shards` based on the destination vertex range using an interval-shard partitioning method. Subsequently, it conducts a second dimension partition on the `shard` to generate `blocks` for a finer-grained graph subpartition. As shown in Figure 3(c), the input edge list is divided into several `blocks` based on their source vertex index range.

3 Challenges and Motivations

To quantitatively evaluate the bottlenecks of the existing designs, we profile the existing state-of-the-art multi-FPGA graph processing solution, ForeGraph [13], with four large-scale graph datasets, namely G25, TW, UK, and FR (detailed in Table 1). The profiling result is obtained through simulation with a system setting of 12.25 Gbps network bandwidth, 19.2 GB/s DDR off-chip memory bandwidth, and the assumption of PEs processing at their peak throughput [13], shown in Figure 2. The profiling results demonstrate that both the communication and imbalanced workload increase the total system runtime significantly in each iteration, leading to poor scale-out performance. As a result, when executing on four FPGAs, the time increase due to communication overhead and imbalanced workload accounts for 11%-28% and 12%-32% of the overall time, respectively. Consequently, this time cost constraints the acceleration to a range of 2× to 2.94× with 4 FPGAs.

Based on the profiling experiment, we identify two critical issues in existing multi-FPGA graph processing systems: communication overhead and imbalanced workload. However, resolving these critical issues presents many design challenges.

Challenge 1: To address the significant overhead caused by the communication stage in graph processing, we can overlap it with computation. However, this approach requires us to accurately capture the behavior of both communication and computation stages to devise an effective scheduling plan. The challenge lies in the massive and irregular data communications during the processing of the partitioned subgraphs, which hinder our ability to obtain accurate behavioral insights necessary for optimizing the overlap of these stages.

Challenge 2: To address the imbalanced workload among multiple FPGAs, we need to obtain an accurate execution time of the workloads (subgraphs) on the FPGAs. Due to the random and irregular data access patterns, it is challenging to accurately predict the execution time of graph processing tasks in each of the FPGAs. This unpredictability complicates the task of achieving balanced scheduling among multiple FPGAs.

To overcome these challenges, we have the following design motivations.

Motivation 1: Instead of using the remote-scatter pattern, we adopt the remote-apply pattern in our solution, which aggregates data communications in the apply stage. This approach mitigates the potentially high-frequency and non-deterministic remote vertex access inherent to the remote-scatter pattern. By aggregating data communications, we aim to improve network transmission efficiency, allowing the execution time of the communication stage to be more accurately modeled as a function of data volume and network bandwidth, providing predictable communication behavior.

Motivation 2: In addition, the absence of a detailed runtime performance model prevents us from balanced execution across FPGAs, it draws the requirement of a customized fine-grained graph computation pipeline that ensures predictable runtime performance.

4 Clementi Overview

Clementi is a multi-FPGA graph processing framework that adopts an edge-centric GAS execution model with a remote-apply pattern. Each FPGA is configured with the same architecture and interconnected via an FPGA-side network. Additionally, each FPGA is attached to a CPU node through PCIe, which manages FPGA kernel control and facilitates data transfers between the CPU and FPGA memory. Notably, the CPU is only active at the beginning and end of the graph processing, managing the initial graph data transfer, and returning the final results.

Clementi adopts a four-phase approach for graph processing on a multi-FPGA architecture: (1) Clementi first partitions the input graph into several subgraphs using an interval-shard partition method based on the destination vertex range. (2) Secondly, the performance model estimates the execution time of the partitioned subgraphs. (3) Thirdly, a greedy-based scheduling method assigns partitioned subgraphs to FPGAs based on the estimated execution times. (4) Finally, each FPGA processes its assigned subgraphs concurrently. Specifically, each FPGA deploys multiple gather-scatter sub-modules to process the subgraph's edge list. An input-aware partition method is used to further partition the edge lists for balanced processing. Figure 4 presents an overview of the Clementi workflow, illustrating how the input graph is partitioned into **T** subgraphs and subsequently assigned to **N** FPGAs via a scheduling method.

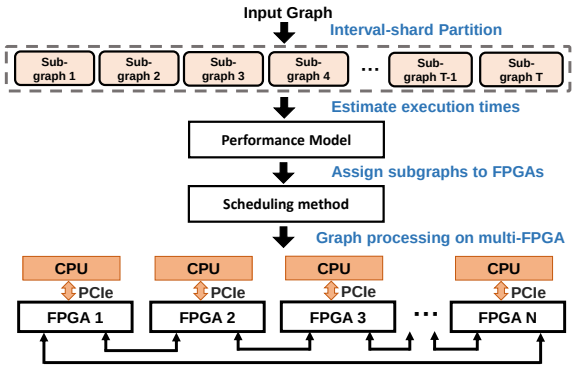


Fig. 4. Overview of the Clementi workflow.

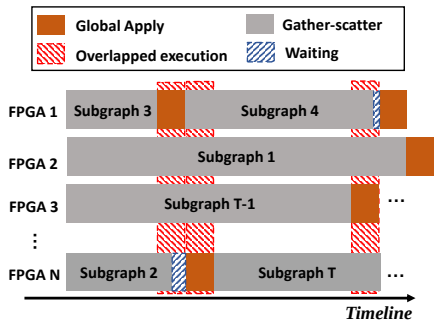


Fig. 5. Execution timeline of Clementi.

The remote-apply pattern implemented in Clementi requires each FPGA to maintain a complete copy of the vertex set in its memory, eliminating the need for remote vertex access during the gather-scatter stage. Updates to vertex data across FPGAs are limited to the apply stage, referred to as the global apply stage, where an all-gather operation is performed to synchronize vertex updates across all FPGAs. Figure 5 depicts the detailed execution timeline, showcasing the overlapping gather-scatter (computation) and global apply (communication) stages across **N** FPGAs for the partitioned **T** subgraphs. During the global apply stage, each FPGA distributes its computed destination vertex properties to all other interconnected FPGAs. To ensure predictable data transfer paths and reduce communication complexity, Clementi adopts a ring topology for FPGA interconnection. This topology enables sequential, single-direction data updates, mitigating the routing complexity and contention issues commonly associated with bus-based, point-to-point, and mesh typologies.

5 Clementi Architecture Details

To minimize the scaling-out effort, Clementi employs a unified architecture across all FPGAs, with each FPGA connected to others through its dedicated 100 Gbps FPGA-side network stack. The architecture of Clementi, depicted in Figure 6, integrates three types of functional modules that collaboratively execute the edge-centric GAS graph processing model, following the remote apply pattern. These modules are designed to address distinct tasks:

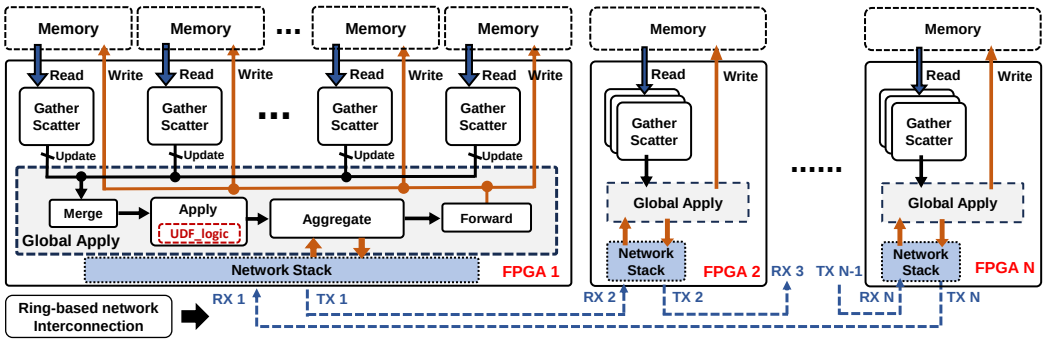


Fig. 6. Clementi architecture overview.

- **Gather-Scatter module:** The gather-scatter module retrieves edge lists and vertex properties from off-chip memory, processes them based on the edge-centric GAS model, and outputs updated vertex properties data streams. To maximize memory bandwidth utilization across multiple memory channels, Clementi uses multiple gather-scatter modules, with each FPGA memory channel assigned a dedicated gather-scatter module to perform scatter and gather operations concurrently. Details are in Section 5.1.
- **Global apply module:** The global apply module merges intermediate results from the gather-scatter modules, applies user-defined functions (e.g., for PageRank or BFS) to the merged results, and updates vertex properties in both local and remote FPGA memories via the network. Positioned between the gather-scatter modules and the network stack, it facilitates efficient data processing and transmission. Details are in Section 5.2.
- **Network stack module:** The network stack module works in conjunction with the global apply module to facilitate inter-FPGA communication. Details are in Section 5.3.

