A Multicore GNN Training Accelerator

Sudipta Mondal, Ramprasath S., Ziqing Zeng, Kishor Kunal, and Sachin S. Sapatnekar
University of Minnesota, Minneapolis, MN, USA

Abstract—**Graph neural networks (GNN) are vital for analytics on real-world problems with graph models. This work develops a multicore GNN training accelerator and develops multicore-specific optimizations for superior performance. It uses enhanced multicore-specific dynamic caching to circumvent the costs of irregular DRAM access patterns of graph-structured data. A novel feature vector segmentation approach is used to maximize on-chip data reuse with high on-chip computation per memory access, reducing data access latency, using a machine learning model for optimal performance. The work presents a major advance over prior FPGA/ASIC GNN accelerators by handling significantly larger datasets (with up to 8.6M vertices) on a variety of GNN models. On average, training speedup of 17× and energy efficiency improvement of 322× is achieved over DGL on a GPU; a speedup of 14× with 268× lower energy is shown over GPU-based GNNAdvisor; and 11× and 24× speedups are obtained over ASIC-based Rubik and FPGA-based GraphACT.**

I. Introduction

In recent years, graph neural networks (GNNs) have achieved unprecedented success on many real-life problems (recommender systems, IC design, embedded sensing, e-commerce, etc.). To accelerate the GNN inference several works have been proposed (GNNIE [1], HyGCN [2], AWB-GCN [3], GNNerator [4], BlockGNN [5], DyGNN [6], BoostGCN [7], etc.) for small- to medium-scale graph workloads. However, a well-trained model is a prerequisite for efficient inference.

Energy-efficient and scalable acceleration of GNN training is an open problem that involves several major challenges:
(i) *High computation and communication costs*: GNN training is more compute-intensive than inference, especially with backpropagation, and incurs high access time and energy costs for communication between memory and on-chip buffers;
(ii) *Scalability for large graph sizes*: Graph sizes in real-world datasets have grown exponentially in recent years [8], necessitating multiple accelerator engines to work together;
(iii) *Load balancing during computation*: High and variable input feature vector sparsity, high adjacency matrix sparsity, and power-law distributions of vertex degrees, result in irregular and random memory accesses during GNN computations, with low utilization of processing elements [1]–[3].
(iv) *Versatility*: A GNN training accelerator must be able to accommodate a wide range of GNN architectures.

GPU-based solutions are energy-inefficient. GNNAdvisor [9], a single-GPU solution is limited to small-to-medium-sized graphs. Multi-GPUs platforms can handle large graphs: RoC [10] uses dynamic techniques for graph partitioning and memory management; NeuGraph [11] employs 2-D graph partitioning and inter-GPU vertex-chunk swapping (with increased communication overhead); PaGraph [12] replicates boundary vertices to reduce communication among partitions, but faces scalability issues due to replica synchronization.

Several FPGA- and ASIC-based accelerators with better energy efficiency have been proposed. Among FPGA-based approaches, GCoD [13] implements algorithm-accelerator co-design, but requires large on-chip buffers due to scatter-based aggregation and incurs high preprocessing overhead for sparsification and polarization; GraphACT [14] proposes a CPU+FPGA platform, with graph sampling and loss gradient calculation offloaded to the CPU, and forward- and back-propagation handled in the FPGA. Among ASIC-based approaches, Rubik [15], uses a hierarchical array of processing elements; GNNear [16] uses an ASIC-based central acceleration engine for some computations and offloads others to near-memory processing engines that reside in the buffer chips of DIMMs. As single-core structures, these methods are not scalable for larger graphs; they largely neglect input feature vector sparsity and power-law degree distribution problems.

Any single-core solution has limited scalability. We accelerate large GNN training by moving past the limitation of single cores and using an *array* of processing cores for training, offering substantial speedup and energy-efficiency improvements. We target much larger graphs than previous ASIC/FPGA training accelerators (we show results on datasets with up to 8.6M vertices in Section VI). We believe this is the first multicore GNN training accelerator to support a wide range of GNNs; the only other multicore accelerator [17] known to us handles inference only and not training. The existing multicore inference accelerators can not handle backpropagation efficiently due to: (i) massive computation/communication overhead for the calculation/propagation of error gradients. (ii) large gradient synchronization overhead. (iii) lack of support for various special functions, e.g., log and softmax.

