2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

NOVA: A Novel Vertex Management Architecture for Scalable Graph Processing

Marjan Fariborz[†][x][*], Mahyar Samani[‡][*], Austin York[§][x], S.J. Ben Yoo[‡], Jason Lowe-Power[‡], Venkatesh Akella[‡]
Ayar Labs[†], University of California Davis[‡], Google[§]
marjan@ayarlabs.com[†],{msamani, sbyoo, jlowepower, akella}@ucdavis.edu[‡], austinyork@google.com[§]

***Abstract*—We propose a scalable graph processing hardware accelerator called NOVA that is based on a novel vertex management architecture that decouples the execution of reduction and propagation operations in the popular vertex-centric graph processing paradigm. This allows us to store the working set in off-chip memory and utilize the available on-chip memory as a buffer to hide the latency of DRAM accesses instead of a traditional cache. This overcomes one of the key drawbacks of almost all the prior works which require temporal partitioning of graphs to scale to large graphs. We develop a cycle-accurate model of the architecture in gem5 and demonstrate that NOVA exhibits near-perfect weak and strong scaling while scaling to large graphs by spatially tiling multiple nodes. In addition, our simulations show that NOVA is 2.35× better than a state-of-the-art graph accelerator (PolyGraph) while using a fraction of the on-chip memory on a synthetic graph with 134M vertices and over 2.14B edges.**

I. Introduction

Modern data-driven scientific discovery is based on understanding relationships between data entities that are modeled as graphs—i.e., graph processing. Traditionally, graphs are processed on CPUs and GPUs using optimized libraries such as Galois [37], Ligra [41], GAPBS [10], Gunrock [46], etc., but the performance of software implementations on large graphs is unacceptably poor in many applications. As a result, there has been strong interest in new hardware accelerators for graph processing, with various proposals in the past few years [4], [13], [18], [32], [38], [43].

Most graph accelerators proposed in the literature leverage a vertex-centric programming model, where the task of vertex processing exhibits low spatial and temporal locality, leading to frequent random memory accesses. For such accelerators, the task of managing and storing active vertices is an important challenge since active vertices (vertices that have been visited and whose neighbors should be visited next) dictate the next work that needs to be done.

To address this challenge, prior research has explored leveraging locality and minimizing data movement overhead by storing vertices (or information such as messages to each vertex) in on-chip memory [4], [13], [18], [34], [38], [43]. Using on-chip memory allows for low-latency access and management of active vertices.

While storing vertices in on-chip memory is suitable for small graphs with a vertex set that fits within available on-chip memory, scalability becomes problematic as graph size increases. Large graphs necessitate partitioning into slices and time-multiplexing to utilize on-chip memory, a technique known as “temporal partitioning. Temporal partitioning reduces irregular off-chip memory accesses. However, for large graphs relying on temporal partitioning imposes overheads primarily caused by switching overheads between partitions. In addition, the generated partitions often exhibit inter-dependencies, requiring multiple time multiplexing iterations.

[*] M. Fariborz and M. Samani contributed equally to this work.
[x] Work done while at University of California Davis.

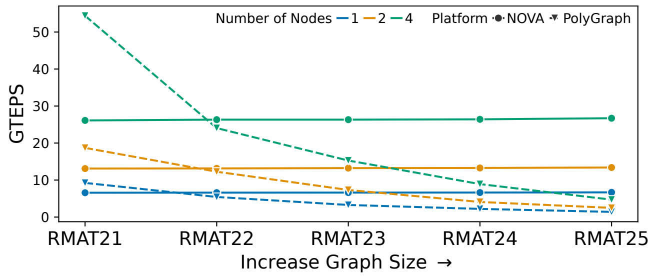


Fig. 1: Both systems have 1.5 MiB on-chip memory and 332.8 GB/s for BFS workload (per node). GTEPS (giga traversed edge per second) represents the system throughput.

Figure 1 shows that in accelerators that utilize temporal partitioning, the processing throughput decreases as the graph size increases. In this example, we use PolyGraph [13], the state-of-the-art graph accelerator with a low-cost temporal partitioning method. Figure 1 shows that PolyGraph achieves higher throughput for smaller graphs. However, the overheads of switching slices prevent PolyGraph from maintaining the same throughput for larger graphs. Thus, relying on temporal partitioning is not a scalable solution.

We propose a scalable architecture for message-driven graph processing called NOVA. NOVA is based on a novel vertex management architecture, which stores vertices in DRAM and uses a small SRAM buffer intelligently to hide the latency of vertex processing. The intuition behind our design is as follows: the two steps in vertex processing, namely, reduction and propagation, have different tolerances to latency. Reduction is more critical than propagation since it is very likely that a given vertex has multiple outgoing edges. In our approach, we prioritize reduction over propagation, which means that if there is insufficient space in on-chip memory, we spill active vertices to slower DRAM to create space for vertices that need to be reduced. This results in a decoupled execution machine (a well-studied paradigm in computing architecture [42]), allowing the reduction and propagation to proceed as fast as the memory bandwidth allows. Our experiments show that there is enough vertex-level parallelism in the asynchronous event-driven implementation paradigm to hide the latency of DRAM accesses. So, in effect, the on-chip SRAM serves more like the decoupling buffer between the reduction and propagation engines. This is in contrast to almost all prior work, where the on-chip SRAM is used as a traditional cache [13], [34], [38], [51]. However, the key challenge with the proposed approach is how to find active vertices if they are spread across both SRAM and DRAM. We develop an efficient vertex tracking mechanism to accomplish this, as described in detail in the paper.

The benefits of the proposed architecture (NOVA) and hence the main contributions of this paper are threefold: (1) We get a potentially much larger window for coalescing updates. Basically, one could view the act of going to the DRAM to get an active vertex as a form of delaying the propagation function, which means the vertex in DRAM can accumulate more updates before a new message is sent out. In turn, with the proposed decoupled execution approach, fewer messages are sent out, which improves work efficiency and overcomes one of the main drawbacks of asynchronous graph processing (compared to synchronous approaches) [13]. Prior work in the area of event-driven asynchronous graph processing resorts to an eager approach to message generation, i.e., a message is potentially generated as soon as an update arrives, which results in more messages being sent and, hence, lower work efficiency. Note that some approaches such as PolyGraph [13] mitigate this to some extent by complex task scheduling schemes to choose a particular vertex to process to improve work efficiency. (2) The proposed accelerator is not limited by the available on-chip SRAM to process a graph if we have sufficient SRAM to hide the latency of the slower DRAM; the proposed architecture is suitable for platforms with a limited amount of SRAM, like FPGA. (3) By removing SRAM size as a constraint, we do not have to resort to temporal partitioning for scaling. We can use spatial partitioning to scale arbitrarily. We are just limited by the available DRAM capacity at each node to store the vertices/edges. The performance of the proposed system is only limited by the available memory bandwidth, allowing us to achieve a balanced system, which is necessary for scalability.

The rest of the paper is organized as follows: we start with the definitions and background, followed by the details of the microarchitecture of the accelerator node. Next, we show how multiple accelerator nodes can be composed to handle arbitrarily large graphs. We discuss the evaluation framework and present results, including comparison with related work, sensitivity analysis on various design factors, and the details of the hardware prototyping of a single accelerator node on FPGA, and conclude with related work.

II. Background and Previous Work

A. Overview of Graph Analytics

Definitions: *Vertex-centric* programming is a common paradigm for implementing graph algorithms [3], [4], [10], [13], [18], [37]–[39], [41], [44]. In this computation paradigm, programmers describe graph algorithms as a series of operations from the perspective of a vertex [29]. Some graph processing workloads can be implemented using the *message-driven* paradigm, which can be thought of as each vertex sending *messages* to its neighbors. Each $message$, such as $< u, \delta >$, has two main attributes: a destination vertex (u) and an update (δ). In this *message-driven* model, an active vertex (such as u) calculates an update for each of its neighbors and sends the update through the message. Graph workloads can be described using *reduce* and *propagate* functions. Reduce determines the new property of a vertex given a message for the vertex and its current property. Propagate uses the property of a vertex along with the weight of its edges to create new messages for the neighboring vertex.

Metrics: Beamer et al. [9] discusses the importance of *traversed edges per second* (TEPS) as the performance metric in graph analytics. Execution models such as asynchronous graph processing might result in higher TEPS, but some edges are traversed redundantly. *Work efficiency* is the number of edges traversed by the sequential code over the number of edges traversed by asynchronous execution. Therefore, for asynchronous processing, the product of work efficiency and TEPS determines the performance. Asynchronous approaches coalesce multiple reductions to a specific vertex to increase work efficiency before propagating from the vertex.