5.1 Gather-Scatter Module

The gather-scatter module performs graph processing tasks in edge-centric pattern. During the scatter operation, the edge list is sequentially loaded from off-chip memory. Based on the edge list values, the source vertex properties are accessed randomly from off-chip memory and assigned to the corresponding destination vertices for updates. To optimize small-granularity memory access, Clementi coalesces source vertex requests into sequential burst reads, loading them into cache for efficient access. A FIFO (First-in-First-Out) queue buffers destination vertices until the corresponding source properties are available. The FIFO depth is a hardware-dependent parameter; with a large value of depth, the pipeline can tolerate longer delays in memory access but requires more hardware resources, while a small value of depth reduces the resource consumption but may cause pipeline stalls. Once the source properties are fetched, the gather operation accumulates properties which share the same destination vertex. To handle the wide range of destination vertices, Clementi uses cascaded shuffle sub-modules to distribute destination vertices across multiple gather sub-modules based on vertex ranges. These sub-modules ensure efficient data routing and enable

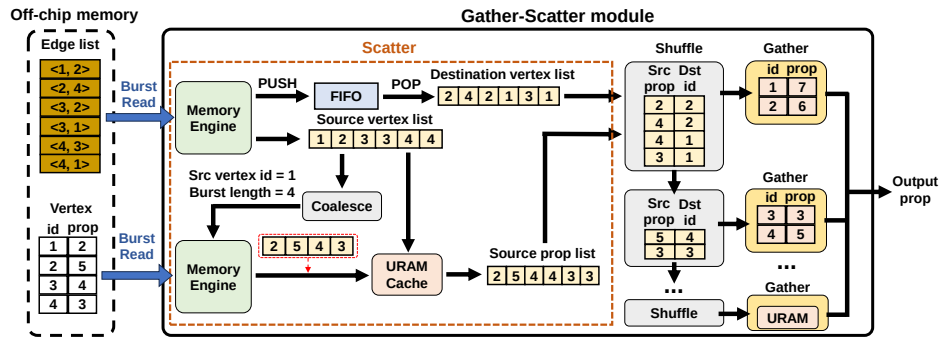


Fig. 7. Architecture of Gather-Scatter module.

parallel gather operations. Destination vertex property data is stored in on-chip URAM (Ultra RAM) to support frequent updates. After processing all edges, the updated destination vertices are written from URAM to the global apply module.

The architecture of the gather-scatter module is illustrated in Figure 7, which includes an example to demonstrate its functionality. As depicted, the scatter sub-module sequentially reads the edge list data from off-chip memory, dividing it into a source vertex list and a destination vertex list. For the source vertex list, the memory requests are coalesced to optimize access. Destination vertices are buffered in a FIFO queue until their corresponding source properties are retrieved, enabling the generation of the source vertex properties list. The shuffle sub-modules pair items from the source vertex properties list and the destination vertex list, distributing them to multiple gather sub-modules for parallel gather calculations.

The processing paradigm in the gather-scatter stage avoids data branching or inter-FPGA transmission, ensuring fully pipelined execution. In terms of implementation, Clementi ensures that the on-chip data processing pipeline (comprising the shuffle and gather sub-modules) maintains an overall throughput greater than that of the scatter sub-module, which is constrained by off-chip data access. As a result, the bottleneck in the gather-scatter stage shifts to off-chip memory access. Specifically, denote the bandwidth of the off-chip memory channel as BW, the size of edge as S_e bytes, II_{gather} and $II_{shuffle}$ are Initial Intervals (II) of the gather sub-module and shuffle sub-module, and the operating frequency as f. Therefore, the number of sub-modules, denoted as N_{gather} and $N_{shuffle}$, are configured with the following equation.

$$N_{shuffle} = N_{gather} \geq \frac{BW * max\{II_{gather}, II_{shuffle}\}}{S_e * f} \tag{1}$$

Furthermore, assume S_{uram} represents the total memory capacity of the URAMs in each gather sub-module, and S_v denotes the size of vertex properties. Due to the limited URAM resources in FPGA, the destination vertex range R_{dest} in each subgraph can be determined by Equation 2.

$$R_{dest} \leq \frac{N_{gather} * S_{uram}}{S_v} \tag{2}$$

5.2 Global Apply Module

Following the remote apply pattern in Clementi, the global apply module consists of merge, apply, aggregate, and forward sub-modules. It first merges output results from multiple gather-scatter modules, then performs user-defined apply functions, and finally updates the calculated vertex properties to both remote and local FPGAs based on aggregate and forward sub-modules.

5.2.1 Merge and forward sub-modules. The merge sub-module merges all intermediate updates from each Gather-Scatter module before passing them to user-defined apply sub-module for calculating new vertex properties. Similarly, the forward sub-module duplicates the received new vertex properties across all memory channels to update properties. Both the merge and forward sub-modules are designed to operate with an Initiation Interval (II) of a single clock cycle, ensuring high throughput for on-chip data processing in the global apply stage.

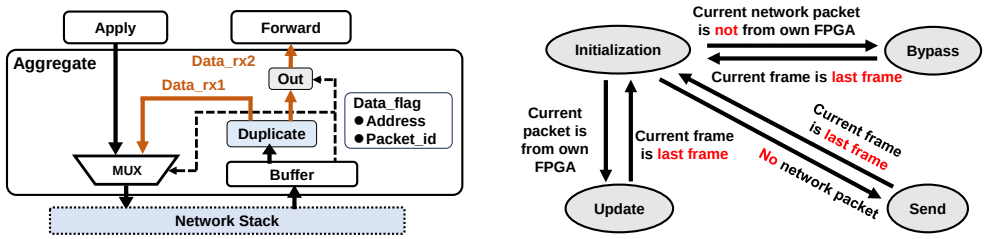
5.2.2 Apply sub-module. The apply sub-module featuring customized computing logic tailored for various applications such as PageRank (PR), Breadth-First Search (BFS), and Weakly Connected Components (WCC), as demonstrated in List 1. It first reads input data from the merge sub-module, calculate user-defined apply function for new vertex properties based on the old vertex properties and the output degree in the graph, and finally outputs the new vertex properties to the aggregate sub-module. In Clementi, the user-defined apply function is synthesized into hardware with a target Initiation Interval (II) of a single clock cycle using the `#pragma HLS pipeline II=1` directive. It is then integrated with the merge and aggregate sub-modules via predefined AXI (Advanced eXtensible Interface) stream interfaces. Note that the AXI Stream interface, is a common protocol in High-Level Synthesis (HLS) for data transfer, facilitating the transmission of control flags and payloads in a streaming manner. It eliminates the need for a separate handshake for each data transfer, instead utilizing a valid-ready handshake mechanism for efficient flow control.

```
void Apply (MergedStream, PropertyStream, DegreeStream, UpdateStream) {
    loop: while (!end) {
     #pragma HLS pipeline II = 1
        DATATYPE NewProperty = Read_from_stream (MergedStream);
        DATATYPE OldProperty = Read_from_stream (PropertyStream);
        DATATYPE OutputDegree = Read_from_stream (DegreeStream);

        DATATYPE UpdateProperty = UDF_logic (OldProperty, NewProperty, OutputDegree);
        Write_to_stream (UpdateStream, UpdateProperty);

        ... /* other built-in logic */
    }
}

inline DATATYPE UDF_logic (OldProp, NewProp, OutDeg) {
    DATATYPE UpdateProp;
    /* Customized user logic */
    UpdateProp = NewProp * 0.85 / OutDeg; /* PR */
    UpdateProp = if(IsActive(NewProp)) ? NewProp: OldProp; /* BFS */
    UpdateProp = NewProp; /* WCC */
    return UpdateProp;
}
```

Listing 1. User-defined apply function.

Implementing a new algorithm requires modifying only the user-defined logic, with no changes needed for other components. Specifically, users define the `UDF_logic` function in C/C++ and synthesize it using high-level synthesis (HLS) tools. The restrictions on this user-defined code arise from limitations of HLS tools, requiring that the function be stateless, deterministic, and free of unsupported features such as dynamic arrays.

5.2.3 Aggregate sub-module. As a part of the global apply module, the aggregate sub-module serves two key functions: (1) it receives incoming network data packets from other FPGAs and processes them based on the data flag within each packet, and (2) transmits calculated results from its own apply sub-module to other FPGAs. The architecture of the aggregate sub-module is illustrated in Figure 8(a). To optimize network bandwidth utilization, the aggregate sub-module aggregates multiple data frames into Maximum Transmission Unit (MTU)-sized data packet.



(a) Architecture of Aggregate sub-module (b) State-machine of Aggregate sub-module

Fig. 8. Aggregate sub-module.

During the runtime, the first data frame in each data packet contains data flags that control the functionality of the aggregate sub-module for the subsequent data frames. These data flags include `Address` and `Packet_id`, which are encoded within specific bits of the data frame. The `Address` specifies the off-chip memory location for storing the data, while the `Packet_id` is used to determine whether the data packet originated from the current FPGA or another FPGA. Within the aggregate sub-module, a multiplexer (MUX) determines the network output destination based on the `Packet_id` of the incoming data packet. If the `Packet_id` indicates that the data originated from another FPGA, the received data is forwarded to the downstream FPGA through the network. Conversely, if the data has completed a full loop through all FPGAs, it no longer needs to be transmitted over the network. In this case, the aggregate sub-module selects the output of the apply sub-module to be transmitted to other FPGAs through the network.

For detailed implementation, the aggregate sub-module adopts a hardware-friendly state-machine design, to process data without external triggers, maintaining high throughput with a single clock cycle Initial Interval (II). Figure 8(b) shows the state transitions, with detailed explanations.