For the core, we choose the GNNIE inference accelerator [1] over other candidates [2]–[7] as it can handle sparsity in input vertex feature vectors and adjacency matrix, support a wide range of GNN topologies (e.g., GCN, GraphSAGE, GAT, GINConv), and shows speedup and efficiency advantages over other methods. However, simply arraying a set of GNNIE cores leads to performance bottlenecks due to: (i) suboptimality in GNNIE's caching scheme in a multicore scenario; (ii) lack of multicore-specific optimizations that consider both DRAM accesses and inter-core communication. We develop novel techniques to address these challenges and develop methods that are *scalable* for training large graphs. Degree-Quant [18] proposes integer-based GNN training and we leverage this in our implementation. Our contributions are:

- A novel *feature vector segmentation* scheme that reduces memory accesses, and a random-forest-based machine learning (ML) model for optimal segmentation.
- *Multicore-specific graph-specific caching* with reduced random DRAM accesses and limited on-chip communication.
- *Demonstrated gains in scalability, speedup, and energy*

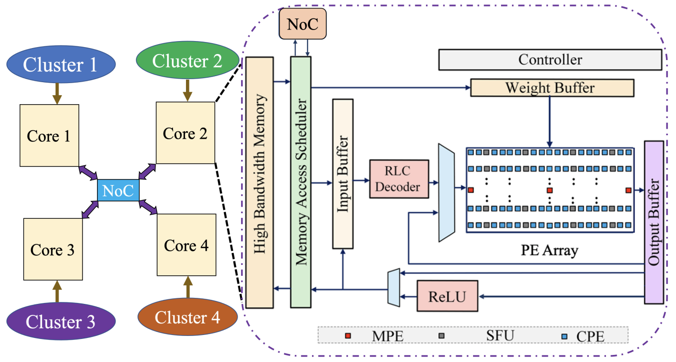

Fig. 1. Block diagram of the proposed architecture (core architecture in inset) with 4 cores; our evaluation considers accelerators with up to 36 cores.

efficiency over prior GPU/FPGA/ASIC solutions across multiple GNN topologies.

II. GNN Training Steps

GNN training involves a *forward pass* similar to inference, and a *backward pass* that feeds gradients back to update weights.

Forward Pass Computations. The forward pass has two steps [1], [2], [16]: ***(a) Weighting*** in layer l multiplies feature vector $\mathbf{h}_i^l$ (dimension F^l) of each vertex i by a weight matrix, W^l (dimension $F^{l-1} \times F^l$). ***(b) Aggregation*** for vertex i combines (sum/max/mean/pool) the weighted feature vectors in a set $\mathcal{N}_i$. For GCN/GAT/GINConv, $\mathcal{N}_i$ is the neighbors $N(i)$ of i; for GraphSAGE, $\mathcal{N}_i$ randomly samples $N(i)$.

Backward Pass Computations. The output node features of the forward pass are compared against the ground truth to compute the loss function. Then, starting from the last layer, the gradients of the loss with respect to the feature vectors and weights are calculated, and weight updates are performed at each layer using the chain rule until the input layer is reached. Backward pass computations consist of Weighting and Aggregation steps similar to the forward pass, and MAC operations for loss computations and gradient updates.

III. Multicore Architecture and Computations

Architecture. Our GNN training engine, shown in Fig. 1, has multiple cores connected by a network-on-chip (NoC).

GNNIE core [1]. A GNNIE core (inset of Fig. 1) consists of an $M \times N$ array of computational PEs (CPEs) for ALU computations; merge PEs (MPEs) within the CPE array that aggregate partial results in their CPE column during Weighting; and special function units (SFUs) for nonlinear functions. Three on-chip buffers cache the input, weight, and output data. The controller for each core orchestrates operations in the PE array (workload reordering for CPEs, sending partial results to MPEs). The memory access scheduler from [19] is modified to handle memory requests from both DRAM and NoC.

Partitioning. For a multicore training engine with m GNNIE cores, the input graph is partitioned into m clusters, and each cluster is the workload for one core. *Intra-cluster edges* are connections between vertices ("*intra-cluster vertices*") within a cluster and can be processed entirely within a core; *inter-cluster edges* connect vertices in the cluster to vertices in another cluster ("*inter-cluster vertices*"). We preprocess the graph with METIS [20] to create clusters that (a) are balanced, i.e., have roughly equal numbers of intra-cluster vertices, (b) have a minimal number of inter-cluster edges.

Weighting. Weighting is performed separately on the feature vectors of each vertex, and can be carried out independently in each core, with no inter-vertex/inter-core communication. The matrix-vector multiplication computations in this step are very structured, but input vector sparsity variations can lead to load imbalance during this computation. These issues are tackled using the workload imbalance strategies using GNNIE's flexible MAC architecture and load redistribution [1].