B. Prior Work on Graph Accelerators

In the past decade, there has been a plethora of work on graph accelerators [3], [7], [18], [28], [31]–[33], [35], [38], [39]. Among these studies, some use a traditional memory hierarchy with multi-levels of private and shared caches [7], [31], [32]. However, these architectures suffer from data movement overheads due to poor data reuse in graph workloads. As a result, cache thrashing leads to more than 50% of all memory accesses missing in the last level cache [28]. Another group of graph accelerators rely on on-chip memory to hide the long accesses to memory and increase locality [18], [38], [39], [51]. These systems rely on the capacity of on-chip memory to exploit locality and remove inefficient access to the off-chip memory. These accelerators use temporal partitioning to scale to graphs larger than their on-chip memory capacity and spatial partitioning for graphs larger than their off-chip memory capacity. In *temporal partitioning*, on-chip memory can be shared by different partitions sequentially over time, while in *spatial partitioning*, multiple accelerator chips can house all slices while an interconnection network streams inter-slice events in real-time.

Accelerators such as GraphPulse [38] and JetStream [39] follow an asynchronous message-driven execution model. These studies separate communication messages from vertex/edge processing and utilize a global on-chip memory to store and coalesce messages targeting specific vertices. For graphs that exceed the on-chip memory capacity, GraphPulse partitions and time-shares the memory among slices. In contrast to GraphPulse and JetStream, NOVA allows vertex updates to spill over to a large-capacity memory. Additionally, like GraphPulse, NOVA assigns the processing of each specific vertex to a processing element (although NOVA uses static assignment while GraphPulse uses dynamic scheduling). Furthermore, NOVA supports both asynchronous message-driven execution and synchronous models.

PolyGraph [13] is a flexible accelerator that supports multiple run-time variants, including non-sliced, sliced, synchronous, and asynchronous modes. The non-sliced variant is only effective for small graphs or phases of the execution with a small number of active vertices that fit on the on-chip memory. PolyGraph takes advantage of intelligent work reordering to increase work efficiency. NOVA can support the same flexibility and workload variants of PolyGraph. In addition, the non-slice variant in NOVA is efficiently implemented even for graph sizes surpassing the accelerator's on-chip memory, resulting in better scalability than PolyGraph.

Dalorex [34] introduces the concept of data-local execution, similar to message-driven processing. It eliminates off-chip memory entirely, leveraging wafer-scale integration. Its performance is dictated by the on-chip memory bandwidth, resulting in significantly higher performance than prior work. Dalorex spatially slices the graph to scale to larger graphs and increases the number of accelerator cores. Processing a graph such as Twitter requires gigabytes of on-chip storage, which is very expensive. While Dalorex stores all the graph information on-chip, NOVA strives to utilize off-chip memory efficiently and significantly reduces the size of on-chip memory.

In addition to microarchitectural changes that allow NOVA to scale to large graphs beyond other studies, we also focus on system-level architecture to enable scaling out without overprovisioning system capacity and bandwidth.

C. Overheads of Temporal Partitioning

Previous accelerator designs enhance spatial and temporal locality by partitioning the graph into slices that fit into on-chip memory, which then time-share the limited on-chip resources. Temporal partitioning ensures that all the vertices that need processing reside in the on-chip memory during the processing of a slice. This design removes random accesses to the off-chip memory and reduces the access time to vertices. However, temporal partitioning comes with additional costs in preprocessing, portability, overhead of switching between slices, and underutilization of hardware resources.

1) Preprocessing Cost and Portability: Ideally, the graph should be partitioned into independent slices, each of which is highly connected. However, optimal algorithms for such preprocessing can exceed the computation complexity of graph processing algorithms. For example, polynomial-time solutions for min-cut have not been found yet, while BFS has a complexity of $O(\|V\| + \|E\|)$ [6], [8], [21] where V and E are the set of vertices and edges of graph respectively. In

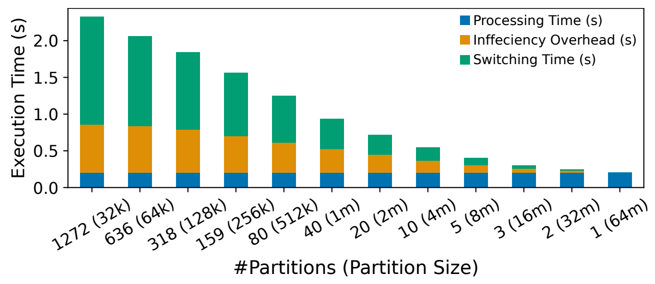


Fig. 2: BFS on Twitter: Temporal partitioning overhead as the number of slices increases. Slice size is shown in vertices.

recent studies, Balaji et al. show that RABBIT++ requires 1047 iterations of SpMV kernels to amortize preprocessing costs [8]. In addition, even with an optimal strategy (e.g., min-cut), it is impossible to eliminate inter-slice edges unless the graph contains disjoint communities.

Low-cost methods, like Gemini [59], partition the graph into chunks based on the *id* of vertices. Gemini works well for graphs as large as the Twitter graph. However, for larger graphs, cross-slice edges outnumber edges inside the slice.

Temporal partitioning lacks portability due to its reliance on available on-chip memory. For instance, a graph partitioned for an 8 MiB on-chip memory accelerator would need repartitioning for a different memory configuration.

2) Switching Cost and Resource Utilization: In addition to the preprocessing cost, there is also the cost of increased processing time due to switching and underutilization of hardware resources. There are three processing steps when switching between temporal slices: (1) the current slice's vertices must be written back to the off-chip memory, and the new slice's vertices must be read from off-chip memory. (2) Each slice has a set of replicated vertices to facilitate temporal partitioning. If a vertex updated by the current temporal slice resides in other slices, it must be read and updated in off-chip memory. (3) All replicated vertices of the new slice must be read to create inter-slice messages. Each slice must be processed multiple times due to inter-slice edges, adding to the work inefficiency.

To quantify the overhead of slice switching, we implemented the partitioning technique used in PolyGraph [13], [59]. We used the Twitter graph and BFS workload and broke the execution time into three components. (1) *Processing time* accounts for time spent processing slices. (2) *Switching time* accounts for time spent switching slices, including writing the current slice, reading the new slice, and reading/writing inter-slice replicated vertices. (3) *Inefficiency overhead* accounts for the time spent processing slices more than once.

Figure 2 shows the execution time breakdown between processing time, inefficiency overhead, and switching time as the number of slices grows (larger graphs). Inefficiency overhead and switching time constitute approximately 20% of the execution time when there are less than three slices. As the number of slices grows, the inefficiency overhead increases quickly. Figure 2 shows that when the graph is divided into 318 slices, inefficiency overhead makes up more than 75% of the execution time. Therefore, relying on temporal partitioning to process large graphs is not a feasible solution for scalability.

New graph partitioning methods are introduced to reduce the overheads of temporal partitioning, such as preprocessing and switching cost [17], [49]. However, these studies target graph applications such as Graph Convolutional Networks (GCNs) that have specific characteristics, such as synchronous execution and being topology-driven (all vertices are active during their execution) [37]. Recent I-GCN [17] introduced an online graph restructuring algorithm that locates highly connected subgraphs using graph traversal algorithms, such as BFS, to partition graphs during the execution of GCN. NOVA targets more general workloads and can process asynchronously and synchronously for data and topology-driven [37] applications.

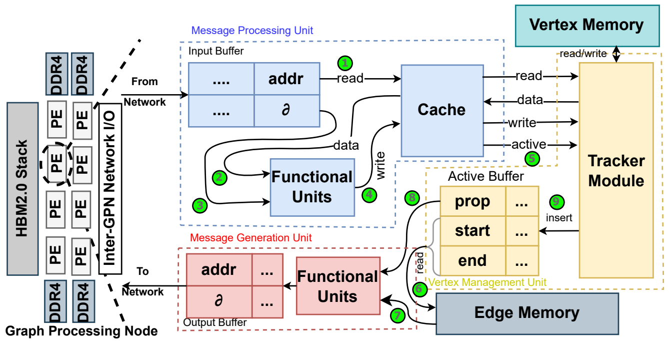


Fig. 3: Organization of a GPN with details of one PE, and its data path. PEs are connected through point-to-point network.

III. Microarchitecture of NOVA

Relying solely on locality to enhance performance in an accelerator core leads to an architecture where throughput is influenced by graph size. As the graph size grows, the accelerator cores performance diminishes, resulting in a non-scalable architecture. NOVA aims to achieve high processing throughput regardless of graph size, addressing the scalability challenges faced by previous approaches. We accomplish this by efficiently utilizing the available memory bandwidth to access graph edges. However, maintaining high processing throughput remains challenging due to the random nature of vertex information accesses within the graph.