- **Initialization:** Checks for the presence of a network data packet from the network stack. If a packet is detected, the aggregate sub-module retrieves the first frame, decodes the `Address` and `Packet_id`, and transitions to either the Bypass or Update state based on the `Packet_id`. If no network data packets are available, it transitions to the Send state.
- **Send**: Broadcasts local calculated data from the apply sub-module to update the vertex properties in other FPGAs.
- **Update**: Receives subsequent network data frames and outputs it to forward sub-module to update local vertex properties based on `Address` in the first frame.
- **Bypass**: Receives the updated vertex properties from the network then bypasses it to the next FPGA, meanwhile outputs it to forward sub-module to update local vertex properties.

When the last frame in the current data packet is detected (based on AXI-stream protocol), Send, Update, and Bypass all return to the state of Initialization.

5.2.4 Pipeline execution and bottleneck analysis. In Clementi, after completing the user-defined apply function for the new vertex properties, the global apply module broadcasts the calculated results to other FPGAs while simultaneously updating the properties locally. It integrates the merge, apply, aggregate, and forward sub-modules, each implemented with a single clock-cycle initial interval (II), enabling a fine-grained pipelined execution. Figure 9 illustrates a simplified example of vertex property updating across two FPGAs, demonstrating how the global apply pipeline in each FPGA operates in a fine-grained manner. This approach ensures that network latency is confined to the transmission of the first data frame, with subsequent latency mitigated through continuous processing, thereby maintaining high overall processing efficiency.

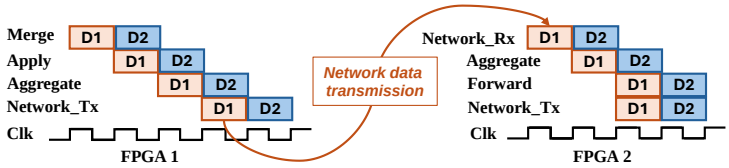


Fig. 9. Fine-grained pipeline of the global apply stage with single clock-cycle initial interval (II).

To conclude, the global apply stage involves three types of I/O: on-chip data processing, off-chip memory access, and network data transmission. Under the detailed implementation conditions, with a frequency of 250 MHz and a data width of 512 bits, both the on-chip data processing throughput and DDR4 memory bandwidth (around 16 GB/s) exceed the network bandwidth (100 Gbps). The aggregate sub-module in the global apply module interacts with the network stack, which operates at the lowest bandwidth among the three I/O types. Moreover, the aggregate sub-module processes the largest data volume as it sends and receives the entire vertex data in total. Hence, the bottleneck of the global apply stage is the execution of aggregate sub-module, due to limited network bandwidth and the large volume of transmitted data.

5.3 Network Stack Module

To provide better network compatibility, we adopt an open-sourced network stack in our design [52]. A reliable network environment is assumed, typical of data center scenarios where network transmissions are free from packet loss, ensuring uninterrupted pipeline execution in the global-apply stage with consistent throughput. Based on our experimental profiling results, the streaming data transmission achieves nearly 100 Gbps with network packet sizes set to MTU of approximately 1500 bytes, and each data frame's size at 64 bytes. Although the first data frame in Clementi is reserved for packet metadata, the effective data accounts for up to 95% of the total transmission. This network stack can be configured via registers to target different IP addresses during runtime.

6 Clementi Runtime

6.1 Performance Model

To accurately modeling the behaviors of the gather-scatter and global apply pipelines, we first define the relevant parameters. Let E represent the number of edges in the subgraph, and denote M_v and M_e as the number of memory access operations for vertices and edges, respectively. The data width of the memory port is defined as W_M bits, typically set to 512 bits. We further define C_{bw1} and C_{bw2} as the memory access bandwidths for edges and vertices, respectively, as they follow different access patterns (detailed in Section 5.1).

6.1.1 Gather-scatter stage. Equation 1 demonstrates that the total processing throughput of multiple shuffle and gather sub-modules is equal to or greater than the maximum memory bandwidth achievable in the scatter stage. This allows the downstream sub-modules to process the vertex properties and edge lists as soon as they are read. However, the irregular distribution of destination vertices leads to a significant number of edges being shuffled to the same gather sub-module, while leaving other sub-modules underutilized. As a result, this imbalanced distribution among gather sub-modules prolongs the total execution time of the gather-scatter stage, further compounded by an additional scaling factor due to data irregularity. Based on these, we can determine the execution time of the gather-scatter stage T_{gs} for each subgraph.

$$T_{gs} = s \cdot \left(\frac{M_e \cdot W_M}{C_{bw1}} + \frac{M_v \cdot W_M}{C_{bw2}}\right) \tag{3}$$

where s is the scaling factor that $s \geq 1$. According to the edge-centric execution model, edge lists are accessed sequentially in gather-scatter module. We assume each edge has two vertices, the data width of the vertex is W_v, usually 32 bits as an int type. Hence, the M_e equals to $\frac{2EW_v}{W_M}$. Then we could obtain the traversed edges per second (TEPS):

$$TEPS = \frac{E}{T_{gs}} = \frac{1}{s} \cdot \frac{1}{\frac{2W_v}{C_{bw1}} + \frac{2W_v}{C_{bw2}} \cdot \frac{M_v}{M_e}} \tag{4}$$

Given that the C_{bw1} is a constant value due to the sequential access of edge, while the C_{bw2} varies with the distribution of the source vertex. Furthermore, taking into account the peak performance achievable by a saturated FPGA pipeline, we define the C_1 as the peak throughput by a gather-scatter pipeline, the above equation can be formulated as follows.

$$TEPS = \min(C_1, \beta \cdot \frac{1}{C_2 + \alpha \cdot \frac{M_v}{M_e}}) \tag{5}$$

where $\alpha = \frac{2W_v}{C_{bw2}}$ and $\beta = \frac{1}{s}$ vary with the distribution of source and destination vertex in different subgraphs, respectively. The constant values of C_1 and $C_2 = \frac{2W_v}{C_{bw1}}$ are determined based on the actual hardware performance of gather-scatter module.

6.1.2 Global apply stage. As discussed in Section 5.2.4, the global apply stage holds a fine-grained pipeline with nearly constant throughput, indicating predictable time consumption to update all destination vertex in each subgraph. Define p as the number of partitioned subgraphs and C_3 as the constant time consumption under the fixed number of destination vertex in the subgraph, and we have the total execution time of the global apply stage:

$$T_{gapply} = p \cdot C_3 \tag{6}$$

To summarize, we give the total execution time T_{all} of graph processing tasks on each FPGA:

$$T_{all} = T_{gs} + T_{gapply} + T_{const} \tag{7}$$

where the T_{const} represents the constant overhead in graph processing, including partition switching and start-up overhead in the pipeline. In practice, We first determine the values of the parameters C_1,C_2,C_3 and T_{const}, which are independent of the graph data distribution, based on real hardware execution. Then, based on the range of C_{bw2} from 2 GB/s to 16 GB/s on DDR4 memory, α ranges from $[0.5, 4]$. Similarly, β ranges from $[0.25, 1]$, where $\beta = 1$ indicates no conflicts in the shuffle stage, and $\beta = 0.25$ represents complete conflicts with all destination vertices having the same value. Clementi sets $\alpha = 2$ and $\beta = 0.75$ as initial values, and then refines them for improved accuracy. As demonstrated in previous studies [1, 44], a sampled subset of subgraphs is used to estimate the values of α and β. Specifically, the gather-scatter stage is executed on a randomly selected subset of subgraphs using a single FPGA. The execution time from this stage is used to fit α and β, which are then applied globally to all subgraphs. This sampling-based approach provides execution time estimation without the need to process all subgraphs.

6.2 Balanced Workload Scheduling

The workload scheduling method aims to distribute subgraphs across N FPGAs, ensuring a balanced workload among FPGAs and meanwhile avoiding communication conflict by a rescheduling mechanism. The output is a workload execution list for each FPGA. Below are the key concepts:

- **Scheduling**: This step assigns the whole subgraph execution workloads (including gather-scatter stages and global apply stages) to each FPGA for balanced execution times.

- **Conflicts**: Conflicts occur when multiple FPGAs attempt to transmit or receive data simultaneously in their global apply stages. In the ring topology, only one data packet can be transmitted at a time, requiring conflict management.
- **Rescheduling**: Rescheduling shifts subgraph executions to new time slots when conflicts are detected. If a conflict occurs, the task is either swapped with another on the same FPGA or postponed to avoid contention.
- **Waiting Period:** The waiting period refers to the idle time when an FPGA has completed its current gather-scatter stage in subgraph execution but cannot proceed to global apply stage due to conflicts from other FPGAs, this is shown in Figure 5.

Following the Equation 3 and 6, we get the execution time predictions of the gather-scatter and global apply stage for each partitioned subgraph, respectively. Each execution of the gather-scatter stage is consistently followed by a global apply stage. Based on these, our workload scheduling method is divided into two main parts: naive workload scheduling and workload rescheduling to optimize the waiting period, shown in Algorithm 2. The naive workload scheduling approach aims to minimize the maximum execution time among FPGAs, a goal commonly referred to as makespan minimization [41]. To achieve this, we utilize a widely adopted greedy-based approach for workload balancing. It processes input subgraphs in descending order of their estimated execution times and assigns each subgraph to the FPGA whose workload currently has the smallest total execution time, ensuring a balanced distribution of execution time across all FPGAs. The rescheduling process rearranges workloads within each FPGA's workload list by interleaving global apply stages across FPGAs. For each workload in the list, we first check for data conflicts to determine if it needs to be rescheduled to a different execution time. If a conflict is detected, the `Choose` operation enumerates all workloads assigned to the same FPGA to find a workload that does not conflict with the current scheduling plan. If the `Choose` operation fails to find a conflict-free workload, our method introduces a waiting period caused by conflicts, referred to as a `Dummy`.

