GraphWave: A Highly-Parallel Compute-at-Memory Graph Processing Accelerator

Jinho Lee*, Burin Amornpaisannon*, Tulika Mitra, Trevor E. Carlson
National University of Singapore

***Abstract*—The fast, efficient processing of graphs is needed to quickly analyze and understand connected data, from large social network graphs, to edge devices performing timely, local data analytics. But, as graph data tends to exhibit poor locality, designing both high-performance and efficient graph accelerators have been difficult to realize.**

In this work, GraphWave, we take a different approach compared to previous research and focus on maximizing accelerator parallelism with a compute-at-memory approach, where each vertex is paired with a dedicated functional unit. We also demonstrate that this work can improve performance and efficiency by optimizing the accelerator's interconnect with multi-level multi-casting to minimize congestion. Taken together, this work achieves, to the best of our knowledge, a state-of-the-art efficiency of up to 63.94 GTEPS/W with a throughput of 97.80 GTEPS (billion traversed edges per second).

I. Introduction

Graph processing is an extremely important technique that is used by many different algorithms to analyze relational data [1], [2]. While the most recognizable techniques have been demonstrated in large datacenter workloads, such as in social-media companies [3], graph data processing is also becoming an important emerging workload for high-performance edge systems [4], [5], [6]. But, understanding and processing graph data in a timely and energy-efficient way is extremely difficult with traditional processor designs because the latency needed to access graph data can be extremely high due to the poor locality characteristics of graph data [7].

To address the need for higher graph processing performance, a variety of solutions have been proposed. A number of approaches explore streaming data from DRAM or other single-memory-sources, such as modern High-Bandwidth Memory (HBM) technologies. These systems [8], [9] pair processor or accelerator designs through a relatively high-bandwidth channel to a very large capacity memory subsystem. Despite the high-bandwidth, the overall parallelism and ability to access many sparse elements in parallel are still limited.

HMC or Hybrid Memory Cube-based solutions [10], [11], [12] were introduced as a way to address these concerns by increasing the effective parallelism to a slice of memory (called a vault), each paired with a memory controller and a specialized hardware accelerator. This allows for an increase in parallelism with the number of vaults in the system. Nevertheless, with millions of edges per vault, the overall parallelism of these accelerators is still limited.

Instead, in this work, we maximize the parallelism of graph processing by storing the graph data for each vertex or node together with its own processing element and edge storage data. In this vertex-centric approach, we look to maximize the potential parallelism of the system, where in the best case, each vertex could be processing data in parallel with all others. By increasing the parallelism to the extreme, we aim to demonstrate how this system can overcome the bandwidth and parallelism limits of previous designs.

When exploring the limits of the parallelism of graph processing, we see that the **limiting bottleneck of the system now becomes the graph data itself, and how it is mapped to the accelerator**. In fact, there are three key challenges that occur when optimizing graph processing: (a) workload imbalance, which often occurs in real-world graphs that exhibit power-law properties [13], e.g., a few vertices have a very high degree while most of other vertices have a low degree; (b) data locality, which tends to be poor for graphs; and (c) communication costs which need to be handled appropriately to minimize latency and overhead. Only after addressing each of these challenges, can we maximize efficiency and throughput.

To address these challenges, we propose a (1) **highly-parallel per-vertex processing accelerator: GraphWave**, which co-locates a functional unit with the data for each vertex. This enables a level of parallelism that has not been seen in prior works. In addition, to address the workload imbalance and communication concerns, we propose an (2) **efficient multi-level data multi-casting technique** to minimize network overhead and network congestion. To the best of our knowledge, this work presents one of the highest levels of efficiency, achieving up to 63.94 Giga-Traversed Edges Per Second per Watt (GTEPS/W) with a throughput of 97.80 GTEPS.

In the rest of this paper, we present our parallel, multicast-enabled accelerator in detail. In Section II, we introduce the background of this work and in Section III, we present our graph processing approach. We present the experimental setup for this work in Section IV, and evaluate the results in Section V. In Section VI, related works are presented, and we conclude with Section VII.