Aggregation. Aggregation consolidates data from the neighbors of each vertex, and may involve *intra-cluster and inter-cluster* edges. For most GNNs (GCN/GINConv/GraphSAGE), this involves summation, but GATs require nonlinear computation of attention coefficients. Aggregation for each vertex in a cluster is performed on its own core, with no synchronization. Operations on inter-cluster vertices, fetched via NoC from the buffers of other cores, are read-only because there are no data dependencies between operations in the same layer.

Ultra-high sparsity of the adjacency matrix and power-law behavior incur numerous irregular and random memory accesses even on one core; this is exacerbated for large graphs on multiple cores by heavy and irregular *communication between cores* due to long feature vectors, limited NoC bandwidth, and small on-chip buffers. We will address novel methods for overcoming these bottlenecks in Sections IV and V.

Dynamic caching. Since the data for each cluster is too large for the cache (input buffer), GNNIE uses a dynamic caching scheme to fetch vertex data from the DRAM. It processes a subgraph of the cluster, called the **computational subgraph**, which is the subset of intra-cluster and inter-cluster vertices of a core currently in the cache, and edges between these vertices.

For our multicore training engine, the intra-cluster vertices of a core and their edge data are stored in CSR format in the DRAM. For intra-cluster vertices, this data for a computational subgraph is fetched into the input buffer from DRAM (*off-chip communication*), and for inter-cluster vertices, the data is fetched from the input buffers of other cores (which are responsible for DRAM fetches) via the NoC (*on-chip communication*). Within each core, the CPEs process the computational subgraph using efficient load-balancing techniques [1].

IV. Dynamic Cache Replacement Policy

We first review the hardware-centric graph-specific caching technique in [1] for the GNNIE inference engine. To maximize cache data reuse, the number of unprocessed edges, e_v, for each vertex v is tracked during Aggregation. Since nodes with larger e_v are involved in a larger number of future computations, they are more likely to be reused; hence they are prioritized for retention in the cache. Specifically, a node is replaced in the cache if $e_v \leq \gamma$, where γ is a threshold.

This strategy is used to promote cache reuse and minimize DRAM fetches. The graph undergoes lightweight preprocessing to store the vertices in descending order of degree in DRAM (initially, $e_v = d_v$, the degree of vertex v). The cache is initially populated with the first set of DRAM blocks with the highest degrees. An iteration is completed when all edges of the computational subgraph in the cache are processed. At this time, the set of cache blocks that meet the replacement criterion are evicted and the next sequential set of DRAM blocks (the DRAM is stored in degree order) is brought into cache. Multiple iterations are needed until all edges are traversed. By construction, DRAM fetches exclusively use sequential blocks, avoiding expensive random access. All random accesses are limited to the on-chip SRAM cache, which is much more inexpensive than random DRAM access.

Multicore-specific Graph-specific Caching. The direct application of the graph-specific caching scheme of [1] to large graphs in the multicore scenario results in bottlenecks related to *stagnation* (described next), and requirements for increased retention of inter-cluster vertices whose data must be sent over the NoC to other cores. We alter the scheme, using new methods that use dynamic thresholds to prevent stagnation.

In the multicore scenario, the preprocessing step stores each cluster of the graph (instead of the entire graph) in degree order. The retention requirements for intra-cluster and inter-cluster vertices are different. Due to the min-cut objective function of clustering, intra-cluster vertices in a core tend to have higher connectivity within the cluster, while inter-cluster vertices are connected to fewer vertices. Using the same γ threshold for both types of vertices would disadvantage inter-cluster vertices, which might then require frequent fetches across the NoC, with high latency and cost overheads. Therefore, separate thresholds γ_{intra} and γ_{inter} are required for retaining intra-cluster and inter-cluster vertices, respectively.

As the distribution of vertex degrees varies across clusters, the values of these γ parameters must be cluster-specific, and using a uniform value of these γ variables for all clusters is inefficient. For each core, we set γ_{intra} (γ_{inter}) to a certain percentile value, κ_{intra} (κ_{inter}) of the degree distribution of intra-cluster (inter-cluster) vertices of the cluster assigned to the core. Empirically, we find that setting κ_{intra} and κ_{inter} to the 50^{th} percentile of the intra-cluster and inter-cluster vertex degree distribution, respectively, is a good choice. Our choice of γ parameters based on the above criterion also makes the approach generalized (i.e., not tuned for a particular dataset).