During the execution of a graph workload, vertex information is accessed for reduction and propagation. Both of these operations require fast memory access to maintain high throughput. However, for each active vertex, multiple propagation operations are necessaryone for each edge. Additionally, reading edges exhibits higher spatial locality. Consequently, the propagation process can handle delays in accessing vertex information better than the reduction process.

Our microarchitectural insight centers around achieving high throughput by simultaneously processing multiple reductions in parallel to mitigate the impact of long memory access latency. We achieve this by storing an optimal number of active vertices in the fast, small on-chip memory. In addition, efficiently utilizing memory bandwidth for edge accesses is crucial for improving processing throughput.

We create a decoupled pipeline that allows independent execution of reduction and propagation. As active vertices are generated by the reduction operation and consumed by the propagation operation, we orchestrate their movement between reduction and propagation to achieve effective decoupling. Our design allows active vertices to spill into off-chip memory to create space for additional reduction operations. Consequently, active vertices not yet processed by propagation may be removed from on-chip memory.

To complement this, we introduce a run-time process that tracks and retrieves active vertices as needed during propagation. Our approach has the following benefits: ❶ Spilling active vertices to off-chip memory allows the active vertex set to span on-chip and off-chip memory, which naturally expands the coalescing space and improves work efficiency. ❷ Our design ensures that reduction operations can occur even when many active vertices are awaiting processing by the propagation operation. As a result, we meet the requirements for deadlock freedom, as outlined by Eicken et al. [14].

The following sections discuss our designs for implementing hardware support for each of reduction, propagation, spill, and retrieval mechanisms.

A. Overall Design of NOVA

NOVA comprises multiple graph processing nodes (GPNs). Each GPN consists of multiple processing elements (PE), which are message-driven processors capable of executing algorithms expressed in Algorithm 1. Algorithm 1 shows the decoupled message-driven implementation of the SSSP workload. Decoupling the reduction from propagation enables NOVA to support both asynchronous [55] and bulk synchronous parallel (BSP) [45] execution models. During BSP execution, the blue block (message processing) and the red block (message generation) are executed in series. This serial execution is enforced by the decoupled *next_active* (shown in yellow). In asynchronous execution, all blocks are executed simultaneously until there are no more messages [55].

A PE consists of three main units corresponding to the three parts of Algorithm 1. The colors of the units in Figure 3 correspond to the colors in Algorithm 1. (1) *Message processing unit* processes messages and updates vertices. It determines the new property of a vertex using the reduce function. (2) *Vertex management unit* keeps track of active vertices from the message processing unit and sends active vertices to the message generation unit. (3) *Message generation unit* uses active vertices and their edges to generate new messages. It produces the update in a message using the propagate function.

In a PE, the vertex management unit interfaces with message processing and message generation unit to create the *illusion* that the *active buffer* (in Figure 3) has capacity equivalent to the size of the off-chip memory. The message processing unit *pushes* vertices into active buffer as they become active and the message generation unit *pulls* active vertices from active buffer when propagating messages.

Algorithm 1: Decoupled message-driven implementation for single source shortest path.

```
dist[:$\|V\|$] = $\infty$;
messages.append({source,0});
while not messages.empty() do
    for u, $\delta$ in messages do
        old_dist = dist[u];
        dist[u] = min(old_dist, $\delta$); // reduction
        if old_dist != dist[u] then
            track_active(u); // Path 5 on Figure 3
// Tracker Module implements next_active()
for v in next_active() do
    s, e = row_ptr[v], row_ptr[v+1]-1;
    v_info.append({dist[v],s,e}); // active buffer
while not v_info.empty() do
    // Path 6 on Figure 3
    for $\alpha$, start, end in v_info do
        for i in range(start, end) do
            dest = edge_dests[i];
            $\epsilon$ = add($\alpha$, edge_wgt[i]); // propagation
            messages.append({dest,$\epsilon$});
```

B. Message Processing Unit

The message processing unit processes vertices based on incoming network messages. To process a message ($< u, \delta >$), the destination vertex (u) has to be read (path 1 in Figure 3). After the vertex is read, its property (e.g., distance in case of SSSP) and the message's update (δ) are used by the reduce function (e.g., minimum in case of SSSP) to determine the new property of the vertex (paths 2 and 3 in Figure 3). After the reduction, the memory block in the cache is updated with the new copy of the vertex (path 4 in Figure 3). If the reduction operation creates an active vertex, the vertex management unit is informed so that it can track the vertex as an active vertex (path 5 in Figure 3).

Due to a lack of locality in accesses, reading vertices from DRAM can result in long access latency. The message processing unit utilizes a direct-mapped cache with a write-back policy to enable parallel and fine-grained access to memory. Nevertheless, due to the massive size of most graphs, the cache is unlikely to capture much locality. In our implementation, we have configured this cache with a size of 64 KiB for each PE. In Section VI-C1, we demonstrate that performance remains consistent across various cache sizes. Consequently, we selected the smallest cache size.

C. Message Generation Unit

The message generation unit generates messages from active vertices to their neighbors. The messages are generated using the edges of active vertices from the vertex management unit (paths 6 and 7 in Figure 3). The unit initiates its process by reading an entry from the active buffer. Each entry in the active buffer has three members $< \alpha, start, end >$. α denotes the property of the active vertex, and $[start, end]$ identifies the location of its edges in the edge memory.

TABLE I: Trade-offs of different active vertices spilling methods.Vertex set overwriting requires no extra coalescing cost.

Spilling Method	Off-chip buffer	Overwriting in vertex set
Number of Writes Per Spill	2: 1 to vertex set and 1 to FIFO	1: 1 to vertex set
Retrieval Cost	Read from FIFO	Search for active vertices
Coalescing Cost	Search FIFO for the same vertex	Overwrite in vertex set
Metadata	Need to store vertex address explicitly	None
Extra off-chip Memory usage	O(VE)	0

For each edge, $< v, w >$, in $[start, end]$, a message is generated. The destination of the message is determined by the destination vertex of the edge (v); the message update is calculated using the propagate function on the property of the vertex (α) (path 8 in Figure 3) and the weight of the edge (w). Subsequently, the message is sent to the network, where it is received by its designated message processing unit. The addressing function is assigned at initialization time since vertices are statically assigned to PEs.

D. Vertex Management Unit

The key component of our proposed pipeline is the vertex management unit. This unit breaks the dependency of the message generation unit on the message processing unit by mediating active vertices between them. The vertex management unit tracks active vertices produced by the message processing unit—on-chip or off-chip, and delivers them to the message generation unit on demand.

A straightforward technique to manage spilled vertices involves storing copies of the active vertices in an auxiliary buffer located in the off-chip memory. To retrieve spilled active vertices, they will be read from this buffer. However, utilizing an additional off-chip buffer presents few disadvantages: a) spilling an active vertex requires two write operations—one to the vertex set and one to the buffer. b) Without coalescing different copies of the same vertex in the off-chip buffer, the buffer size could increase significantly. c) To enable coalescing, it is necessary to maintain the address or ID of each vertex in the buffer, which requires additional capacity, and to search the buffer for the same vertex, which requires additional bandwidth.

In NOVA, we allow every spilled active vertex to overwrite its previous value in the off-chip memory. In turn, NOVA does not require extra capacity or bandwidth to spill the active vertices into off-chip memory. However, with our approach, retrieving active vertices from the off-chip memory requires searching the vertex set for active vertices—requiring additional bandwidth from the off-chip memory and incurring long access latency. To reduce the required bandwidth and access latency for retrieving spilled active vertices, we store on-chip information (meta-data) about the location of such vertices in the off-chip memory (line 13 in Listing 1).

Table I summarizes the trade-offs between using an off-chip buffer and overwriting vertices in the vertex set.