II. Background

The vertex-centric processing approach is an established graph processing abstraction that is suitable for distributed, parallel graph analysis [14]. Each vertex collects incoming messages, updates its value, and propagates the updated value to its neighbors. Algorithm 1 shows the vertex-centric graph processing abstraction. In the outer loop, here called a superstep, all the vertices propagate their values to their neighboring vertices,

*The authors contributed equally to this work.

978-3-9819263-6-1/DATE22/©2022 EDAA

Algorithm 1 Vertex-centric graph processing

Input: directed graph $G = (V, E)$

1: $activeVertices \leftarrow V$
2: $superstep \leftarrow 0$
3: **while** $activeVertices$ **do** ▷ One superstep
4: **for all** $v \in V$ **do** ▷ In parallel
5: $v.new\_val = rcvMsgs(v)$ ▷ Reduce
6: $sendMsgs(v.dsts, v.val)$ ▷ Propagate
7: **end for**
8: **for all** $v \in V$ **do** ▷ In parallel
9: $v.val = v.new\_val$ ▷ Apply
10: **end for**
11: $superstep \leftarrow superstep + 1$
12: **end**

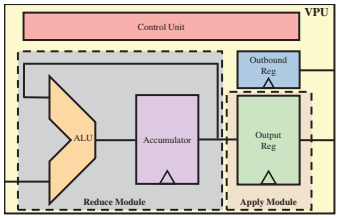
and collect messages from others. Each vertex first receives the messages from its inbound neighbors and *Reduces* the messages to update its own value (line 5). Meanwhile, it *Propagates* messages to its outbound neighbors (line 6). After finishing sending and collecting the messages, the vertices update their values with the new value generated in this superstep (line 9). Then, the next superstep can begin (line 11). In this approach, all the vertices can *Reduce* and *Propagate* messages in parallel (line 4-6), which makes this algorithm suitable for processing graphs in a distributed manner [14].

Lumsdaine et. al [7] present and analyze the challenges in parallel graph processing. In this work, we have chosen to address the most common and relevant challenges observed in the vertex-centric approach, with a description of each of those main challenges below.

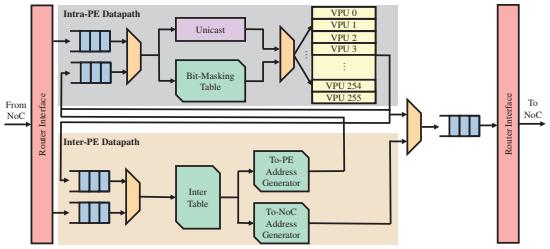
1) *Workload imbalance*: The vertices are processed in parallel (line 4-6 and 8-9 in Algorithm 1). However, we can only move on to the next superstep after all the vertices finish propagating and receiving messages, and updating their values (line 11). In this case, existence of high fan-out or high fan-in vertices can result in a longer time to proceed to the next superstep.
2) *Data locality*: If the data propagation is done using a shared memory (line 6), each vertex has to iterate through its neighbors' pointers ($v.dsts$) to send its value ($v.val$). If the memory addresses of the neighbors are not sequential, this can cause frequent random memory accesses, which leads to the problem described below.
3) *Communication cost*: The frequent random memory accesses due to the poor data locality can cause cache misses and off-chip accesses that lead to long latency and poor energy efficiency.

III. GraphWave Methodology

In this section, we present the GraphWave methodology, a combination of a novel highly-parallel microarchitecture and software techniques that, when combined, exposes parallelism at the per-vertex level. The GraphWave accelerator supports efficient message propagation where a vertex sends a single packet and it is multi-casted to all its destinations no matter how many neighbors it has. In this way, it tackles the workload imbalance issue. The packet routing path is statically determined and stored in distributed routing tables that can be easily accessed via local memories. Due to the fast, local access of these tables, communication cost is significantly reduced enabling GraphWave to attain both high performance and energy efficiency.