We track the unprocessed intra-cluster (inter-cluster) edges of an intra-cluster (inter-cluster) vertex through a simple decrement operation, i.e., whenever an intra-cluster (inter-cluster) edge of an intra-cluster (inter-cluster) vertex is processed in the CPE array, the controller decrements the intra-cluster (inter-cluster) edge count of the vertex by 1. Then, based on γ_{intra} (γ_{inter}) cache replacement is performed. Fetch operations for the next set of intra-cluster and inter-cluster vertices via off-chip and on-chip communication for the next subgraph are overlapped with the computation in the CPE array.

Dynamic Thresholds for Preventing Stagnation. Intra-cluster (inter-cluster) stagnation occurs when the number of cached intra-cluster vertices that meet the eviction criterion based on γ_{intra} (γ_{inter}) is small, as the changes in the computational subgraph across iterations are minor. This results in low computation and low PE utilization per iteration.

We define the metric $e_{intra}[i]$ ($e_{inter}[i]$) as the ratio of the number of intra-cluster (inter-cluster) edges processed up to iteration i, to the total number of intra-cluster (inter-cluster) edges of the cluster associated with the core. After a *detection interval* of every I iterations, we detect stagnation as:

$$e_{intra}[i] \leq (1+\delta)e_{intra}[i-I], \quad e_{inter}[i] \leq (1+\delta)e_{inter}[i-I]$$

where δ is a user-defined threshold. If this is satisfied, we

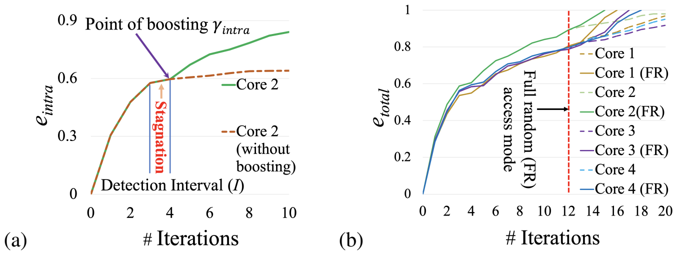

Fig. 2. (a) Boosting γ_{intra} to break intra-cluster stagnation on Core 2. (b) Invoking full random access after most edges are processed on all cores.

boost the relevant γ to κ_{boost}^{th} percentile of the vertex degree distribution. After one iteration with the boosted value evicts numerous vertices and overcomes stagnation by changing the computational subgraph, we revert to the original γ.

We tune the parameter values over a range of datasets. Varying $\kappa_{boost} \in [70, 95]$, the optimal value was found to be $\kappa_{boost} = 90$; varying $I \in [1, 15]$, an optimum was found at $I = 5$; varying $\delta \in [0.01, 0.1]$ yielded the best value of $\delta = 0.05$. For the amazon0601 dataset on a 4-core system, Fig. 2(a) shows the change in e_{intra} with each iteration and shows regions of stagnation that is found after the detection interval of I. At this point, γ_{intra} is boosted, and as shown in Fig. 2(a) this increases the rate of progress of e_{intra} (the dotted line shows the slower trajectory of e_{intra} without boosting).

As the Aggregation computation nears completion, when a large fraction of edges has been processed, it becomes increasingly difficult to find unprocessed edges in the computational subgraph. To detect this, we monitor e_{total}, the ratio of the number of intra-cluster/inter-cluster edges processed up to iteration i, to the total number of edges in the cluster. When e_{total} exceeds a threshold, we move to full random access: the cache now has random access to the DRAM to complete Aggregation. This is shown in Fig. 2(b), where the original trajectory (dotted lines), is accelerated to faster completion (solid lines). At this stage, the number of random DRAM accesses is relatively small and the benefit of faster convergence outweighs the cost of slower random DRAM accesses during this final phase. We find the e_{total} threshold values of 0.8 to be optimal over a range of datasets.

V. Scaling on Large GNNs

A. Bottlenecks of Scaling on Large GNNs

While graph-specific caching significantly improves latency and power/energy due to increased data reuse, the benefits of this approach face bottlenecks due to *fundamental limitations* in the traditional structure of GNN computations. Since node feature vectors for each node can be long and the input buffer size is small, the computational subgraph in each iteration constitutes a very small fraction of the total number of vertices in each cluster. Thus, only a small fraction of edges can be processed in each iteration, leading to high rates of cache replacement and slow convergence. Switching to full random access mode can overcome this issue, with significant costs due to the high energy of random DRAM access. The problem becomes more acute for large GNNs that require more cores: with more cores, more inter-cluster vertices are sent over NoC, leading to higher injection rates and/or larger packet sizes. This increases NoC latency, worsening performance.