In the following, we discuss the details of how the vertex management unit tracks the location of active vertices in the

Listing 1: Vertex management unit procedures to track spilled active vertices and provision active vertices to message generation unit. on_evict is called when a cache block is evicted.

```
 1
 2 struct Vertex{cur_prop, next_prop, active_now, active_next}
 3
 4 superblock_size = superblock_dim * block_size
 5 def active(block):
 6   is_active = False
 7   for vertex in block:
 8     is_active |= vertex.active_now
 9   return is_active
10
11 def get_superblock_number(block):
12     return floor(block/superblock_dim)
13
14 def track_as_active(block)
15     # each superblock has a counter in tracker module
16     counter[get_superblock_number(block)] += 1
17
18 def untrack(block):
19     counter[get_superblock_number(block)] -= 1
20
21 def on_evict(block):
22   if active(block):
23     track_as_active(block)
24   write(block)
25
26 def next_superblock():
27   for index in len(counters):
28     if counters[i] > 0:
29       return i * superblock_size
30
31 def prefetch():
32   start = next_superblock()
33   end = start + superblock_size - block_size
34   step = block_size
35   # below range is inclusive of end
36   for block_addr in [start, end, step]:
37     block = read(block_addr)
38     if active(block):
39         active_buffer.insert(block)
40         untrack(block)
41
42 def next_active():
43     block = active_buffer.front()
44     ret = null
45     for vertex in block:
46         if vertex.active_now:
47             ret = vertex
48     if active_buffer.critically_low:
49         prefetch()
50     return ret
```

off-chip memory. In addition, we discuss the trade-off between the capacity requirements of storing tracking information and the precision of locating active vertices in the off-chip memory.

The vertex management unit has two main components. (1) *Tracker module* that includes on-chip memory and control logic used to store metadata to track active vertices in the vertex memory and to recover active vertices from the vertex memory (line 13 in Listing 1). (2) A FIFO buffer to store prefetched active vertices for the message generation unit (*active buffer* in Figure 3).

In our design, we use three techniques to reduce the required on-chip storage and memory bandwidth and hide the long latency of retrieving vertices: (1) we track memory blocks (memory atoms) with active vertices, **not** the vertices themselves, (2) we group memory blocks into superblocks to reduce the tracking metadata, and (3) we prefetch active vertices before the message generation unit requests them. Techniques (1) and (2) reduce the capacity to store tracking information and the required bandwidth to search off-chip memory and technique (3) hides the long memory access latency.

To illustrate the required on-chip capacity for the tracker module for a real use case, we use the WDC12 [2] graph, which has approximately 3.6 billion vertices and 129 billion edges. We store the vertices in HBM2 memory, which has an atom size of 32 bytes. We assume 16 bytes as the size of the vertex data structure (vertex set size in WDC12 is 57.6 GiB).

Naively, we can keep a single bit for every vertex in the graph, denoting whether the vertex is active (a bit vector). This solution will require approximately 440 MiB of on-chip storage for this example. Furthermore, the size of the bit vector limits the number of vertices that can be tracked.

To reduce the capacity requirements of the tracker module, we can track active vertices at the granularity of memory blocks. We will refer to a memory block that stores **at least** one active vertex as an *active block*. With this approach, we can reduce the tracker module's required capacity to approximately 220 MiB in our example.

While storing a bit vector can accurately locate active vertices in the off-chip memory, they require large storage for operation. In NOVA, we track active vertices based on a *superblock* of N memory blocks and count the number of active blocks in the superblock (function *track_as_active* in Listing 1). Therefore, grouping more blocks into a superblock significantly reduces the capacity of the tracker module. The total capacity required by such an implementation is calculated using Equation 1 and Equation 2, where $superblock\_dim$ denotes the number of memory blocks grouped into a superblock, and $block\_size$ denotes the block size for the vertex memory. We use $superblock\_dim = 128$ and $block\_size = 32B$ (HBM2 supports 32 B accesses). Our design requires only 16 MiB of on-chip memory to track all active vertices in WDC12, which is 27× smaller than tracking via a bit vector.

Using superblocks to track active vertices introduces a trade-off between the amount of on-chip memory (for the tracker module) and the number of associative searches for each retrieval. Instead of **eliminating** associative searches, our approach **reduces** the number of required associative searches while using small on-chip resources. The maximum size of the tracker module only depends on the size of the vertex memory assigned to each PE, $superblock\_dim$, and $block\_size$, and it is independent of the size of the stored graph.

$$\text{cap}_{\text{bits}} = (\log_2 \text{superblock\_dim} + 1) \times \text{num\_superblocks} \quad (1)$$

$$\text{num superblocks} = \frac{\text{vertex memory capacity}}{\text{superblock dim} \times \text{block size}} \quad (2)$$

To hide the latency of searching for active blocks in the vertex memory, the vertex management unit prefetches (function *prefetch* in Listing 1) active blocks into the active buffer (path 9 in Figure 3). The prefetch logic is configured to read 16 blocks of memory from a superblock whenever 16 or more entries are available in the active buffer (line 45 in Listing 1). Only the active blocks are placed in the buffer, and the remaining blocks are dropped. Our simulation results showed that making the active buffer bigger than 80 entries (each entry holding one memory block) has diminishing returns. Therefore, in our simulation, we assign 80 entries to active buffer.

Listing 1 demonstrates the runtime procedures performed by the vertex management unit that allows it to mediate vertices between the message processing unit (*on_evict*) and the message generation unit (*next_active* also found in Algorithm 1).

Since the process of retrieving active vertices is done by doing associative searches within superblocks, we expect to see wasted bandwidth in the accesses to the vertex memory (for reading non-active blocks while searching for active blocks). We investigated the effect of vertex recovery on vertex memory bandwidth utilization and the performance sensitivity to the tracker module's size in Section VI-C2.

IV. System-level Architecture of NOVA

We propose a scale-out approach to handle graphs with tera/peta-scale capacity that exceeds the memory assigned to a single processing element (PE). We distribute the graph by assigning each vertex to a single PE. As a result, only one PE can update a specific vertex address, removing any need for atomic accesses. Messages between PEs are transmitted through the network to the PE that owns the vertex. In this setup, there will never be a situation where two PEs need to update the value of a single vertex. We allocate vertex and edge memory to each PE to facilitate this interaction. Consequently, we separate the PE-to-memory interaction from the PE-to-PE interaction, reducing traffic on the inter-PE interconnect fabric.

A. Choice of off-chip Memory

While NOVA can utilize any off-chip memory to store vertices and edges, ensuring a scalable architecture requires optimizing bandwidth and capacity utilization while avoiding resource overprovisioning. Therefore, we must carefully select memory systems that align with the unique access patterns and capacity/bandwidth trade-offs inherent in graph processing. This approach enables the development of a balanced system.

In graph workloads, vertices and edges have different requirements for capacity and bandwidth. In general, edges require a larger memory capacity compared to vertices. In most graph programming models [25], [37], access to edges is sequential and read-only. In contrast to edges, vertices have low spatial and temporal locality in most graph algorithms. Due to these differences, we have chosen a heterogeneous system for the off-chip memory in PE.

We have chosen HBM2 as the off-chip memory to store vertex information. HBM2 is an appropriate choice for storing vertices because it offers substantially more bandwidth under **random** access patterns [47]. Moreover, HBM allows for finer grain accesses (32 bytes), resulting in less bandwidth waste. We have chosen DDR4 for storing the edges because of its high capacity density compared to HBM2 and high bandwidth under sequential access.

Inside a GPN, we assigned eight PEs to one stack of HBM2 memory (eight channels). Each PE is dedicated to one channel of the HBM2 stack and operates only on the vertices stored in that channel. Previous studies show that vertex memory needs to offer $4\times$ the bandwidth of the edge memory [16]. So we allocate four DDR4 channels for the edge memory for each GPN. Though we use HBM2 as the vertex memory and DDR4 as the edge memory, our design is not limited to these specific memory technologies. Any memory technology that provides the required bandwidth and capacity for vertices and edges can be used as long as the required balance is achieved.

B. Spatial Vertex Mapping

Our proposed architecture assigns each vertex and its edges to a single PE. Choosing the vertex assignment is a tradeoff between preprocessing cost, load balancing, and locality. In a *load balanced* system, a similar number of edges are assigned to each PE for processing. Optimizing load balance requires sorting the vertices by their out-degree and uniformly distributing the vertices with the highest outdegrees across PEs. Interleaving vertices between PEs with a fine granularity ensures load balance. In the locality-based approach, we used community detection techniques such as RABBIT [6] to detect highly connected vertices and assign sequential ids to vertices in each community which improves locality. However, dividing a graph into communities is computationally expensive and should be avoided. By taking advantage of locality, we can reduce network traffic at the cost of a load balance. Finally, we can also use the original ordering made by the graph publisher and eliminate any pre-processing. In this case, we interleave the vertices based on their vertex ids between PEs, assigning a similar number of vertices to each PE. In Section VI-C3, we show the sensitivity of our design to different spatial partitioning methods.

C. Interconnection Network

A scalable accelerator needs to process any graph of any size with minimum impact on performance. To handle large graphs that exceed the memory capacity of a single GPN, we need to increase the number of GPNs. In multi-GPN systems, it is essential to prevent message communication between accelerators from becoming a performance bottleneck, thereby maintaining high performance and high memory utilization per GPN. A well-partitioned graph with a low rate of inter-slice messages does not require a high-bandwidth network. However, creating a well-partitioned graph with few inter-slice messages involves a high pre-processing cost. Thus, we aim to use an interconnect technology to handle the traffic when the graph is randomly distributed among the accelerators.