While the optimal solution to workload scheduling could be achieved by exhaustively evaluating all possible workload permutations, this approach has an $O(n!)$ complexity. Therefore, we opt for a practical and efficient solution that only considers task swapping within the workload list of a single FPGA to reduce the scheduling cost. This may involve sub-optimal scheduling while the experimental results remain highly efficient in practice.

Algorithm 2 Workload Scheduling Method

Input: N: number of FPGAs; $W = \{W_1, W_2, \ldots W_m\}$: input partition workloads ;
Output: $SW = \{SW_1, SW_2, \ldots SW_N\}$: Scheduled Workload list;

Set each $SW_i = \emptyset, i = 1, 2, 3, ..., N$;
$SW = GreedyAlgorithm(W, N)$ — Stage.1. Makespan minimization
for each $SW_j \in SW$: — Stage.2. Rescheduling workload
  for each $W_i \in SW_j$:
    while (HaveConflict(W_i) == True): /* Select a workload in same workload list without conflict */
      W_{temp} = Choose(SW_j);
      if ($W_{temp} \neq \emptyset$) **then**:
        Swap(W_{temp}, W_i);
      else
        SW_j.append($Dummy$); /* Waiting */
  end for
end for
Return $SW = \{SW_1, SW_2, \ldots SW_N\}$;

6.3 Input-Aware Partitioning

When executing a subgraph on a single FPGA, input-aware partitioning generates finer-grained edge list blocks to ensure balanced workloads across multiple gather-scatter sub-modules in a 2D partition manner. As indicated by the Equation 3, the aim of input-aware partitioning is to minimize the values of $\frac{M_v}{M_e}$ while maintaining a balanced edge number across all blocks.

The partition process begins with grouping the original graph G into multiple *Groups* based on the read burst length in the gather-scatter module. This ensures that a single burst read operation covers all source vertex accesses within a *Group item* through the vertex coalescing. Next, a greedy allocation method is employed to assign the most edge-heavy items to destination sets with the fewest existing edges. This allocation step aims to maintain an even edge distribution across groups to avoid load imbalance. Given the skewed data distribution typical of real-world graphs, we then apply an iterative refinement process. In each iteration, the *Group item* with the largest edge count is divided into smaller *items* to further balance the edge distribution. The execution time of each generated edge list block is estimated during every iteration based on the performance model to ensure load balancing across all blocks. While a finer granularity of *items* can offer a better balanced execution, it inadequately increases the memory access operations due to redundant data access in the generated *items*. To prevent over-partitioning, the iterative refinement process terminates upon reaching a local minimum in the total estimated execution time.

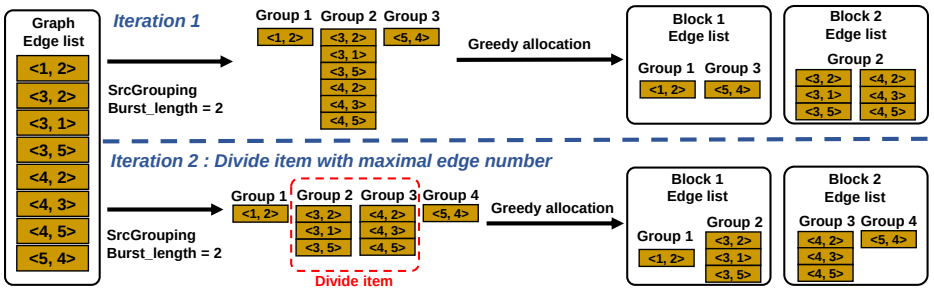


Fig. 10. Example of input-aware partition method.

Figure 10 illustrates an example of the input-aware partition method. The original input edge list is initially grouped based on the source vertex range and then allocated into different blocks. The group with the maximum number of edges undergoes division using an iterative partitioning strategy to achieve balanced edge numbers and $\frac{M_v}{M_e}$ across generated blocks.

7 Experimental Results

7.1 Experimental Setting

7.1.1 Hardware platforms. We deploy Clementi on publicly accessible HACC [2] cluster, which consists of six AMD Alveo U250 [50] FPGAs, each managed by a virtual CPU and connected to network switches via the 100 Gbps FPGA-side network stack. We use Vitis HLS [49] toolchain 2021.2. Additionally, we employ OpenMPI [35] 4.1.4 for control across multiple virtual CPUs.

7.1.2 Implementation settings. We set the interval size to 1 million vertices in the initial interval-shard graph partition. Additionally, we configure each gather-scatter module with a read burst length of 512, a FIFO depth of 512, and 16 gather and 16 shuffle sub-modules.

7.1.3 Graph datasets. The graph datasets used in the experiments are detailed in Table 1, ranging from synthetic to real-world graphs with various scales and distributions. The data types in all graphs are set to 32-bit integers.

Table 1. Graph datasets in our experiment

Dataset Name	|V|	|E|	Categories
enwiki-2013 (WK) [4]	4.2 M	101.4 M	Web
rmat-25-16 (R25) [25]	33.6 M	536.87 M	Synthetic
graph500-scale24-ef16 (G24) [37]	8.9 M	520.5 M	Synthetic
graph500-scale25-ef16 (G25) [37]	17.0 M	1.0 B	Synthetic
Twitter-2010 (TW) [24]	41.6 M	1.47 B	Social
gsh-2015-host (GSH) [5]	68.7 M	1.80 B	Web
Friendster (FR) [26]	65.6 M	1.81 B	Social
SK-2005 (SK) [5]	50.6 M	1.95 B	Web
UK2007-05 (UK) [5]	105.9 M	3.74 B	Web
Kron-28-32 (K28) [25]	268.4 M	8.59 B	Synthetic

7.2 Resource Utilization and Frequency

We evaluate the resource utilization and maximum frequency of Clementi on AMD U250 [50] with three commonly-used graph processing algorithms: Page Rank (PR), Breadth-First Search (BFS), and Weakly Connected Components (WCC), shown in Table 2.

Table 2. Resource utilization and frequency on U250 [50] FPGA

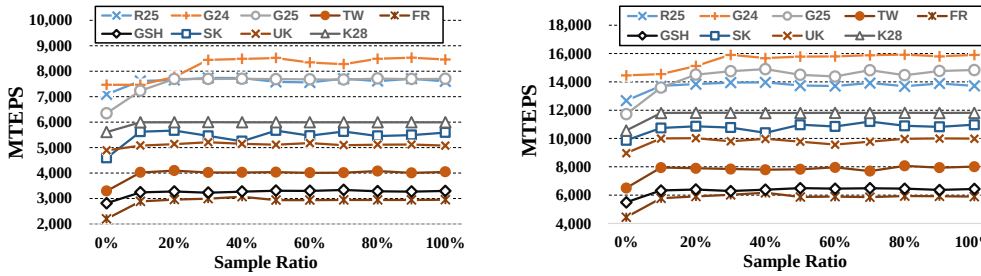
Resource Type	PageRank	BFS	WCC
CLB	67.92%	66.86%	66.77%
BRAM	40.87%	39.29%	39.29%
URAM	60.70%	60.70%	60.70%
DSP	0.40%	0.14%	0.14%
Frequency (MHz)	250.0	250.0	250.0

7.3 Performance Model Evaluation

7.3.1 Accuracy of performance model. Our performance model is designed to capture the key bottlenecks in graph processing pipelines, such as off-chip memory access in the gather-scatter stage and network communication cost in the global-apply stage. While we omit certain factors, such as on-chip data processing overhead and on-chip memory access latency, these factors contribute significantly less to performance variability compared to the major bottlenecks. For detailed implementation, we determine all the constant parameters in Equation 7 by utilizing the hardware execution times on single FPGA from randomly selected subgraphs. Subsequently, we fit corresponding α and β to predict the execution time on each dataset. To evaluate the accuracy of our performance model, we calculate the subgraph error ratio, defined as the average discrepancy between the predicted execution times and the actual hardware execution times for all subgraphs within a dataset. As shown in Table 3, under the optimal alpha and beta settings, the subgraph error ratios are predominantly below 5%, demonstrating that it is sufficiently accurate for making reliable workload scheduling decisions in practice.

Table 3. Results for performance model

Dataset	R25	G24	G25	TW	FR	GSH	SK	UK	K28
α	1.57	2.02	1.92	1.33	1.95	1.14	2.65	1.36	1.63
β	1.00	0.76	1.00	0.72	1.00	0.61	0.61	0.61	0.98
Subgrph Error Ratio (%)	2.22	1.57	2.59	0.81	3.28	2.21	6.97	5.00	0.29

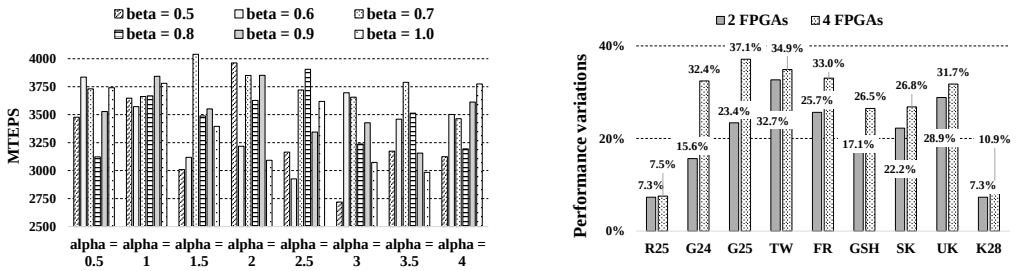


(a) Performance variation on two FPGAs (b) Performance variation on four FPGAs

Fig. 11. Sensitivity study of α and β in Clementi's approach.