The key to overcoming this problem is to increase the size of the computational subgraph during Aggregation, subject to the cache size. We achieve this by proposing *feature vector segmentation*, splitting a vertex feature vector into multiple segments, processing one segment at a time. We show that the choice of segment size involves balancing off-chip and on-chip communication latency as we seek to efficiently overlap computation with communication for high performance.

B. Feature Vector Segmentation

During Aggregation, there is no dependency between operations in different elements of the feature vector of a vertex. Therefore, Aggregation over the neighborhood for each feature vector segment can be carried out independently. We develop the concept of feature vector segmentation under a fixed cache size, illustrated in Fig. 3. The conventional approach at left ("Full") uses full feature vectors of length F. For a cache size of C, the number of vertex feature vectors that can fit in the cache is roughly $n_F = C/F$, and this limits the size of the computational subgraph. We can increase the subgraph size by using a *subset* of the entire feature vector. If we split the feature vector into two segments ("2-segments," middle), we can fit a subgraph of $2n_F$ vertices in the cache. For the j-segments case (right), where each segment length is $q = \lceil F/j \rceil$, we increase the size of the computational subgraph by a factor of j relative to the "Full" case.

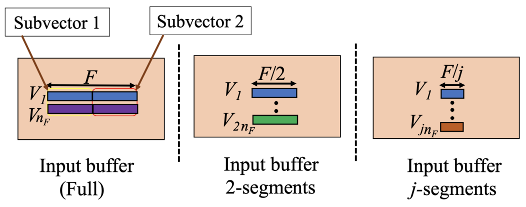

Fig. 3. Feature vector segmentation.

Using j segments, Aggregation operations are performed on one feature vector segment at a time, over all nodes, using the graph-specific caching method of Section IV. For larger j, the computational graph in each iteration is larger, and more edges are available for Aggregation, so that CPEs in each core are kept busy. However, as j increases, more vertices fit into each core and have more neighbors in other cores. Hence, traffic in the NoC also increases as more vertices are sent to other cores, increasing the injection rate and slowing communication.

A few prior approaches have used concepts similar to segmentation, but have significant limitations: our solution gains efficiency by exploiting segmentation in harmony with other schemes that reduce cache access latencies, including graph-specific caching and on-chip fetches from other cores using the NoC. P^3 [21] uses a superficially similar method that segments the feature vector into $\lceil F/c \rceil$ segments, where c is the number of GPU cores. This accommodates the entire graph into each core, but their power-hungry GPU-based solution requires a much larger cache than our power-efficient ASIC solution. BoostGCN [7] and GNNerator [4] implement 2-D graph partitioning and use feature vector segmentation to increase the number of vertices in each partition so that the frequency of DRAM communication decreases. However, as shown in [1], the lower bound of DRAM access for 2-D partitioning is always higher than the graph-specific caching

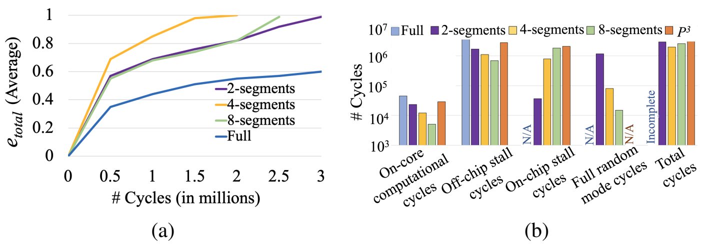

Fig. 4. Performance analysis of feature vector segmentation: (a) e_{total} (Average) vs. Execution Cycles (b) Aggregation cycle comparison.

of GNNIE. In addition, while BoostGCN and GNNerator rely solely on DRAM communication to fetch neighboring vertices, we fetch neighbors from DRAM (off-chip communication) or from the cache of other machines (on-chip communication); since DRAM accesses are more costly than on-chip communication, our approach is more efficient.

Performance Analysis. Fig. 4 shows the results of combining feature vector segmentation with dynamic graph-specific caching (Section IV), showing the number of cycles required for Aggregation in the first GCN layer for the amazon0601 dataset (403,394 vertices/3,387,388 edges) on a 4-core system. The results are shown for $j = 1$ (Full), 2, 4, and 8.

Fig. 4(a) shows the progress in processing graph edges as the execution progresses, showing the fraction e_{total} of all edges that are processed (averaged over all cores). For the Full case, e_{total} rises very slowly and does not reach the threshold of 0.8 required to transition to full random access mode. The 2-segments approach progresses faster, and the 4-segments approach is still faster; however, increasing to 8-segments slows down the progress of e_{total}. We will understand this trend based on Fig. 4(b), which shows the total number of cycles for the computation, and the components of this total.