(a) A **Vertex Processing Unit (VPU)** hardware represents a vertex in a graph. It *reduces* received messages and *applies* changes when all messages are collected. The outbound and output registers together are used to transfer data to other components outside the VPU.



(b) A **Processing Element (PE)** consists of 256 VPUs and tables to store routing information. In the *bit-masking table*, each entry has 256 masking bits to specify the destination ports. *inter table* contains the routing information to support inter-PE communication. The PEs are connected using a Network on Chip (NoC).

Fig. 1: **GraphWave Microarchitecture**. A VPU reduces received messages and directly applies the update within it. A message can be multi-casted to multiple VPUs in a PE. The PEs are interconnected through NoC and all vertices communicate with each other in parallel.

A. GraphWave Microarchitecture

Vertex Processing Unit: Reduce and Update. A Vertex Processing Unit (VPU) is a dedicated hardware component for processing requests destined for a single vertex. It contains a *Reduce* module that processes received messages and reduces the results immediately, and an *Apply* module that updates its state when the *Reduce* stage is completed. The *Apply* module also propagates its data outside of the VPU.

The design of the VPU is shown in Figure 1a. The *Reduce* module consists of an ALU and accumulator register. When a message is received, the ALU calculates a new value based on the selected graph algorithm and the value of the accumulator, and then updates the accumulator. The *Apply* module is comprised of an output register that only updates its value with the output from the accumulator at the end of the *Reduce* step. While the *Reduce* module receives messages, the output register in the *Apply* module and outbound register propagate its data to outbound destinations through on-chip connections. In this design, the communication overhead between phases is reduced as the previous accumulated value remains at the ALU, and

its vertex state can be updated using the accumulator directly connected to it.

Processing Element: Simultaneous message propagation. The Processing Element (PE) is designed to support message passing between VPUs in a massively parallel and energy efficient way. The design, shown in Figure 1b, consists of two datapaths that handle intra-PE and inter-PE message passing, respectively. The PEs are interconnected via a NoC. A message coming from the NoC goes to either one of the paths depending on whether the message is required to be sent to other VPUs in other PEs. Note that this methodology is independent from the selection of the NoC architecture.

The intra-PE datapath consists of 256 VPUs, a unicast unit, and a bit-masking table. It handles messages that are sent to the VPUs in the PE[1]. The bit-masking table, used to handle a message with multiple destinations, contains masking bits of destination VPUs to receive the message. When a message has the destination address at the m^{th} entry of the bit-masking table, it retrieves the entry containing 256 masking bits. Then, the message is delivered to the VPUs where the masking bits are high by connecting each VPU's write enable signal to its associated masking bit. Thus, with this technique, a single message can be propagated up to 256 vertices inside this PE without random memory accesses. When a message has only one destination, the unicast unit directly sends it to the particular destination VPU. Also, the VPUs generate messages that connect to the inter-PE datapath, intra-PE datapath or NoC depending on the type of the messages.

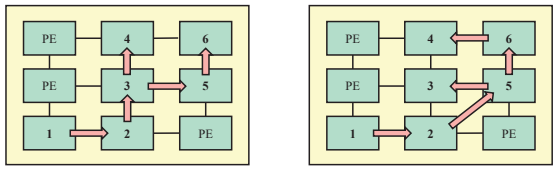
The inter-PE datapath handles input and output messages through the NoC. A message that arrives at this datapath contains an entry number to access the *inter table* which contains its neighbors' destinations. The output of this table is used to generate new addresses from two address generators. The to-PE address generator is responsible for generating an address and forwarding this message with the address to the intra-PE datapath to update the VPUs. At the same time, the to-NoC address generator generates new addresses for the message to be sent to other PEs.

As mentioned earlier, our goal is to address the main graph processing challenges presented in Section II. First, (1) workload imbalance issues are mitigated by moving the message propagation workloads from the vertices to the intra-PE and inter-PE datapaths. In this way, regardless of the number of outbound edges, a VPU sends only one message and it is multi-casted to its neighbors in parallel. Next, (2) the frequent irregular memory accesses typically seen during graph processing are minimized by storing data where it is easily accessible by compute units (compute-at-memory). Finally, (3) communication latency is significantly reduced through the use of our parallel multicasting methodology.