7.3.2 Sensitivity study for α and β. In this part, we discuss the impact of α and β on the end-to-end hardware execution performance. We first analyze the sample ratio factor in Clementi, as α and β are fitted based on real hardware execution data from sampled subgraphs. Figure 11 illustrates the results: As the sample ratio increases from 0% to 100%, performance improves due to a more accurate estimation of execution time, eventually converging to the best-case scenario in multiple datasets. Figure 11(a) and Figure 11(b) presents the performance under two and four FPGAs, respectively. When the sample ratio exceeds 30%, the performance stably approaches the peak, demonstrating that this sampling-based method is both practical and effective.

Furthermore, we conduct an extended sensitivity analysis by varying the configuration parameters α and β over a broader range using manually assigned values. Specifically, α is varied from 0.5 to 4.0 with an interval of 0.5, and β is varied from 0.5 to 1.0 with an interval of 0.1. Figure 12(a) presents the end-to-end performance variation on two FPGAs using the TW dataset. The measured throughput ranges from 2720 MTEPS to 4045 MTEPS, resulting in a 32.7% variation. Figure 12(b) further illustrates the performance variability across all evaluated datasets under various combinations of α and β on both two-FPGA and four-FPGA configurations. The results show that performance fluctuates between 7.3% and 32.7% on two FPGAs, and between 7.5% and 37.1% on four FPGAs. These findings indicate that manually assigned values for α and β yield significant performance variations due to insufficient consideration of graph data distribution, thereby highlighting the effectiveness of the sampling-based method in Clementi.



(a) Performance variation on two FPGAs on TW dataset (b) Performance variation on two and four FPGAs

Fig. 12. Sensitivity study for manually specified α and β values.

7.4 Input-aware Partition Time Cost

We conducted input-aware partitioning tests on the AMD7V13 CPU, measuring two key components: the time required to load the entire graph from the hard disk into CPU memory and the time

Table 4. Graph Load & Partition time cost

Time(min)	Dataset								
	R25	G24	G25	TW	FR	GSH	SK	UK	K28
Load	3.52	3.18	6.83	9.77	11.78	12.7	22.57	18.13	59.98
Partition	2.52	2.65	4.50	13.45	38.62	5.00	14.38	13.82	58.55

taken for input-aware partitioning, as shown in Table 4. While the loading and partitioning times are longer than the graph processing time, it is important to note that these represent one-time costs. Once the graph is loaded and partitioned, the same partitions can be reused for multiple processing runs, which is typical in large-scale graph processing systems, effectively amortizing the initial setup cost.

7.5 Overall Performance Evaluation

We employ PR, BFS, WCC graph processing algorithms on Clementi with 1, 2, 4, and 6 FPGAs. The overall performance is shown in Table 5.

Table 5. Performance on large-scale graph datasets

Algo.	# of	Performance (MTEPS)								
	FPGAs	R25	G24	G25	TW	FR	GSH	SK	UK	K28
PR	1	3936	4345	4149	2043	1493	1658	2858	2544	**OOM**
	2	7601	8459	7702	4045	2951	3296	5589	5075	5995
	4	13722	15901	14834	8012	5893	6436	10971	9991	11812
	6	15780	19081	18712	10518	8476	9522	15416	11787	17378
BFS	1	3948	4457	4308	2231	1519	1819	3078	3352	**OOM**
	2	7420	8792	8313	4354	2980	3619	5709	6611	5941
	4	13377	16402	15267	8450	5870	6988	11290	10973	11779
	6	15484	18638	19706	10128	8466	9223	15792	11153	17416
WCC	1	3965	4411	4454	2252	1520	1834	2978	3381	**OOM**
	2	7712	8644	8206	4342	2997	3630	5701	6641	5956
	4	14085	16198	15178	8413	5937	7071	11043	11094	11762
	6	15471	18400	19342	10268	8420	9272	15653	11762	17354

We measure throughput in millions of traversed edges per second (MTEPS). As shown in Table 5, the results indicate that as the number of FPGAs in Clementi increases, the performance correspondingly increases, reaching a peak of 19.7 GTEPS on 6 FPGAs with the BFS application. However, Clementi's performance on real-world graphs displays a decline compared to synthetic graphs. This discrepancy can be attributed to the higher $\frac{M_v}{M_e}$ (vertex memory access operations constitute a larger proportion relative to edge memory access operations, defined in Section. 6.1) in real graphs, resulting in reduced processing efficiency according to the Equation 4. Taking the FR dataset as an example, its average degree is 2.13 × higher than that of G25, suggesting a higher $\frac{M_v}{M_e}$ during graph processing. This increase in the vertex-to-edge ratio results in more frequent vertex data accesses, which in turn leads to higher memory access overhead in edge-centric graph processing pattern, thus causing declined performance.

7.6 Scalability Evaluation

For scalability evaluation, we run PageRank on 1 to 6 AMD U250 [50] FPGAs, using gigabit traversed edges per second (GTEPS) as the performance metric. As illustrated in Figure 13, Clementi achieves near-linear scalability, with a 3.42x–3.88x throughput improvement under a 4-FPGA configuration and a 4.0x–5.74x throughput improvement under a 6-FPGA configuration. This scalability validates the effectiveness of Clementi's architecture-oriented performance model and workload scheduling

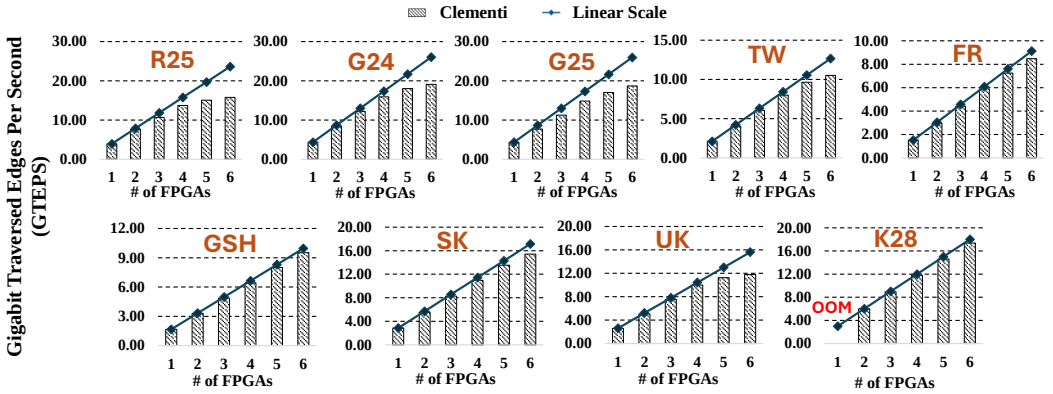


Fig. 13. Scalability evaluation of PageRank algorithm on various datasets in Clementi (Higher is better).

method, which balance execution across FPGAs while minimizing communication overhead. The deviation from ideal linear scalability can be attributed to two primary factors. First, for smaller datasets such as G24 and G25, which contain only 8 and 17 subgraphs respectively, the limited number of subgraphs results in imbalanced workload distribution across 4 to 6 FPGAs. Second, in graphs with high power-law distributions, such as UK, a small number of subgraphs with high-degree nodes dominate the execution time, leading to load imbalances and reducing overall scalability. Note that the out-of-memory (OOM) in K28 dataset occurs because its large data volume exceeds the memory capacity of a single FPGA.

7.7 Performance Comparison with SOTA

We compare our Clementi with SOTA FPGA-based designs Foregraph [13] and GraVF-M [14], as well as open-sourced baselines: CPU-based solution Gemini [57] and GPU-based solution Lux [21]. Table 6 shows the platform specifications of evaluated designs.