Fig. 4(b) shows improvement in *on-core computation* cycles from the Full to the segmented cases with larger j. Compared to the Full approach, the 2-, 4-, and 8-segment cases reduce *off-chip stall cycles* by 60%, 74%, and 83%, respectively: increasing j reduces DRAM access frequency as the computational subgraph becomes larger. While for the Full case the computational subgraph, on average, contains just 8% of the vertices in a cluster; for 2, 4, and 8 segments, the fraction rises to 16%, 32%, and 64%, respectively. The number of *full random mode cycles* reduces as j increases, because the computation subgraph grows larger as j increases and fewer edges are unprocessed at the switch to full random mode.

Under segmentation, the NoC injection rate increases with higher j due to the increased size of the computational subgraph on each core, which results in the transmittal of multiple feature vector segments of inter-cluster vertices over the NoC. However, since individual segments are smaller, the message size is reduced. This tradeoff between the increased injection rate and the reduced message size implies there is an optimal j for which the on-chip stall cycles are minimized. We develop an ML model to optimize j. The impact of stalls can be further mitigated by overlapping on-chip and off-chip communication during each iteration. In particular, no on-chip stall cycles are required for the Full approach, because on-chip communication per iteration is so low (due to a small number of cached vertices) that it can be completely overlapped.

Fig. 4(b) also shows that the *on-chip stall cycles* increase from 2- to 4- to 8-segment cases. Hence, the total cycle count required to complete the computation has its minimum for the 4-segments case. This explains the trend in Fig. 4(a).

ML Model for Optimizing j. To optimize the number of segments j, we trained a machine learning (ML) model using a random forest (RF) regressor. This trained ML model is then used on unseen graphs. Input parameters include graph attributes (vertex and edge counts, a power-law metric capturing the fraction of edges adjacent on the top 10 percentile of high-degree vertices, and the number of cores). The total number of samples corresponds to 144 synthetic graphs, ranging from 100K–10M vertices and 200k–100M edges, and with power-law metric from 0.27–0.95. A train/test split of 80/20 was used. The RF regressor used 100 estimators, and the model achieved 95% training and test accuracy based on R^2 score.

On real datasets, the model prediction was close to the results of a much more costly enumeration of all $j \in [2, 16]$. For example, the optimal j predicted by the RF model is 5 for amazon0505 and amazon0601 datasets, close to optimal enumerated value of 4 ; the prediction for com-amazon dataset is $j = 4$, which matches the optimum from enumeration.

Comparison with P^3. Fig. 4(b) also show results for P^3 [21], which switches between a segmented method (model parallelism, across feature vector segments) to the Full method (data parallelism, across cores) after the first layer, incurring a large communication overhead due to a burst of communication at that stage; by its very nature, this step provides no opportunities for overlapping off-chip and on-chip communication. Hence, it incurs high on-chip stalls: 89%, 63%, and 7% higher on-chip stall cycles compared to 2-, 4-, and 8-segments, respectively, in our implementation of their method.

PE Utilization. The segmented approach also leads to higher PE utilization compared to full-length feature vectors. The larger computational subgraph size allows more edges to be processed per iteration, increasing the computational intensity of data fetched from DRAM and keeping GNNIE PEs busy. For the first layer of Aggregation of amazon0601 for GCN, the average PE utilization for Full, 2-segments, 4-segments, and 8-segments are 67%, 86%, 100%, and 100%, respectively.

VI. Evaluation

Hardware/Simulation Setup. Each core is implemented in Verilog, synthesized with Synopsys DC in a 12nm standard VT library, placed and routed using Innovus, and verified via RTL simulations. The area, energy, and latency of on-chip buffers are estimated using CACTI 6.5 [22]. Post-P&R metrics for each core are: 4.97mm^2, 0.93W, 934 MHz. The controller has 0.26 mm^2 area and 0.1W power. For the NoC, latency and throughput were analyzed using BookSim2 [23], and power and area using Orion3.0 [24]. The NoC power overhead ranges between 2.9%–6.3% of the total chip power. An in-house simulator computes the execution cycles for our accelerator, with Ramulator [25] modeling off-chip HBM access (256 GB/s, 3.97pJ/bit [26]).