B. Inter-PE message propagation and congestion avoidance

To utilize the hardware design efficiently, our software mapper pre-determines all the routing paths and generates the mapping and routing tables based on an input graph. In this subsection, we explain how our mapper alleviates the network congestion to achieve high throughput and efficiency.

[1]For graph algorithms requiring weight values, a weight table can be added between the bit-masking table and the array of VPUs.



(a) When the multi-casting is applied to minimize the hop count following the arrows, it that can cause traffic congestion at PE 3 as it is at the center of the NoC.

(b) To mitigate the congestion issue, the messages are propagated following the alternative route instead by having the packets arrive at the center last.

Fig. 2: Inter-PE multicasting techniques.

Inter-PE multicasting. When numerous PEs send packets simultaneously, this can potentially flood the NoC and adversely affect accelerator throughput. In addition, when many packets are required to be sent to the NoC, the PEs can suffer from a contention at the output ports. To overcome these issues, an inter-PE multi-casting technique is proposed.

When a vertex has destinations to multiple PEs, a message is sent through the inter-PE datapath. The inter-PE datapath then sends packets with the message only to the physically neighboring next-hop PEs. At that point, the PEs that receive the packets relay them to their neighbors. This is repeated until all destination PEs receive the original message. To find the multi-casting path of the packet, the packet arrival order between the destination PEs is determined first. The PEs are sorted based on the hop distance from the source PE. Then, from the furthest PE, we recursively find a relay PE and store the routing information in the inter-PE datapath of the relay PE. For example, in Figure 2a, a source vertex is at PE 1 and its destinations are in PE 1-6. The PEs are sorted in the order of 6, 5, ..., 1. Then, PE 6 picks PE 5 as the relay PE, PE 5 picks PE 3, and so on. When the source vertex propagates a message, it is propagated following the arrows based on the routing tables. The messages start propagation from PE 1. When it reaches PE 2, the message is propagated to internal destination vertices and a packet is sent to the next destination, PE 3. These steps are repeated until all the destination PEs receive the message.

The relay PE selection is done to minimize the travel distance of packets. However, the router in the center of the NoC (PE 3 in the example) can suffer from heavy multi-casting traffic. It can also cause output port contention of the center PEs. To alleviate this network and output port contention at the center region, we propose a load balancing technique.

Congestion avoidance in Inter-PE multicasting. The PEs in the center area tend to receive and send more packets as they are the bridge between the PEs on the periphery when the relay PE selection method only considers minimizing the packets' travel distance. These PEs can be a bottleneck that reduces the overall throughput by making the supersteps longer. To alleviate this traffic congestion issue, the mapper can help to plan alternate routes for a subset of the packets.

We solve this issue by changing the packet arrival order

TABLE I: Input graph dataset details. D_{avg} is the average vertex degree, or the number of outbound edges.

	$\|V\|$	$\|E\|$	D_{avg}	domain
gnutella05 (GT) [16]	9k	31k	3.4	web
wiki-vote (WV) [16]	7k	103k	14.7	web
grid-yeast (GY) [17]	6k	314k	52.3	biology
spam-detection (SD) [17]	9k	506k	56.2	web
reality (RE) [17]	7k	9,429k	1,344.4	social

among the destination PEs. To choose alternative routes, the destination PEs are sorted depending on $travel\ distance + \#\ of\ relay\ packets$ instead of considering the $travel\ distance$ alone. In our example (Figure 2), if PEs 3 and 4 are handling many relay messages, based on the new metric, the PEs can be sorted in the order of [3, 4, 6, 5, 2, 1]. Then, from PE 3, the destination PEs recursively look for the best relay PE to minimize the new metric and the packets are multicasted following the arrows in Figure 2b.