Table 6. Platform specifications for target designs

Design	Type	Platform	Litho	Memory Bandwidth	Interconnect type
ForeGraph [13]	Multi-FPGA	XCVU190	20 nm	19.2 GB/s	Network
GraVF-M [14]	Multi-FPGA	KU060	20 nm	30 GB/s	PICe
Gemini [57]	Multi-CPU	XeonE5-2670v3	22 nm	68 GB/s	Network
Gemini [57]	Multi-CPU	AMD7V13	7 nm	230 GB/s	Network
Lux [21]	Multi-GPU	NVIDIA V100	12 nm	898 GB/s	PCIe
Clementi	Multi-FPGA	Xilinx U250	16 nm	77 GB/s	Network

7.7.1 Throughput comparison with FPGA designs. Note that ForeGraph [13] is not publicly available, we prototype its performance based on details from their paper. Specifically, we derive results from both their reported data and our simulation using a 12.25 Gbps network stack, referred to as ForeGraph-sim. Table 7 presents a performance comparison between Clementi and ForeGraph, using 4 FPGAs for both systems, across the PR, WCC, and BFS applications. The results demonstrate that Clementi achieves between 2× and 8.75× performance improvement compared to ForeGraph and ForeGraph-sim. While Clementi benefits from a 4× improvement in hardware bandwidth (76.8 GB/s DRAM bandwidth vs 19.2 GB/s DRAM in ForeGraph and 12.25 Gbps network bandwidth), it achieves more than 4× overall performance improvement, peaking at 8.75× for datasets such

Table 7. Throughput comparison with ForeGraph [13]

Designs	App.	Datasets	Throughput (MTEPS)	**Clementi (MTEPS)**	**Speedup**
ForeGraph [13] (Results from paper)	PR	TW	1856	8012	**4.32x**
	WCC		1727	8450	**4.87x**
	BFS		1458	8413	**5.77x**
ForeGraph-sim	PR	G25	2209	14834	**6.72x**
	WCC		2055	15267	**7.43x**
	BFS		1735	15178	**8.75x**
	PR	TW	1856	8012	**4.32x**
	WCC		1727	8450	**4.89x**
	BFS		1458	8413	**5.77x**
	PR	UK	2264	9991	**4.41x**
	WCC		2107	10973	**5.21x**
	BFS		1779	11094	**6.24x**
	PR	FR	2728	5893	**2.16x**
	WCC		2539	5870	**2.31x**
	BFS		2143	5937	**2.77x**

as TW, G25 and UK. This demonstrates that the performance gain is attributable not only to the platform but also to the superior hardware-software co-design of Clementi.

Compared to GraVF-M [14], which uses a PCIe Gen3x8 interconnection for 4 FPGAs with a bandwidth of 64 Gb/s and a memory bandwidth of 30 GB/s per FPGA, we configure Clementi with 4 FPGAs to ensure a fair comparison. Using RMAT graph datasets and following the same uniform distribution and degree variation settings as the GraVF-M paper, Table 8 shows that Clementi achieves a performance improvement ranging from 2.21× to 3.85×. The most notable gains occur when the average degree of the graph exceeds 4, which is common in widely used graph datasets. Although having approximately 2.5× hardware bandwidth compared to GraVF-M, Clementi achieves more than 2.5× the performance improvement, peaking at 3.85×. This underscores that our design plays a critical role in improved performance beyond just the hardware capabilities, especially in handling high-degree graph workloads.

Table 8. Throughput (MTEPS) comparison with GraVF-M

Degree of RMAT [25]	**2**	**4**	**8**	**16**	**32**	**64**
GraVF-M [14]	652	856	1280	2083	3568	4623
Clementi	1443	2519	4204	6172	13722	15430
SpeedUp	**2.21x**	**2.94x**	**3.28x**	**2.96x**	**3.85x**	**3.34x**

7.7.2 Throughput comparison with CPU/GPU designs. To evaluate the performance of Clementi against state-of-the-art designs, we first compared it with the CPU-based Gemini system [57], using data from the original publication. This comparison is justified by the comparable memory bandwidths of the CPU used in Gemini (68 GB/s) and the U250 FPGA (77 GB/s), as well as the identical 100 Gbps network configurations used in both systems. We selected PageRank as the benchmark algorithm, as it provides a fair comparison for both edge-centric and vertex-centric graph processing systems. We measure throughput in MTEPS (Million Traversed Edges Per Second) across various node configurations. Due to the unavailability of clusters with 8 U250 FPGAs, we present results for configurations with a maximum of 6 FPGAs in HACC [2], detailed in Table 9.

The results show that for the smaller WK dataset, Clementi consistently outperforms Gemini [57] in both scalability and overall performance. For the TW dataset, Clementi achieves comparable performance to Gemini while demonstrating superior scalability. The TW dataset, which contains a high concentration of high-degree vertices within a limited vertex range, clusters these vertices on

Table 9. Throughput (MTEPS) comparison with Gemini [57] (Results collected from paper) on PageRank

Number of nodes (CPU/FPGA)	WK		TW		UK	
	Gemini	Clementi	Gemini	Clementi	Gemini	Clementi
1	4188	2646	2304	2043	7430	2544
2	3811	4736	4191	4045	14466	5075
4	3560	7088	7423	8012	25648	9991
6	-	8108	-	10518	-	11787
8	4188	-	9724	-	50523	-

a single node in multi-node setups. This clustering leads to an increase in communication overhead in Gemini, which lacks efficient communication optimization. Conversely, for the UK dataset, characterized by edge concentration around adjacent vertices, Gemini's performance surpasses our system due to its low Last Level Cache (LLC) miss ratio, which is well suited to the CPU architecture. Additionally, Gemini benefits from reduced data communication volumes, enhancing its scalability for this dataset. Overall, Clementi still demonstrates a strong scalability in handling various graph data distributions.

To ensure fairness in the comparison, we re-evaluated the multi-CPU based Gemini [57] and the multi-GPU based Lux [21] using current hardware: AMD 7V13 CPUs and V100 GPUs, providing a more up-to-date reflection of their performance. We used up to three AMD 7V13 CPUs, each with 230 GB/s memory bandwidth, all connected through a 100 Gbps network. For the GPU-based system, we utilized a single NVIDIA Tesla V100 SXM2 machine, which consists of four GPUs interconnected by PCI-E 3.0 x16 interfaces with a bandwidth of 16 GB/s. The results of these evaluations are summarized in Table 10. Note that the performance results for Clementi are provided separately in Table 5.

Table 10. Throughput (MTEPS) for Gemini [57] (on AMD CPUs) and Lux [21] (on V100 GPUs) on PageRank

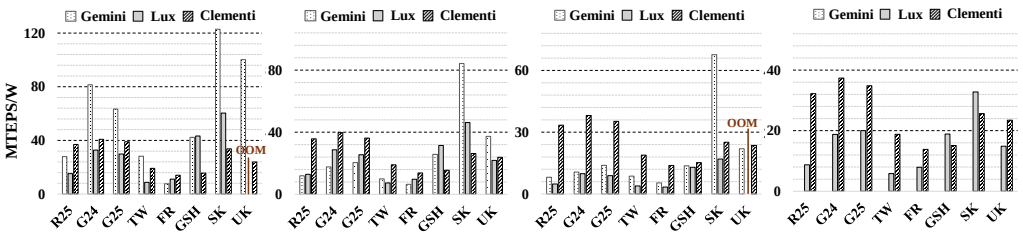
Design	# of nodes	Throughput (MTEPS)							
		R25	G24	G25	TW	FR	GSH	SK	UK
Gemini [57]	1	4615	13450	10459	4660	1260	6974	20271	16535
	2	3937	5820	6731	3291	2142	8488	27814	12316
	3	4028	5272	6885	4302	2666	8765	33646	10874
Lux [21]	1	3835	8236	7489	2198	2787	10868	15155	**OOM**
	2	6457	14419	12799	3689	4882	15778	23210	10978
	3	3669	7457	6741	2953	2547	9770	12797	**OOM**
	4	8762	18859	20095	5856	7970	18958	32934	14937

For the multi-CPU results, the upgrade in CPU memory bandwidth enhances Gemini's performance in single-node setting, delivering high throughput. In terms of bandwidth, a single CPU is comparable to three FPGAs (230 GB/s per CPU compared to 77 GB/s per U250 FPGA). Similarly to the phenomenon observed in the previous comparison with Gemini (data from the original paper), Gemini benefits from higher LLC hit rates due to the CPU cache system, such as SK and GSH. In contrast, Clementi excels in handling other datasets.

For multi-GPU comparison, the experimental results underscore Lux's ability to leverage substantial computational power and high memory bandwidth (898 GB/s), particularly with graph data characterized by dense edge concentrations, as observed in SK and GSH. However, despite operating with a significant bandwidth of 898 GB/s, Lux's performance does not meet the projected efficiency of Clementi, which uses only 77 GB/s. Considering that graph processing is predominantly memory-bound, this performance gap is significant, suggesting a notable underutilization of bandwidth by

Lux. Notably, in the configuration utilizing three GPUs, Lux exhibits a performance degradation attributable to resource under-utilization—only a single GPU is actively engaged in computation, while the remaining two remain idle. Consequently, the suboptimal resource allocation fails to deliver any performance improvement over the single-GPU baseline.

7.7.3 Energy efficiency comparison. We evaluate the energy efficiency of Clementi in comparison to the CPU-based Gemini [57] and GPU-based Lux [21], using AMD 7V13 CPUs and V100 GPUs, respectively. Power consumption was measured using `turbostat`[20] for the CPU, `nvidia-smi`[33] for the GPU and `xbutil`[51] for the FPGA. We define energy efficiency as the ratio of million traversed edges per second per watt (MTEPS/W), and test this metric across 1 to 4 nodes on various graph datasets, shown in Figure 14.



(a) Single-node setting (b) Two-node setting (c) Three-node setting (d) Four-node setting

Fig. 14. Energy efficiency comparison.