For a *scalable* accelerator, we must make sure that the interconnect has sufficient bandwidth to prevent data movement between GPNs from becoming the bottleneck. In NOVA, each GPN has dedicated edge and vertex memory, which means a GPN does not need to access the (remote) memory in another GPN. This approach separates memory traffic from inter-node traffic and uses the interconnection fabric only to send messages to neighboring vertices residing in remote GPNs.

We can calculate the maximum traffic on the interconnect by the maximum messages generated by each GPN, which in turn depends on the edge memory bandwidth. The traffic generated from each GPN is equivalent to the bandwidth of edge memory. In NOVA, we assume 4 DDR4 channels/GPN as the GPN edge memory with the aggregated bandwidth of 76.8 GiB/s.

In addition to the bandwidth requirement, we aim for low-latency message communication between GPNs. Therefore, a high-radix, low-diameter fabric that connects multiple GPNs and provides low-hop communication between GPNs is desirable. Point-to-point network topologies or crossbar topologies, where every GPN in the network is connected to every other GPN through a grid of high radix switches, are desirable designs. For example, if one were to implement this system today, one could consider using Broadcoms Tomahawk switch [11] that provides an overall bandwidth of 51.2 Tb/s with 256 lanes. By assigning two lanes per GPN, we can scale up to 128 GPNs using this switch, accomodating graphs requiring up to 16 terabytes of memory. Note that the large publicly available graph today is WDC12 [2], which requires 1 terabyte of memory, which means with an off-the-shelf switch as the interconnection network with the proposed accelerator nodes, we can scale to graphs that are 16 times larger than the largest publicly available graph today.

The key takeaway is that separating GPN-to-GPN and GPN-memory traffic reduces the amount of bandwidth needed on the interconnection fabric, allowing us to scale-out to process significantly large graphs using today's switch technology.

V. Methodology

We implemented our model in gem5 v22.0 [26]. We used gem5 to model and evaluate our design and baseline. gem5 is a cycle-level simulator that measures execution time directly without depending on indirect methods. gem5 has pre-existing memory models that are validated [40]. Furthermore, gem5 provides a straightforward interface for modeling interactions with memory. We integrated new models (called SimObjects in gem5) into the gem5 code base to model our design. We have developed models for all the components in a PE. In our design, we used already available models in gem5 for HBM2 and DDR4 to model off-chip memory. We modeled an 8×8 point-to-point electrical interconnect between PEs in a single GPN and a model of a crossbar switch as the interconnect between GPNs. Table II shows our system specifications.

State-of-the-art accelerators such as PolyGraph and Delorax have already demonstrated superior performance and efficiency compared to GraphPulse, Chronos, Graphicionado, and Ozdal. Hence, we only compared PolyGraph, which shows the best performance compared to other prior work. We also implemented a model of **PolyGraph** [13] in gem5, including the temporal slicing mechanism. We chose the most optimized variant and slice switching of PolyGraph as our baseline. We have modeled PolyGraph in its S_s, A_c, T_w variant. Before switching to a new slice, we process a temporal slice until no new messages are generated. We assumed the accelerator could parallelize the process of switching slices and fully utilize the memory bandwidth for our modeling. To compare against software platforms, we used **Ligra** [41] on an 8-core x86 machine with 32 MiB L3 cache and 400 GB/s memory bandwidth.

We used random partitioning to assign vertices to different PEs. We implemented five graph analytics workloads. We used BFS, CC, and SSSP in the asynchronous mode and PR and BC in the bulk-synchronous mode. BC, in its proposed asynchronous implementations, requires forward and backward passes, which doubles the number of edges required to be stored. Our implementation of PR-delta, as specified by [38], proved to be very sensitive to the order of the traversal of the graph. Finding the optimal order of traversal requires an overall graph view at the time of scheduling updates. Therefore, ordering is not a feasible solution for problem sizes significantly bigger than the size of on-chip resources. Hence, we have chosen to implement PR in BSP mode.

TABLE II: System specifications.

Specifications per GPN	NOVA
# PE	8 @ 2GHz
Spad	512 KB (cache) + 1 MiB (VMU)
Vertex memory	HBM2 stack - 4 GiB cap. - 256GB/s
Edge memory	4 DDR4 channels - 128 GiB cap. - 76.8GB/s
Functional Units	16 for reduction + 48 for propagation
PE-PE Network	8×8 Electrical Network, 1.2 GB/s bandwidth per link
Inter-GPN Net.	8×8 crossbar switch with 60 GB/s port bandwidth [11]

TABLE III: Graph Workloads used in evaluations.

Graph	Footprint	Vertices	Edges	# Slices with 32 MiB on-chip memory
RoadUSA [1]	805.7 MiB	23.9M	58.3M	3
Twitter [22]	14.4 GiB	41.65M	1.46B	5
Friendster [22]	15.4 GiB	65.6M	1.8B	8
Host [2]	16.6 GiB	101M	2B	13
Urand [15]	34.0 GiB	134.2M	4.2B	16

Our objective is to evaluate how NOVA performs for large graphs. Table III demonstrates the details of each of our input graphs. We have used a combination of synthetic graphs (Urand and RMAT [23]) along with real-world graphs such as Twitter. Previous accelerators used Twitter and RMAT 2^{26} as their largest graph input [13], [34]. We evaluate NOVA using Urand, which has $2\times$ more vertices and $1.5\times$ edges compared to Twitter, as the input to our workloads.

VI. Evaluation

To evaluate the performance of NOVA, we conducted a thorough evaluation across multiple dimensions. Our evaluation includes: (1) we compare our performance with PolyGraph and identify the overheads of each design, (2) we evaluate the scalability of our design for both strong and weak scaling, (3) we analyze the sensitivity of our performance to various design parameters, and (4) we provide an estimation of hardware resources needed for PolyGraph, Dalorex, and NOVA to scale to WDC12 size graphs. (5) We estimate the power and area of a single GPN based on our implementation on FPGA.

Due to the significant simulation time, we could not simulate beyond 8 GPN systems (64 PEs), which is a system that can process large graphs such as WDC12.

A. Comparison to State-of-the-art

Figure 4 compares NOVA with PolyGraph and Ligra with the same amount of off-chip memory bandwidth. In this comparison, both NOVA and PolyGraph are provisioned with

332.8 GB/s of off-chip memory bandwidth, which is equivalent to the aggregate bandwidth of one NOVA GPN with one HBM stack (256 GB/s) and four DDR4 channels (76.8 GB/s). While NOVA uses 1.5 MiB of on-chip memory (512 KiB for the cache and 1 MiB for the tracker module), PolyGraph uses 32 MiB of on-chip memory.

Figure 4 shows that when running BFS for the Twitter graph PolyGraph is 30% faster than NOVA. In this case, PolyGraph can process Twitter graph using only 5 temporal slices. However, as discussed in Section II-C, the overhead of switching temporal slices increases significantly as the number of slices grows. For graphs such as Friendster, Host, and Urand that are larger than Twitter, NOVA outperforms PolyGraph. Overall, as the size of the graph increases, NOVA gets higher speedup compared to PolyGraph, ranging from 1.15× faster for Host (PR) to 2.35× faster for Urand (SSSP).

In all cases, NOVA utilizes 80% to 85% of the edge memory bandwidth. However, PolyGraph does not leverage the memory bandwidth efficiently, using only 25% to 35% of the bandwidth for processing edges while the rest is spent switching slices. As input graphs become larger, switching slices constitutes a bigger part of the execution time, resulting in NOVA exhibiting higher performance than PolyGraph.

In previous studies [13], [38], the on-chip memory is used to decrease data access latency and consolidate the number of updates, leading to improved work efficiency. NOVA takes this step further by expanding the coalescing space from the on-chip memory to the off-chip memory. As shown in Figure 5, NOVA can coalesce up to 3× more messages than PolyGraph, resulting in better overall work efficiency. This shows that NOVA's approach to expanding the coalescing space and decoupling processes can significantly improve work efficiency.

Figure 6 shows the breakdown of total execution time between processing time and overhead time for NOVA and PolyGraph. Processing time is the time that the accelerator spends processesing vertices (reduction and propagation). For PolyGraph, overhead time constitutes the amount of time spent on switching slices; for NOVA this time constitutes the time spent reading inactive vertices while fetching active vertices in the memory (overfetching). Although in most cases the processing time is shorter for PolyGraph, as the graphs grow larger, the overhead time starts to outweigh the benefits of the increased locality.