In-flight reduce operation. When many VPUs in a single PE send messages to a potentially high fan-in VPU at a remote PE, we can reduce the number of NoC messages by first reducing the data inside the local PE. We take advantage of local, unmapped VPUs to collect and distribute outgoing results. After merging the messages, the VPU generates a single, final message and transfers it to the target PE. By distributing the message reduction closer to the source vertices, this technique helps to reduce the amount of messages in the NoC, mitigating network congestion and port contention of the PEs, especially at high fan-in nodes[2]

IV. Experimental Setup

GraphWave is implemented in SystemVerilog, uses the OpenSMART NoC [18] implemented in Bluespec, and is synthesized using Synopsys Design Compiler version 2019.03 targeting a commercial 22 nm technology node with 6×7 processing elements at 200 MHz to support all the input graphs. The bit-masking table, inter table, and the address tables inside the to-PE address generator and to-NoC address generator are implemented using single-port ultra-low-power SRAMs. We use Synopsys VCS-MX 2015.09 for gate-level simulation, and Synopsys PrimePower 2019.03 for power evaluation. We apply power gating to disable unused PEs and VPUs when running graphs that are smaller than the maximum capacity. Note that the rest of the routers remain enabled to facilitate data transfer between PEs. Evaluations are conducted after the data has been loaded into the accelerator.

Input graphs & target algorithms. We use realistic graphs from different domains with different edge counts to demonstrate their impact on throughput and efficiency (See Table I). The number of vertices supported is limited by the hardware configuration, or 10,752 (256 VPUs in each of the 6×7 PEs). Extensions to this work could include stream graph processing. We implement the VPUs to support the commonly used graph processing algorithms listed in Table II. Each VPU is equipped with functional units to support all of the selected algorithms.

[2]Previous work [15] has proposed in-flight reduction for server workloads.

TABLE II: Target algorithms: Page Rank (PR), Breadth First Search (BFS), and Connected Components (CC).

	Reduce	Apply
PR[19]	$v.acc = v.acc + m.val$	$v.val = \frac{(\alpha+(1-\alpha)\cdot v.acc)}{v.degree}$
BFS	$v.acc = Min(v.acc, m.val)$	v.val = v.acc
CC	$v.acc = Min(v.acc, m.val)$	$v.val = Min(v.acc, v.val)$

TABLE III: Throughput and power efficiency of GraphWave. This work tends to show higher performance and efficiency for the graphs with larger average degree. All unused PEs and VPUs in each graph are power gated for efficiency.

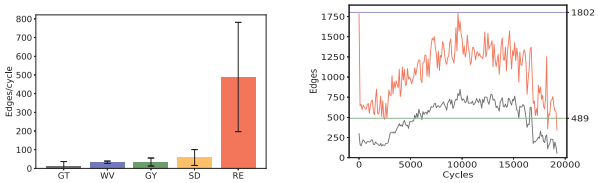
Input graph	Edges/cycle	GTEPS	GTEPS/W
GT	13.17	2.63	1.36
WV	32.47	6.49	4.02
GY	33.97	6.79	4.99
SD	58.14	11.63	5.67
RE	488.99	97.80	63.94

V. Evaluation

Performance and power efficiency. Table III shows the average number of edges per cycle, throughput, and efficiency for GraphWave, which has an area of 91.88 mm^2. It achieves 489 edges per cycle for the reality (RE) graph, which we configured to use a subset of the accelerator, 30 PEs with 7,057 VPUs. This configuration performs at 97.80 GTEPS with a power efficiency of 63.94 GTEPS/W. The average throughput achieved for each input graph dataset with their standard deviation is shown in Figure 3a. The performance of the accelerator depends on the size of an input graph. Bigger graphs tend to show higher throughput as they have more edges to be traversed. In our methodology, the messages are propagated in parallel, so that the higher output edge degree can directly translate to higher performance. This is different when compared to traditional approaches in which a graph with a higher degree (e.g., RE in Table. I) can exacerbate workload imbalance, data locality, and communication overhead, reducing overall performance.