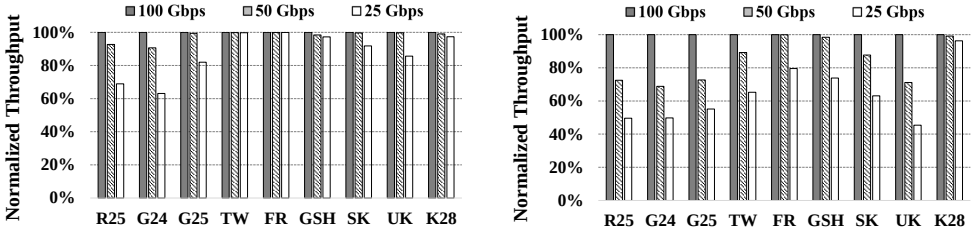
In Clementi, each FPGA card is attached with a CPU node for control, following the configuration in HACC [2] cluster. Hence, we combined the power consumption of the CPU and FPGA when calculating the energy efficiency of Clementi. Each FPGA board in Clementi peaks at 69.4W during graph processing, with the attached CPU consuming 37W. In contrast, the V100 GPU in Lux consumes approximately 250W, while the AMD 7V13 CPU in Gemini consumes around 165W for the same tasks. The results demonstrate that Clementi outperforms multi-CPU/GPU designs in energy efficiency, delivering an average 1.52× improvement in a two-node setting and an average 1.95× improvement in a four-node setting. This energy advantage arises from the design of customized fine-grained hardware pipelines for graph processing, where unnecessary components are eliminated, thereby minimizing energy consumption.

7.7.4 Cost effectiveness comparison. We collect publicly available pricing data for cloud devices, widely used in research due to their accessibility and scalability. On the Microsoft Azure cloud platform [32], the hourly prices are as follows: NCv3 series with NVIDIA V100 GPU at $3.00/hour, Eadsv5 series with AMD EPYC 7763 CPU (comparable to 7V13) at $1.31/hour, and NP-series with Xilinx U250 FPGA at $1.65/hour [31]. While GPUs offer strong raw performance, their high cost reduces performance-per-dollar compared to FPGAs. By leveraging the cost-efficiency of FPGAs, Clementi became a cost-effective solution for graph processing in cloud computing.

7.8 Network Bandwidth Sensitivity Evaluation

Network bandwidth is critical for Clementi's multi-FPGA communication, especially in large-scale deployments where high-speed data transfers are essential. Limited bandwidth increases communication overhead and potentially degrades overall performance. To evaluate its impact, we test normalized throughput under varying bandwidths (100 Gbps to 25 Gbps).

Figure 15 demonstrates that Clementi maintains high throughput under decreased inter-FPGA bandwidth conditions. Reducing the bandwidth from 100 Gbps to 50 Gbps results in throughput



(a) Two-FPGA setting (b) Four-FPGA setting

Fig. 15. Network sensitivity evaluation.

reductions of up to 9.3% on two FPGAs and 31.2% on four FPGAs. At 25 Gbps, throughput drops by up to 36.9% on two FPGAs and 54.6% on four FPGAs. This highlights the effectiveness of Clementi's communication-computation overlap, which tolerates the network bandwidth reductions.

7.9 Skewness Sensitivity Evaluation

To evaluate the impact of skewness in graph datasets, we conducted experiments using the synthetic RMAT [25] dataset R25, which features a uniform distribution and varying average degrees. We employed the PageRank algorithm to assess both performance and scalability. The experiments were performed on configurations of 2, 4, and 6 FPGAs.

Table 11. Throughput (MTEPS) with different average degree

Degree of RMAT [25]	**2**	**4**	**8**	**16**	**32**	**64**
2 FPGAs	935	1546	2153	3219	7728	8075
4 FPGAs	1443	2519	4204	6172	13722	15430
6 FPGAs	1740	2938	5316	7818	14575	15400

The results in Table 11 show that Clementi's performance varies with the degree distribution of the graph. For sparse graphs with very low edge-to-vertex ratios, Clementi's performance is constrained by increased loading overhead, as its edge-centric processing model requires loading a substantial portion of vertex properties along with the edge list. Additionally, duplicating the entire vertex set across each FPGA further exacerbates memory limitations, particularly for large-vertex datasets such as web-scale graphs. For high-degree graphs, which are common in real-world graphs such as social networks and web graphs, Clementi achieves better performance, peaking at 15.4 GTEPS. The limited scalability arises from the R25 dataset's 32 subgraphs, which are difficult to distribute evenly across four or more FPGAs, causing workload imbalance. In contrast, larger graphs with a greater number of subgraphs facilitate more balanced workload distribution, thereby improving scalability and overall performance with increased FPGA numbers.

7.10 Comparison between Remote-scatter and Remote-apply Patterns

This section compares the Remote-scatter (RS) and Remote-apply (RA) patterns, focusing on FPGA-side memory consumption and end-to-end performance. The RS pattern performs remote accesses to vertex data during the scatter stage, transmitting only vertex properties not present on the current FPGA while updating vertex data locally. In contrast, Clementi adopts RA pattern, synchronizing vertex updates during the apply stage, which can be overlapped to minimize communication cost.

The impact of vertex replication on FPGA-side memory is summarized in Table 12. The results indicate an average memory usage increase of 1.12×, 1.24×, and 1.37× under two-, four-, and six-FPGA configurations, respectively. This vertex replication limits the graph size that can be

Table 12. Memory consumption comparison

Design	# of nodes	Memory consumption under different datasets (Gbytes)								
		R25	G24	G25	TW	FR	GSH	SK	UK	K28
RS pattern	2-6‡	4.46	4.14	8.13	11.81	14.60	14.65	15.63	30.05	69.21
Clementi	2	5.37	4.45	8.74	13.09	16.60	16.58	17.22	33.31	77.31
	4	6.45	4.73	9.28	14.42	18.80	18.68	18.84	36.70	85.90
	6	7.52	5.02	9.82	15.75	21.00	20.78	20.46	40.09	94.49

‡ Memory consumption remains consistent regardless of the number of nodes.

managed by Clementi, as defined by the formulation: $(E + N \times V) < N \times M$, where the E is the memory consumption of edge and V is the memory consumption of vertex properties, M is the memory capacity per FPGA, and N is the number of FPGAs.

When performing the remote-scatter pattern, the limited on-chip memory capacity requires the vertex set to be loaded in chunks that fit within the available memory. For example, a single U250 FPGA's on-chip memory (including URAM and BRAM) can only store a maximum of 6.75M vertices, including both vertex indices and properties, which is significantly smaller than the large-scale graphs described in Table 1. Therefore, before edge processing, vertex data must be divided into chunks and loaded from off-chip memory. For vertex chunks not currently stored on the FPGA, remote vertex access is issued through the network stack.

To optimize remote vertex access, we set the chunk size to 1M vertices to improve bandwidth utilization of the network and off-chip memory when handling large data volumes. However, remote vertex access across two FPGAs still incurs a time cost 6.63× higher than direct network data transmission. This overhead is primarily caused by additional operations such as sending requests and performing off-chip memory accesses, which add complexity beyond the straightforward process of receiving responses during direct data transmission.

To further investigate the end-to-end performance differences between the RS and RA patterns, we conduct experiments under a two-FPGA setting. Figure 16 provides a breakdown of the execution time for the RS pattern, revealing that remote vertex access operations constitute an average of 61.7% of the total graph processing time, while the remaining time is allocated to loading the local edge list and performing on-chip gather and apply operations. Moreover, Figure 17 compares the end-to-end performance of the RS and RA patterns, showing that the RA pattern achieves an average performance improvement of 1.99× over the RS pattern. In conclusion, although the RA pattern in Clementi demands higher FPGA-side memory consumption due to vertex duplication, it delivers significantly superior performance compared to the RS pattern.

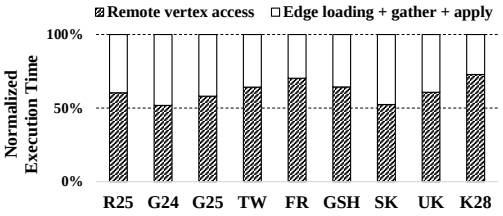


Fig. 16. Normalized execution time breakdown for RS pattern under two-FPGA setting.

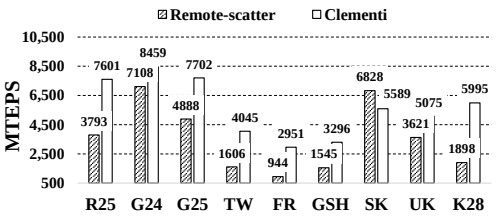


Fig. 17. Throughput comparison between RS and RA patterns under two-FPGA setting.

8 Future work

Despite the advancements presented in Clementi, several application scenarios still need further investigation. These include:

Memory consumption optimization: Vertex compression techniques, including advanced data encoding schemes and adaptive compression methods, will be explored in Clementi to reduce memory consumption and address memory capacity limitations.

Dynamic graph processing: Clementi is primarily designed for processing static graph structures. For dynamic graph, Clementi employs a batching approach for graph updates, which requires a high-cost repartitioning stage for the entire graph dataset. As part of our future work, we aim to explore techniques for incremental updates. The core idea is to develop a novel partitioning algorithm that recalculates only the affected subgraphs, thereby avoiding the need for full reprocessing. Furthermore, we plan to integrate Packed Memory Arrays to manage dynamic edge insertions.