Configuration of the Multicore Accelerator.
Individual GNNIE cores Configuration per core is as follows:
Buffer sizes: Output: 1MB; Weight: 128KB; Input: 512KB
CPE array with flexible MACs: 16×16 array; 4 MACs (rows 1–8), 5 MACs (rows 9–12), 6 MACs (rows 13–16).
NoC Buffer size: 128 KB, 4 links per router, 50GB/s BW/link.

Table I: Type A datasets
(DD: D&D, TW: TWITTER-Partial, YT: Yeast, SW: SW-620H, OV: OVCAR-8H)

Dataset	Vertices	Edges	(FL, m, j)
DD	335K	1.7M	(89, 2, 4)
TW	581K	1.4M	(1323, 4, 2)
YT	1.7M	3.6M	(74, 16, 2)
SW	1.9M	3.9M	(66, 16, 2)
OV	1.9M	3.9M	(66, 16, 2)

Table II: Type B datasets
(SB: soc-BlogCatalog, CA: com-amazon, A-05: amazon0505, A-06: amazon0601, EN: enwiki, A-8M: amazon8M)

Dataset	Vertices	Edges	(FL, m, j)
SB	89K	2.1M	(128, 1, 2)
CA	335K	1.9M	(96, 2, 4)
A-05	410K	4.9M	(96, 4, 4)
A-06	403K	3.4M	(96, 4, 4)
EN	3.6M	276.1M	(300, 16, 16)
A-8M	8.6M	231.6M	(96, 36, 16)

Number of GNNIE cores The number of cores for a dataset is based on the ratio, ϑ, of vertices per computational subgraph (i.e., the full-length vertex features that can fit in cache) to the vertices assigned per core. Empirically, we determined that its optimal range is $0.03 \leq \vartheta \leq 0.15$. Using this, we find the number of cores m (see Tables I and II) for the optimal ϑ that optimizes speedup gain vs. area/power overhead.

We analyze the change in speedup when the number of cores is altered from the optimal m. For the A-06 dataset, $m = 4$; for 2, 16, and 36 cores, the speedup changes by 0.43×, 3.1×, and 7.29×, respectively. In each case, the speedup change is sublinear, indicating that $m = 4$ is optimal.

Benchmark GNN Datasets and Models. We evaluate the performance of our platform using Type A and Type B benchmark graph datasets from Table I and II, respectively. Type A datasets consist of multiple small graphs with no inter-graph edges, while Type B datasets are large monolithic graphs with a high amount of structural irregularity, i.e., higher adjacency matrix sparsity and power-law behavior. Table I and II also provide the input feature length (FL), number of cores (m), and feature vector segments (j) used for each dataset.

We evaluate the accelerator for training four GNN models: GCN, GINConv, GAT, and GraphSAGE. All GNNs have one hidden layer, except GINConv which has five; for GCN, GINConv, and GraphSAGE each hidden layer has 16, 64, and 256 channels, respectively. The GAT hidden layer uses eight 16-dimensional attention heads. All speedup and energy numbers include preprocessing times, including runtime for graph partitioning, degree-based vertex reordering, workload reordering, and neighborhood sampling time (performed on Intel Xeon Gold@2.60GHz CPU) for GraphSAGE. The preprocessing overhead over 500 epochs for amazon0601 is 18%.

Performance comparison with DGL. We compare all GNNs against Deep Graph Library (DGL) [27] on a V100 Tesla GPU with V100S-PCIe@1.25GHz, 32GB HBM2 ("*DGL+Tesla V100*"). The training latency for speedup comparison are averaged over 500 epochs. As shown in Fig. 5(a) and (b), the average speedup of our approach against DGL+Tesla V100 for GCN, GINConv, GAT, GraphSAGE ranges from 8.9×–46.6× across Type A datasets and 3.3×–15.5× for Type B.

The speedup comes from several of our optimizations: (i) Feature vertex segmentation improves scalability for large GNNs. (ii) Dynamic cache replacement mitigates irregular random memory accesses and on-chip communication overhead. (iii) Distributed computation across multiple batches ensures weight reuse. The speedup is particularly high for GINConv: unlike DGL, we use dimension-aware stage reordering (DASR) [1], [3], which requires fewer computations. To determine their impact, we removed these optimizations successively on A-06. Without segmentation, the computation did not complete (as in Fig. 4). With optimal segmentation,