Figure 3b shows the maximum and average number of edges computed over time on the reality (RE) graph in 100 cycle windows. GraphWave attains a peak performance of 1,802 edges/cycle with an average performance of 489 edges/cycle. Intra-PE multi-casting provides 36 times higher throughput than unicast solutions, and the version without both intra-PE and inter-PE has 39 times lower throughput than the version that utilizes both. Intra-PE multi-casting and inter-PE multi-casting using our mapping algorithm enables the accelerator to achieve high performance throughout the run. Note that, at the second half of the run, the throughput goes down as some PEs finish collecting and emitting messages before others. This is due to the synchronous nature of the vertex-centric graph processing algorithm shown in Algorithm 1, that requires all PEs to complete before moving to the next superstep.

Number of packets vs. number of edges. We introduce inter-PE multicasting in order to achieve higher throughput by reducing the congestion on the NoC. Figure 4 shows the number of messages (number of edges) and the generated packets that are delivered between PEs. It shows how well



(a) The number of edges per cycle achieved for each input graph dataset. The error bars show the standard deviation of the runs. The reality (RE) graph shows the highest throughput compared to other graphs.

(b) Throughput over time in a single superstep on the reality (RE) graph. The red and black lines represent the maximum and average values over time. The blue and green horizontal lines represent the maximum and average values of the superstep.

Fig. 3: Traversed edges per cycle.

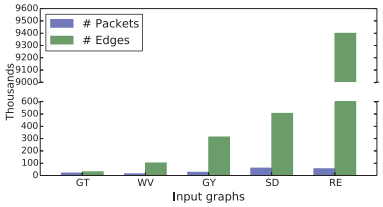


Fig. 4: Number of edges vs. number of packets, in thousands.

GraphWave scales with the number of edges. Specifically, in the reality graph (RE), to send 9.4M messages, GraphWave sends only around 200k packets, eliminating 97.87% of messages from being transmitted on the NoC.

Power breakdown. Table IV illustrates the power breakdown of all the components in GraphWave based on the reality (RE) graph. The accelerator consumes approximately 1.54 W when 30 PEs and 7,057 VPUs are enabled. The unused PE and VPUs are power gated to save power. The majority of the power is consumed by the VPUs which represents 68.96 % of the total; one-fifth of the power is taken by the SRAMs. The lowest and highest power consumption are at 1.36 W and 2.05 W from the grid-yeast and spam-detection graphs, respectively.

TABLE IV: GraphWave component power breakdown based on the reality (RE) graph.

		Power(mW)		
	#	Static	Dynamic	Total
Control Logic	-	0.16	24.74	24.90 (1.62 %)
FIFO	5×30	0.12	42.86	42.98 (2.80 %)
Network	-	0.21	120.86	121.07 (7.88 %)
SRAM Table	4×30	0.21	287.72	287.93 (18.74 %)
VPU	7057	20.71	1,039.09	1,059.80 (68.96 %)
Total	-	21.41	1,515.27	1,536.68

TABLE V: Storage required for the input graphs. Internal multi-casting, external multi-casting and total sizes are listed in KB. The storage saving shows the percentage savings between original and compressed size.

Graph	Internal Multi.	External Multi.	Total	Savings (%)
GT	146.34	187.11	333.45	1.23 %
WV	498.31	255.40	753.71	31.45 %
GY	1,521.56	687.04	2,208.60	23.84 %
SD	2,559.59	1,165.71	3,725.30	30.19 %
RE	5,402.21	1,576.91	6,979.13	83.93 %

TABLE VI: Scalability with large degree synthetic graphs.