B. *Scalability Analysis*

Figure 7 shows how the performance changes as we increase the number of GPNs for a fixed graph size (**strong scaling**) for BFS and BC. However, other workloads see a similar scaling trend. BFS is an example of a data-driven workload that experiences dynamic changes in the number of active vertices. In contrast, BC is a topology-driven workload in which the graph determines the active nodes.

In general, NOVA shows a near-perfect performance scaling as the number of GPNs grows. We observed a maximum 19% difference between the ideal scaled performance and NOVA's performance (Twitter in Figure 7b). For the Urand graph, performance grows beyond the ideal scaling due to increased work efficiency. Overall, NOVA achieves excellent scalability in performance as the number of GPNs increases.

Figure 8 demonstrates how NOVA's performance improves as we increase the number of nodes for a fixed problem size *per node* (**weak scaling**). Weak scaling is typically employed for memory-bound applications that require a memory capacity beyond the capabilities of a single node. As illustrated in Figure 8, increasing the resources and problem size will not lead to performance degradation. In an ideal scenario of perfect weak scaling, twice as many GPNs can process a graph twice the size as the baseline in the same amount of time.

C. *Sensitivity Analysis*

1) Sensitivity to Cache Size: Each PE uses a cache to hide the long off-chip random access latency. Due to the significant size of the graphs, the cache cannot capture much locality in accesses to the vertex memory. Figure 9a shows that the cache size does not affect performance for cache sizes beyond 64KiB. For roadUSA with 4M of on-chip memory, most of the graph fits within on-chip memory resulting in higher speedup.

Overall, we find less than a 2% performance improvement when increasing the cache size from 64KiB to 4MiB per PE (512KiB to 32MiB per GPN). The average bandwidth between two graphs and different cache sizes is 6.4 GTEPS, which is 80% of this system's peak achievable bandwidth, showing a high off-chip bandwidth utilization. Moreover, Figure 9a demonstrates a small change in the execution time for both graphs, showing that the throughput and work efficiency in large graphs are independent of cache size for each GPN.

The takeaway from this analysis is that our performance is independent of the size of the on-chip memory, even when we scale-out the number of GPNs to process large graphs.

2) Sensitivity to Tracker Module Size: As discussed in Section III-D, it is important to identify the effect of the superblock dimension of the vertex management unit on the system behavior. Therefore, we have evaluated three different grouping dimensions, 32, 64, 128, and 256, to implement the vertex management unit, which requires 3MiB, 1.75MiB, 1MiB, and 576KiB of on-chip storage, respectively. We have used BFS and PR as representative programs and Twitter and RoadUSA as input for our experiments.

The blocking technique cannot directly pinpoint the location of active blocks in the memory. Therefore, evaluating the overheads introduced by the search for active vertices in each block is important. We measured the bandwidth waste of the vertex memory for different blocking dimensions. Figure 10 summarizes the division of the vertex memory bandwidth between useful reads, writes, and wasteful reads in proportion to the peak theoretical bandwidth of the vertex memory. The bandwidth marked as wasteful reads represents the bandwidth used to read inactive vertices while searching a superblock for active vertices. Figure 10 does not show observable change as the size of the vertex management unit changes.

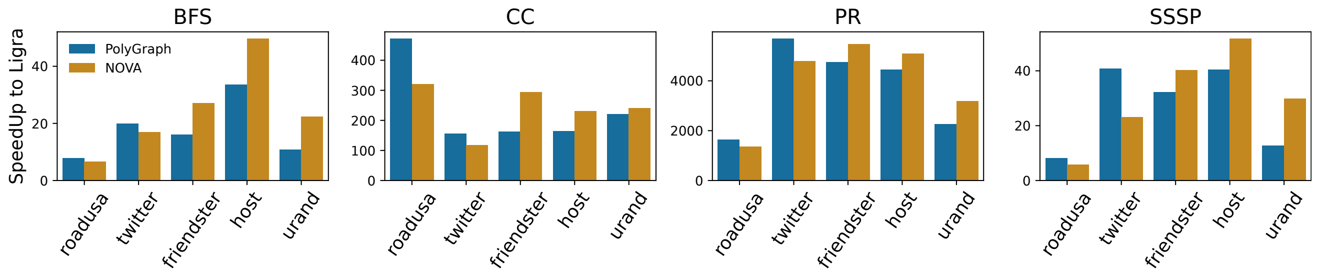


Fig. 4: NOVA vs. PolyGraph (iso-bandwidth 332.8 GB/s) vs. Ligra: For larger graphs, NOVA beats PolyGraph with 32 MiB of on-chip memory while using only 1.5 MiB of on-chip memory.

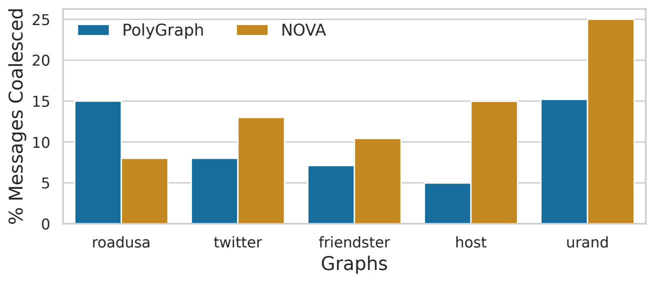


Fig. 5: The percentage of messages coalesced in NOVA compared to PolyGraph using BFS workload.

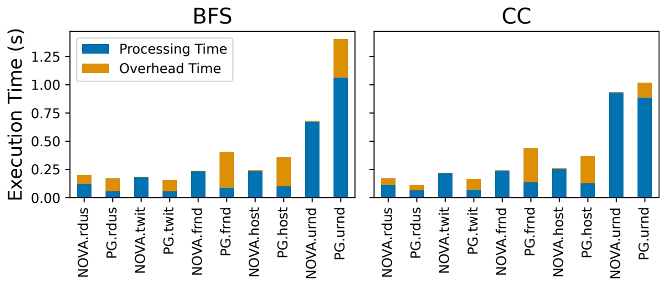
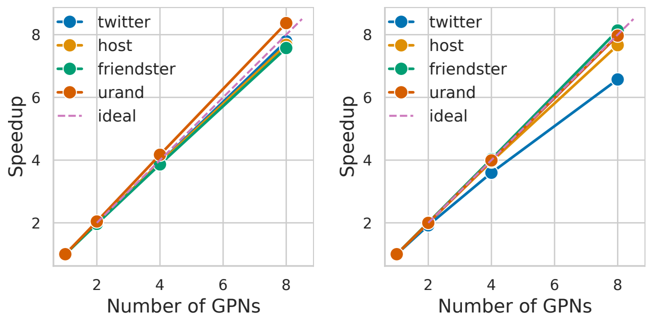


Fig. 6: Comparing execution time breakdowns between NOVA and PolyGraph (PG). Although PG processes the graph faster due to its fast vertex access, the overheads of switching slices negate the benefits from increased locality.



(a) BFS (asynchronous)

(b) BC (bulk-synchronous)

Fig. 7: Strong Scaling Analysis: How number of GPNs affect performance for a fixed graph.

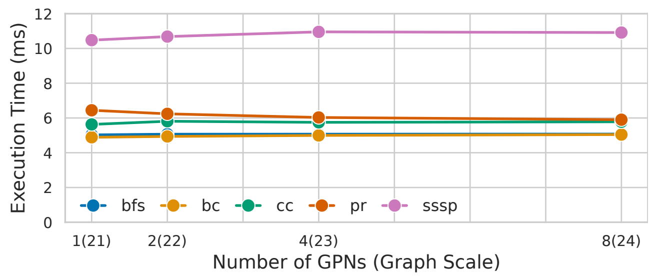
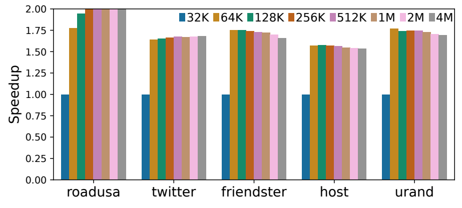


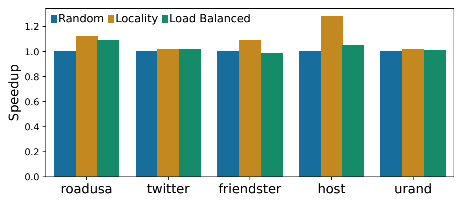
Fig. 8: Weak scaling analysis: Increasing both GPNs and graph size. using synthetic graphs RMAT21-24 and BFS. Ideally, performance stays constant.