9 Related Work

Current FPGA-based graph processing contains single FPGA-based and multiple FPGA-based designs. In 2016, Dai et al. proposed FPGP [12], which uses an interval-shard method to partition large graphs into sub-partitions that can be stored by on-chip RAMs to be processed by parallel processors. Shao proposed FabGraph [42], which uses a two-level vertex caching mechanism on FPGA-DRAM platforms to reduce the amounts of vertex data transmissions and pipeline stalls during the execution of graph algorithms using on-chip BRAM and URAM. MiKhil used a non-blocking cache design [3] to handle tens of thousands of outstanding read misses to increase the ability of the memory system. By coalescing accesses from multiple accelerators into fewer DRAM memory requests, they can achieve similar performance as the large capacity caches with lower cost. Chen et. al. proposed ThunderGP [11], optimizing the graph processing with a pipelined edge-centric GAS model. In 2022, Chen et. al. proposed ReGraph [10] that utilizes the high memory bandwidth of HBM with heterogeneous pipelines. GraphLily [19] provides a graph linear algebra overlay, to accelerate graph processing on HBM-equipped FPGAs.

For Multi-FPGA based designs, Dai et al. proposed ForeGraph [13], which uses vertex-centric interval-shard graph partition methods in the partition stage and extended the single FPGA-based framework into multi-FPGAs with the adoption of Microsoft Catapult [7]. In 2019, Nina et. al. proposed GraVF-M [14], in which multiple FPGAs connected through PCI-e and incorporated filtering delivery to reduce communication overhead, as well as a floating barrier to mitigate workload imbalance. Wu et. al. proposed FDGLib [48], a communication library that enables graph processing in data centers, also based on Microsoft Catapult [7].

10 Conclusion

In conclusion, this paper presents Clementi, an efficient multi-FPGA-based graph processing framework designed to address the poor scalability of the state-of-the-art systems. Clementi provides hardware/software co-design method to mitigate communication overhead and workload imbalance among FPGAs. Experimental results demonstrate that Clementi out-performs SOTA multi-FPGA designs by as much as 8.75×, achieving near-linear scalability.

Acknowledgments

This work is supported in part by the Ministry of Education AcRF Tier 2 grant, Singapore (MOE-T2EP20121-0016), and a Google South & Southeast Asia Research Award. We also thank the AMD Heterogeneous Accelerated Compute Clusters (HACC) program [2] for the generous hardware donation. Yao Chen from the National University of Singapore is the corresponding author.

References

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 716–727. doi:10.1109/SC.2016.60

[2] AMD Xilinx 2024. *Heterogeneous Accelerated Compute Cluster (HACC) at NUS.* https://xacchead.d2.comp.nus.edu.sg/.

[3] Mikhail Asiatici and Paolo Ienne. 2021. Large-scale graph processing on FPGAs with caches for thousands of simultaneous misses. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 609–622.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. 587–596.

[5] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[6] Erik G Boman, Karen Dragon Devine, and Sivasankaran Rajamanickam. 2014. *2D Partitioning for Scalable Matrix Computations on Scale-Free Graphs.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[7] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. doi:10.1109/MICRO.2016.7783710

[8] Linchuan Chen, Xin Huo, Bin Ren, Surabhi Jain, and Gagan Agrawal. 2015. Efficient and Simplified Parallel Graph Processing over CPU and MIC. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 819–828. doi:10.1109/IPDPS.2015.88

[9] 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)*. 67–73. doi:10.1109/FPL.2019.00020

[10] 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)*. 1342–1358. doi:10.1109/MICRO56248.2022.00092

[11] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-Based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 69–80. doi:10.1145/3431920.3439290

[12] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*. Association for Computing Machinery, New York, NY, USA, 105–110. doi:10.1145/2847263.2847339

[13] 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* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 217–226. doi:10.1145/3020078.3021739

[14] Nina Engelhardt and Hayden K.-H. So. 2019. GraVF-M: Graph Processing System Generation for Multi-FPGA Platforms. *ACM Trans. Reconfigurable Technol. Syst.* 12, 4, Article 21 (nov 2019), 28 pages. doi:10.1145/3357596

[15] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems* (Snowbird, UT, USA) *(GRADES'14)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/2621934.2621936

[16] Abdullah Gharaibeh, Tahsin Reza, Elizeu Santos-Neto, Lauro Beltrao Costa, Scott Sallinen, and Matei Ripeanu. 2013. Efficient large-scale graph processing on hybrid CPU and GPU systems. *arXiv preprint arXiv:1312.3018* (2013).

[17] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 17–30.

[18] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin yu Chen, Xiao-Fei Liao, and Hai Jin. 2019. A Survey on Graph Processing Accelerators: Challenges and Opportunities. *Journal of Computer Science and Technology* 34 (Feb. 2019), 339–371. doi:10.1007/s11390-019-1914-z

[19] 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)*. 1–9. doi:10.

1109/ICCAD51958.2021.9643582
[20] Intel. 2024. Intel turbostat. https://www.linux.org/docs/man8/turbostat.html.
[21] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment* 11, 3 (2017), 297–310.
[22] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.
[23] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 239–252. doi:10.1145/2600212.2600227
[24] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.
[25] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research* 11, 2 (2010).
[26] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.
[27] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of massive datasets*. Cambridge University Press.
[28] Hang Liu and H. Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1145/2807591.2807594
[29] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (apr 2012), 716–727. doi:10.14778/2212351.2212354
[30] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. doi:10.1145/1807167.1807184
[31] Microsoft Azure. Accessed: January 2025. *Microsoft Azure Pricing Calculator*. https://azure.microsoft.com/en-us/pricing/calculator/.
[32] Microsoft Azure. Accessed: January 2025. *Microsoft Azure Virtual Machines Documentation*. https://learn.microsoft.com/zh-cn/azure/virtual-machines/.
[33] NVIDIA [n. d.]. Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface.
[34] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 21-23, 2016*, Deming Chen and Jonathan W. Greene (Eds.). ACM, 111–117. doi:10.1145/2847263.2847337
[35] Open MPI [n. d.]. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/.
[36] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35, 3 (2015), 10–22. doi:10.1109/MM.2015.42
[37] 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.
[38] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 410–424. doi:10.1145/2815400.2815408
[39] Amin Sahebi, Marco Barbone, Marco Procaccini, Wayne Luk, Georgi Gaydadjiev, and Roberto Giorgi. 2023. Distributed large-scale graph processing on FPGAs. *Journal of Big Data* 10, 1 (2023), 95.
[40] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 979–990. doi:10.1145/2588555.2610518
[41] Ravi Sethi. 1977. On the complexity of mean flow time scheduling. *Mathematics of Operations Research* 2, 4 (1977), 320–330.

[42] 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* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 320–329. doi:10.1145/3289602.3293900

[43] Xuanhua Shi, Xuan Luo, Junling Liang, Peng Zhao, Sheng Di, Bingsheng He, and Hai Jin. 2018. Frog: Asynchronous Graph Processing on GPU with Hybrid Coloring Model. *IEEE Transactions on Knowledge and Data Engineering* 30, 1 (2018), 29–42. doi:10.1109/TKDE.2017.2745562

[44] Wonseok Shin, Siwoo Song, Kunsoo Park, and Wook-Shin Han. 2024. Cardinality Estimation of Subgraph Matching: A Filtering-Sampling Approach. *Proceedings of the VLDB Endowment* 17, 7 (2024), 1697–1710. doi:10.14778/3583140.3583155

[45] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. *SIGPLAN Not.* 51, 8, Article 11 (feb 2016), 12 pages. doi:10.1145/3016078.2851145

[46] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. 1078–1086. doi:10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199

[47] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1813–1828. doi:10.1145/2882903.2915220

[48] Yu-Wei Wu, Qing-Gang Wang, Long Zheng, Xiao-Fei Liao, Hai Jin, Wen-Bin Jiang, Ran Zheng, and Kan Hu. 2021. FDGLib: A Communication Library for Efficient Large-Scale Graph Processing in FPGA-Accelerated Data Centers. *Journal of Computer Science and Technology* 36 (Oct. 2021), 1051–1070. doi:10.1007/s11390-021-1242-y

[49] Xilinx [n. d.]. Vitis HLS. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html.

[50] Xilinx. 2021. U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.

[51] Xilinx 2021. Vitis Xbutil. https://xilinx.github.io/XRT/master/html/xbutil.html.

[52] Xilinx. 2022. xup_vitis_network_example. https://github.com/Xilinx/xup_vitis_network_example.

[53] Fan Zhang, Long Zheng, Xiaofei Liao, Xinqiao Lv, Hai Jin, and Jiang Xiao. 2022. An Effective 2-Dimension Graph Partitioning for Work Stealing Assisted Graph Processing on Multi-FPGAs. *IEEE Transactions on Big Data* 8, 5 (2022), 1247–1258. doi:10.1109/TBDATA.2020.3035090

[54] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 45–58.

[55] Jianlong Zhong and Bingsheng He. 2014. Medusa: A Parallel Graph Processing System on Graphics Processors. *SIGMOD Rec.* 43, 2 (dec 2014), 35–40. doi:10.1145/2694413.2694421

[56] 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. doi:10.1109/TPDS.2019.2910068

[57] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A {Computation-Centric} Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.

Received October 2024; revised January 2025; accepted February 2025