$\|V\|$	$\|E\|$	D_{avg}	GTEPS	# packets	Storage savings
4k	0.04M	10	1.74	27,957	-25.69 %
4k	0.40M	100	8.33	59,548	37.95 %
4k	4.00M	1,000	83.80	60,000	93.80 %
4k	8.00M	2,000	166.61	60,000	96.82 %
4k	16.00M	3,999	337.01	60,000	98.41 %

Storage usage. In GraphWave, the routing information is stored in the distributed routing tables. The proposed data representation not only enables efficient message multicasting, but it also saves storage by compressing the routing information into masking bits. Table V shows the data compression rate compared to the original graph size. The graphs with a high number of edges gain more from this technique. For the reality graph (RE) that has the largest number of edges, the technique can compress the information to only one-fifth of the original size. This high compression rate can be beneficial in larger scale graphs. Specifically, when an input graph has more vertices than the existing VPUs, we might have to stream the graph information in and out dynamically. In that case, the graph information streaming can be the bottleneck. By compressing the graph information, we can shorten the streaming time.

Scalability with respect to the degree. In Table VI, the scalability test results are shown on synthetic graphs generated using SNAP [20]. As the average degree grows, the number of packets becomes stable. In the worst case, in which each vertex has to propagate its message to all existing PEs, the maximum number of packets is limited by $number\ of\ VPUs \times number\ of\ PEs$, which is significantly smaller than the number of edges in large graphs. Furthermore, the storage taken for the routing tables is limited for the same reason. Thus, the compression rate goes up as the graph has more edges. It shows that the proposed methodology is suitable for processing high degree graphs efficiently.

VI. Related work

Due to the increasing importance and interest in graph processing, various types of graph accelerators are presented below. Table VII shows the summary of state-of-the-art graph accelerators along with their performance and power efficiency.

ASIC based approaches. Graphicionado [4] adds an eDRAM scratchpad which stores the randomly accessed data to avoid long latency access to the main memory and save the limited bandwidth. GraphPulse [5] introduces parallel dataflow-style graph processing. When an update occurs, events are generated and sent to destination vertices. To alleviate the communication cost, the events that are heading to the same destination are coalesced in-flight. The event coalescing is done using an event queue that is searched and updated for every new event, which takes most of the power in this design. PolyGraph [6] points out the importance of flexibility in graph processing and introduces their microarchitecture to offer the flexibility in graph processing. While they achieve high performance, their results fall below GraphWave because our work supports both reductions and a higher level of parallelism.

PIM based vertex-centric approaches. The works [12], [11], [10] utilize the Hybrid Memory Cube (HMC), which

TABLE VII: State of the art graph accelerators.

	Throughput (GTEPS)	Power (W)	Efficiency (GTEPS/W)	Tech (nm)	Arch.
GraphWave (this work)	97.8	1.5(1)	63.94	22	ASIC
PolyGraph [6]	60.0	2.3(2)	26.1	28	ASIC
GraphH [12]	350	29.1(3)	12.03(3)	-	HMC(7)
GraphPulse [5]	27.9(4)	8.9	3.13(4)	28	ASIC
HitGraph [21]	4.9	7.5	1.09	20	FPGA
Graphicionado [4]	4.5	7 (5)	0.64	<28	ASIC
ThunderGP [22]	6.5	43	0.15	16	FPGA
Tesseract [10]	-	21.2	-	-	HMC(6)
GraphQ [11]	-	-	-	-	HMC(7)

(1) A power range from 1.36 to 2.05 W is due to active PEs in use. Maximum efficiency for RE input. (2) Excludes main memory energy. (3) Approximated power with density (128.7 mW/mm^2) times area of [10]. (4) Approximated, work [5] reports 6.2× speedup from previous work [4]. (5) Excludes DRAM controller power. (6) HMC 1.0. (7) HMC 2.1.

provide high bandwidth and density for graph processing. By equipping small processors inside the HMC, they can access and update graph data more quickly in the memory compared to other approaches using off-chip memories. In Tesseract [10], a vertex-centric approach, the vertices send messages to each other and update their values inside the HMC. In the HMC vaults, an in-order core handles the computations for the nodes and updates their values without reaching the processor out of the HMC, reducing the distance between the data and the computation components. GraphQ [11] shows higher throughput by batching the inter-cube data transfers for more efficient communication between cubes. GraphH [12] adds buffers in HMC vaults to reduce random memory accesses and introduces a load balancing algorithm. In GraphWave, the proposed methodology makes the distance between computation and storage components even closer by storing the vertices' status in the vertex processing units.