RoadUSA shows significant bandwidth waste because it has a high diameter with few active vertices. Furthermore, high-diameter graphs commonly have smaller average degrees, which creates less slack for the vertex management unit to search for active vertices in the DRAM. In this case, the vertex management unit's prefetching mechanism aggressively overfetches, resulting in the bandwidth waste. Figure 10 shows that dense frontier workloads, such as PR (right), result in a smaller wasted bandwidth than sparse frontier workloads (BFS). When the frontier is dense, the number of active vertices in a block grows, resulting in lower wasted bandwidth. For the case of running BFS with RoadUSA, increasing the size of the tracker module does not reduce the amount of wasteful reads noticably. This is due to the particular sparsity of the frontier. Morever, there are very few active vertices that reside on DRAM (since the graph is small). Therefore, the tracker module with 3 MiB of capacity does not have enough resolution to reduce the wasted bandwidth. We observed a drop in work efficiency when we changed the size of the vertex management from 1 MiB to 576 KiB. For all other performance evaluations, we used 1 MiB as the size of our vertex management unit.

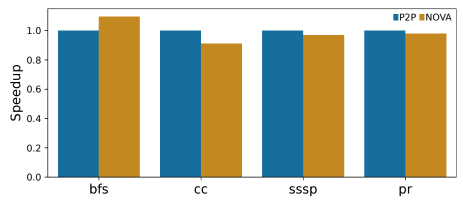
3) Sensitivity to Spatial Vertex Mapping: Figure 9b shows the sensitivity to different vertex placement mechanisms. We compared 3 placements: one that is load-balanced, one that is locality-optimized (RABBIT [6]), and one random vertex assignment with no preprocessing cost. We observed that locality optimized shows at most a 20% improvement compared to load-balanced and random due to better overall



(a)



(b)



(c)

Fig. 9: (a) Sensitivity of a single GPN to size of cache in each PE. (b) Sensitivity to the arrangement of vertices across PEs (8-GPN system. (c) Sensitivity to fabric topology. NOVA: the network described in Table II. P2P: PEs connected through a point-to-point network with infinite bandwidth (8-GPN system).

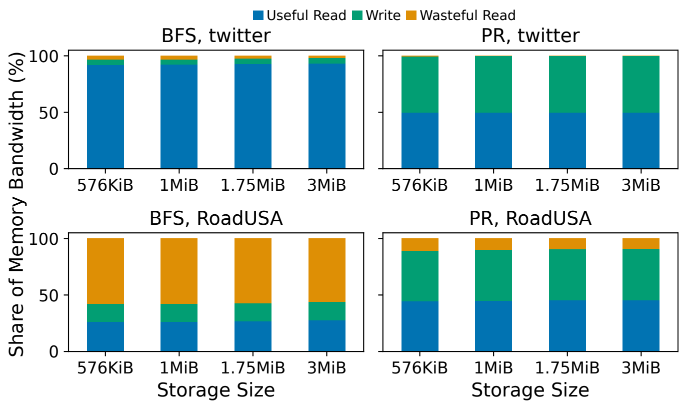


Fig. 10: Vertex memory bandwidth breakdown: useful access (vertices read for reduction or propagation), and wasteful access (inactive vertices read searching for active vertices). The bandwidth distribution is insensitive to the storage size.

work efficiency achieved from lower network traffic. However, both load-balanced and random vertex mapping exhibit lower preprocessing costs than methods for optimizing locality, such as RABBIT.

D. Characterizing the Performance of Interconnect

As we scale up the number of GPN to improve performance or handle larger graphs, the required interconnect bandwidth increases. To analyze the inter-GPN network requirements, we studied the traffic pattern and bandwidth requirements of PE-to-PE communication in the accelerator. We simulated a system with 64 PEs connected with a point-to-point network with no bandwidth restriction.

Figure 9c compares the performance of NOVA while configured with a point-to-point interconnect against the performance of NOVA while configured with a hierarchical interconnect similar to our proposal, using the same configurations in Table II. Figure 9c shows that the performance of NOVA (with the hierarchical interconnect) is similar to the configuration with the ideal point-to-point connection. The results demonstrate that we can scaled-out the number of GPNs using a crossbar without communication bottlenecks.

TABLE IV: Requirements to support WDC12. NOVA, PolyGraph, and Dalorex have 8, 16, and 256–4096 cores/node. Every accelerator is scaled to fit WDC12 graph.

Accelerator	HBM Stacks	DDR Channels	SRAM/ eDRAM	Cores	# of Slices
NOVA	14 (56 GiB)	56 (1TiB)	21 MiB	112	1
PolyGraph	136 (1.088 TiB)	-	4 GiB	2176	15
PolyGraph non-sliced	128 (1 TiB)	-	56 GiB	6400	1
Dalorex	-	-	1 TiB	249661	1

E. Scaling to Terascale Graphs

WDC12 [2] is a hyperlink graph representing 3.5 billion web pages and 128 billion hyperlinks. This is representative of future terascale graph analytics. We compared the resource cost of NOVA with two recent works, PolyGraph [13] and Dalorex [34]. We assumed that all accelerators use the vertex size of 16 bytes and the edge size is 8 bytes. WDC12 requires 53 GiB for vertex and 959.15 GiB for edge capacity. Table IV shows the system requirements to meet the minimum memory for WDC12. We do not consider the additional capacity that PolyGraph requires for temporal partitioning. Dalorex requires 1 TiB of on-chip capacity to support WDC12. Table IV shows that PolyGraph and Dalorex will have extremely high costs (many 100s of HBM stacks or a terabyte of on-chip memory) required to process tera-scale graphs. NOVA still requires significant resources to process large graphs, but by storing the high-capacity data structure in lower-cost memory (storing edges in DDR instead of HBM or SRAM) and vertices in high-performance memory, NOVA scales to large graphs more practically than PolyGraph and Dalorex.

F. FPGA Prototype

We implemented a register transfer level (RTL) model of NOVA to make sure the gem5 model is correct and complete, and ensure that proposed architecture is feasible and practical, in terms of area and power. Table V shows the post synthesis results of 1 GPN (which is a collection of 8PEs) on the Xilinx Alveo U280 FPGA that has both DDR4/HBM memory which are required in our proposal [48]. As described in section IV-C, multiple GPNs could be connected together to realize larger graph processing systems. Alveo U280 has enough resources

TABLE V: Hardware Implementation Details for a single GPN running at 1 GHz. MPU: Message Processing Unit, VMU: Vertex Management Unit, MGU: Message Generation Unit.

Unit	LUT	FF	BRAM	URAM	Power (mW)
8 MPU	6032	7472	16	24	1120
8 VMU	5160	5560	64	64	1396
8 MGU	1640	4840	16	8	752
NoC	3	145	0	0	6
Total	1725 (1.12%)	2379 (0.8%)	12 (4.96%)	96 (7.1%)	3274

to fit 14 GPNs (112 PEs) and process graphs as large as AliGraph (492.9 million vertices and 6.82 billion edges) [58], compared to ScalaBFS [24] can accommodate graphs only as large as Twitter (41.65 million vertices and 1.46B edges) on the same FPGA.

VII. Related Work

Hardware Accelerators: Previous hardware graph accelerators improve performance by creating customized pipelining mechanisms [3], [7], [18], [31], [32], [35], [38], [39] or by decoupling data access and computation [28], [33], [52]. All of these studies focus on improving off-chip memory efficiency for graph processing. Mukkara et al. [30] reduces random off-chip memory accesses using a hardware-accelerated traversal scheduler that allows the system to improve locality. GraphDyns [50] represents a new programming model to extract data dependencies in graph processing dynamically. It uses a load-balanced scheduling mechanism and a specialized prefetcher for off-chip edge data access. ScalaGraph [51] proposes the use of high bandwidth memory and a distributed memory assignment to improve the edge throughput while eliminating the atomic memory accesses. Chronos [3] avoids temporal partitioning and uses an on-chip cache and speculative execution model to avoid coherency overheads. Ozdal [35] defines a custom cache corresponding to each different data type (edge indices, edge data, vertex info, vertex data) of each graph object type to reduce the data access energy. Graphlily [20], ScalaGraph [51], and PolyGraph [13] utilized HBM memory for higher bandwidth access to the edge memory. However, in all these designs, HBM was used only to improve the edge bandwidth throughput. It should be mentioned that in all these studies, the goal has been to improve performance by reducing off-chip memory, whereas in NOVA, we prioritize scalability.

ScalaGraph [51] scales the number of PEs by reducing the complexity of multiple PEs accessing a shared on-chip memory by using a distributed on-chip memory hierarchy among all PEs. In ScalaGraph, vertex information is stored on-chip, and the graph is divided into disjoint subgraphs stored in each PEs dedicated HBM. However, similar to previous work, the performance of ScalaGraph comes from reducing off-chip memory access by using a large on-chip scratchpad.

Recent studies have focused on reducing pre-processing costs and off-chip memory access, while improving data reuse on on-chip memory for GCNs and GNNs by partitioning the graph during runtime. However, these studies utilize graph traversal to identify highly connected parts of graphs (i.e., communities). These accelerators are specific to GCNs and GNNs and are not designed to improve the performance of simpler graph traversal applications like BFS. NOVA, on the other hand, aims to enhance the performance of all graph primitives, including graph traversals.

PIM-based Accelerators: A promising solution to remove the memory wall challenges in the graph workloads is to use processing in the memory (PIM). Some studies rely on emerging memory technologies such as ReRAMs [5] to perform computing in memory in addition to storing data [12], [43], [57]. Other PIM-based architectures use 3D stacked memory technologies, such as Hybrid Memory Cube (HMC) [36] to eliminate the irregular data movement [4], [54], [60]. These accelerators propose using Hybrid Memory Cube and non-volatile memories to store their randomly accessed data structures. In NOVA, depending on the capacity, performance, and available resources, the architect can decide on the vertex or edge memories.

Sofware Frameworks: Software solutions accelerate graph applications by improving data placement. Software platforms such as GraphIt [56] optimize performance by using a variety of techniques, such as loop fusion, loop tiling, and memory reuse, to minimize the number of memory accesses required and reduce the amount of data that needs to be moved around. For distributed systems, GiraphUC [19] uses a barrierless and asynchronous model that removes global synchronization barriers. However, it has high communication costs. Pregel [27] is a model used for large-scale graph computing, but it suffers from communication overhead. Recent work shown by Yin *et al.* introduces Glign [53], which automatically aligns different graph traversals of concurrent queries to maximize graph access sharing. As a result, it can significantly reduce the cache misses compared to other systems.

VIII. Conclusion

In this paper, we present NOVA, a scalable graph accelerator with DRAM-based work management. NOVA achieves high performance for large graphs by creating a balanced system. NOVA is an appropriate design for deployments where resources are limited. Compared to previous hardware accelerators, NOVA uses off-chip memory bandwidth for both sequential edge read and random vertex updates. The throughput in NOVA for different graph sizes will remain constant, while other studies require temporal partitioning, which results in degradation of performance for large graphs. In addition, we propose a scaled-out mechanism that allows scaling out to multiple cores without communication overheads. Combining these two insights, the scalable NOVA architecture charts the path toward tera-scale graph analytic accelerators.

Acknowledgements

We thank the reviewers of HPCA 2025 for their invaluable feedback. This work was supported by NSF grants 1925724 and 2225882, ARO award W911NF1910470, and Los Alamos National Laboratory.

References

[1] 9th dimacs implementation challenge: Shortest paths. [Online]. Available: http://www.diag.uniroma1.it/challenge9/

[2] "WDC - Hyperlink Graphs." [Online]. Available: http://webdatacommons.org/hyperlinkgraph/

[3] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1247–1262.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[5] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[6] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 22–31.

[7] A. Ayupov, S. Yesil, M. M. Ozdal, T. Kim, S. Burns, and O. Ozturk, "A template-based design methodology for graph-parallel hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 420–430, 2017.

[8] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 203–214.

[9] S. Beamer, K. Asanović, and D. Patterson, "Gail: The graph algorithm iron law," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 2015, pp. 1–4.

[10] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[11] Broadcom, "Bcm78900 series," https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78900-series, accessed: 2024-07-28.

[12] N. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan, "Gaas-x: Graph analytics accelerator supporting sparse data representation using crossbar architectures," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 433–445.

[13] V. Dadu, S. Liu, and T. Nowatzki, "Polygraph: Exposing the value of flexibility for graph processing accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 595–608.

[14] T. Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: A mechanism for integrated communication and computation," in *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 256–266.

[15] P. Erdős, A. Rényi *et al.*, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[16] M. Fariborz, M. Samani, T. O'Neill, J. Lowe-Power, S. B. Yoo, and V. Akella, "A model for scalable and balanced accelerators for graph processing," *IEEE Computer Architecture Letters*, vol. 21, no. 2, pp. 149–152, 2022.

[17] T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M. Herbordt, Y. Lin, and A. Li, "I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO-54: 54th annual IEEE/ACM international symposium on microarchitecture*, 2021, pp. 1051–1063.

[18] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[19] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[20] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[21] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[22] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 591–600.

[23] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: an approach to modeling networks." *Journal of Machine Learning Research*, vol. 11, no. 2, 2010.

[24] C. Liu, Z. Shao, K. Li, M. Wu, J. Chen, R. Li, X. Liao, and H. Jin, "Scalabfs: A scalable bfs accelerator on fpga-hbm platform," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 147–147.

[25] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[26] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

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

[28] A. Manocha, T. Sorensen, E. Tureci, O. Matthews, J. L. Aragón, and M. Martonosi, "Graphattack: Optimizing data supply for graph applications on in-order multicore architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, pp. 1–26, 2021.

[29] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 1–39, 2015.

[30] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 1–14.

[31] Q. M. Nguyen and D. Sanchez, "Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 596–608.

[32] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1064–1077.

[33] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, "Tiny but mighty: Designing and realizing scalable latency tolerance for manycore socs," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 817–830.

[34] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi, "Dalorex: A data-local program execution and architecture for memory-bound applications," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 718–730.

[35] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 166–177, 2016.

[36] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, 2011, pp. 1–24.

[37] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *ACM SIGPLAN Notices*, vol. 46, pp. 12–25, 6 2011. [Online]. Available: http://iss.ices.utexas.edu/galois

[38] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 908–921.

[39] S. Rahman, M. Afarin, N. Abu-Ghazaleh, and R. Gupta, "Jetstream: Graph analytics on streaming data with event-driven hardware accelerator," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1091–1105.

[40] M. Samani, "Methodologies for evaluating memory models in gem5," Ph.D. dissertation, UC Davis, 2021.

[41] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[42] J. E. Smith, “Decoupled access/execute computer architectures,” *ACM SIGARCH Computer Architecture News*, vol. 10, no. 3, pp. 112–119, 1982.

[43] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “Graphr: Accelerating graph processing using reram,” vol. 2018-Febru. IEEE Computer Society, 3 2018, pp. 531–543.

[44] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, “Graphmat: High performance graph analytics made productive,” *arXiv preprint arXiv:1503.07241*, 2015.

[45] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, p. 103111, aug 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[46] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, “Gunrock: A high-performance graph processing library on the gpu,” in *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, 2016, pp. 1–12.

[47] Z. Wang, H. Huang, J. Zhang, and G. Alonso, “Shuhai: Benchmarking high bandwidth memory on fpgas,” in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 111–119.

[48] Xilinx, “Amd alveo u280 product brief,” 2024, accessed: 2024-08-01. [Online]. Available: https://www.xilinx.com/publications/product-briefs/alveo-u280-product-brief.pdf

[49] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, “Hygcn: A gcn accelerator with hybrid architecture,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[50] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, “Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: Association for Computing Machinery, 2019, p. 615628. [Online]. Available: https://doi.org/10.1145/3352460.3358318

[51] P. Yao, L. Zheng, Y. Huang, Q. Wang, C. Gui, Z. Zeng, X. Liao, H. Jin, and J. Xue, “Scalagraph: A scalable accelerator for massively parallel graph processing,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 199–212.

[52] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, “An efficient graph accelerator with parallel data conflict management,” in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–12.

[53] X. Yin, Z. Zhao, and R. Gupta, “Glign: Taming misaligned graph traversals in concurrent graph processing,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2022, pp. 78–92.

[54] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, “Graphp: Reducing communication for pim-based graph processing with efficient data partition,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 544–557.

[55] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2091–2100, 2013.

[56] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, “Graphit: A high-performance graph dsl,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[57] M. Zhou, M. Imani, S. Gupta, Y. Kim, and T. Rosing, “Gram: graph processing in a reram-based computational memory,” in *IEEE Asia and South Pacific Design Automation Conference*, 2019.

[58] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, “Aligraph: A comprehensive graph neural network platform,” 2019. [Online]. Available: https://arxiv.org/abs/1902.08730

[59] X. Zhu, W. Chen, W. Zheng, and X. Ma, “Gemini: A computation-centric distributed graph processing system.” in *OSDI*, vol. 16, 2016, pp. 301–316.

[60] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, “Graphq: Scalable pim-based graph processing,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.