FPGA based edge-centric approaches. These techniques utilize flexible hardware and its large memory bandwidth to achieve high throughput. HitGraph [21] utilizes abundant controllable on-chip memory and dense programmable logic elements. It focuses on reducing global memory accesses by buffering vertex information and combining updates in PEs before sending the updates to the global memory. ThunderGP [22] pipelines the data propagation stage and the reduce stage so that they can mitigate the communication overheads between the stages. The updates are directly gathered in on-chip storage and updated to the global memory in batches. In our proposal, instead of increasing the memory bandwidth utilization, GraphWave completely removes global memory accesses and maximizes parallelism using efficient data propagation.

Software based vertex-processing. Pregel [14] is a fully software-based approach with an API. The users can input the graph and the behavior of each vertex for processing. A SW-HW co-design BDFS-HATS [8] introduces BDFS that determines the graph traversal order to improve the temporal locality and a prefetcher called HATS to bring the data earlier utilizing the temporal locality.

VII. Conclusion

In this paper, we propose GraphWave, a hardware/software codesigned graph accelerator. We enable high performance and efficiency through (1) a novel highly-parallel compute-at-memory microarchitecture, (2) an on-chip, multi-level data multicasting technique and (3) a network congestion avoidance method to avoid NoC hotspots. This work scales with the number of edges in the graph, which can be different from traditional approaches where a large number of edges lead to large communication overheads. GraphWave achieves one of the highest levels of efficiency, achieving up to 63.94 GTEPS/W with a throughput of 97.8 GTEPS.

Acknowledgements

This research is supported by the Singapore National Research Foundation Award NRF2018NCR-NCR002 and Competitive Research Programme Award NRF CRP23-2019-0003.

References

[1] K. J. Burch, "Chapter 8 - chemical applications of graph theory," in *Mathematical Physics in Theoretical Chemistry*, 2019, pp. 261–294.
[2] G. George and S. M. Thampi, "A graph-based security framework for securing industrial iot networks from vulnerability exploitations," *IEEE Access*, vol. 6, pp. 43 586–43 601, 2018.
[3] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at Facebook-scale," *VLDB*, vol. 8, no. 12, p. 1804–1815, Aug. 2015.
[4] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*, 2016.
[5] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "GraphPulse: An event-driven hardware accelerator for asynchronous graph processing," in *MICRO*, 2020.
[6] V. Dadu, S. Liu, and T. Nowatzki, "PolyGraph: Exposing the value of flexibility for graph processing accelerators," in *ISCA*, 2021.
[7] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing." *Parallel Processing Letters*, vol. 17, pp. 5–20, Mar. 2007.
[8] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *MICRO*, 2018.
[9] G. M. Slota and S. Rajamanickam, "Experimental design of work chunking for graph algorithms on high bandwidth memory architectures," in *IPDPS*, 2018.
[10] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.
[11] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable PIM-based graph processing," in *MICRO*, 2019.
[12] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "GraphH: A processing-in-memory architecture for large-scale graph processing," *TCAD*, vol. 38, no. 4, pp. 640–653, 2019.
[13] M. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary Physics*, vol. 46, no. 5, p. 323–351, 2005.
[14] 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 *SIGMOD*, 2010.
[15] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi, "Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction," in *COMHPC*, 2016.
[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.
[17] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.
[18] H. Kwon and T. Krishna, "OpenSMART: Single-cycle multi-hop noc generator in bsv and chisel," in *ISPASS*, 2017.
[19] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *WWW*, 1998.
[20] J. Leskovec and R. Sosič, "SNAP: A general-purpose network analysis and graph-mining library," *TIST*, vol. 8, no. 1, p. 1, 2016.
[21] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "HitGraph: High-throughput graph processing framework on FPGA," *TPDS*, vol. 30, no. 10, pp. 2249–2264, 2019.
[22] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "ThunderGP: HLS-based graph processing framework on FPGAs," in *FPGA*, 2021.