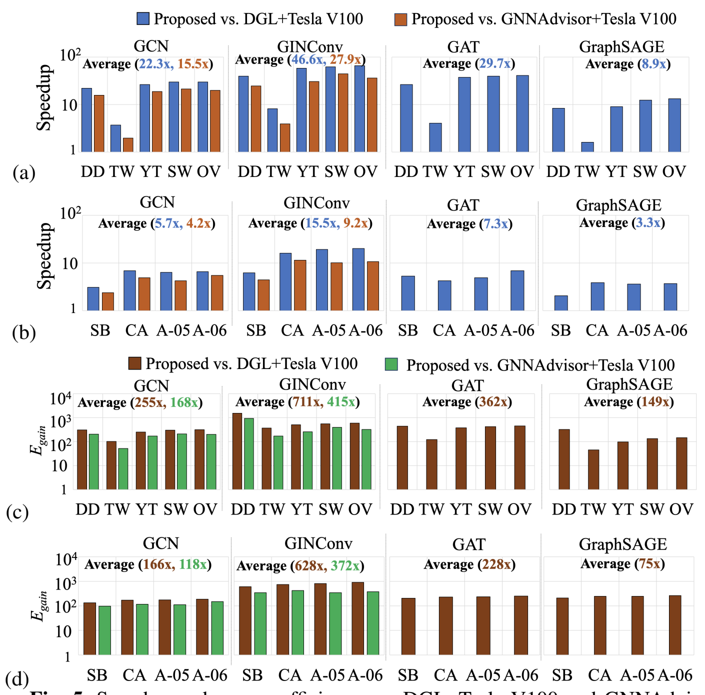

Fig. 5. Speedup and energy efficiency vs. DGL+Tesla V100 and GNNAdvisor+Tesla V100: (a), (c): Type A datasets (b), (d): Type B datasets.

removing dynamic cache replacement increases runtime by 34%; also removing weight reuse raises the penalty to 43%.

GraphSAGE shows lower speedup than other models due to: (i) inclusion of preprocessing time for neighborhood sampling on our platform, but not on DGL+Tesla V100. (ii) mitigation of power-law behavior in real-world graphs by sampling. Type A datasets have higher speedups than Type B datasets due to the lack of on-chip communication overheads. Larger datasets (e.g., OV, A-06) show higher speedups than smaller datasets (e.g., DD, SB) for both Type A and B, indicating scalability.

Comparison with GPU-based accelerators. Speedup: GNNAdvisor implements only GCN and GINConv. For the same configurations for these GNNs, Fig. 5(a) and (b) shows that relative to GNNAdvisor, we achieve 15.5×–27.9× speedup for Type A and 4.2×–9.2× for Type B datasets.

NeuGraph uses 2-D graph partitioning to process large graphs using one NVIDIA Tesla P100 GPU. We achieve 12.2× and 16.9× speedup for GCN on EN and A-8M, respectively, over NeuGraph. The corresponding speedups over GNNAdvisor are 3.1× and 6.8×, respectively.

Energy: Fig. 5(c) and (d), illustrate the energy efficiency comparison with Tesla V100, reporting E_{gain}, the ratio of the energy required by the GPU to the energy of our approach. Compared DGL+Tesla V100, our average E_{gain} ranges from 149×–711× over Type A datasets and 75×–628× over Type B. Against GNNAdvisor+Tesla V100, E_{gain} ranges from 168×–415× and 118×–372×, respectively.

Comparison with FPGA-/ASIC-based accelerators. Our approach achieves an average speedup of 11× and 24× over Rubik and GraphACT, respectively; neither reports absolute power numbers. Our speedup over Rubik is due to its inefficient reuse of cache data which incurs high on-chip and off-chip communication costs, and over GraphACT since it does not consider the power-law behavior of real-world graphs and makes no explicit efforts to address the random off-chip memory accesses. In comparison with GNNear, we achieve 17× average speedup over DGL+Tesla V100, but the speedup of GNNear is only 2.5×. Unlike our approach, the graph partitioner of GNNear is oblivious to community structure in real-world graphs, resulting in high communication costs due to the high number of cut edges between the partitions. GCoD handles only small graphs (up to 233K vertices, as against 8.6M vertices for our approach), and uses a whopping 180W of power even for these graphs, which can be handled by our approach on a single core using < 1W.

VII. Conclusion

Our multicore GNN training accelerator has GPU-like scalability and accelerator-like efficiency for large GNNs, leveraging novel feature vector segmentation and dynamic caching schemes for scalability and to mitigate communication costs.

References

[1] S. Mondal *et al.*, "A Unified Engine for Accelerating GNN Weighting/Aggregation Operations, with Efficient Load Balancing and Graph-Specific Caching," *IEEE T. Comput. Aid. D.*, 2022.
[2] M. Yan *et al.*, "HyGCN: A GCN Accelerator with Hybrid Architecture," in *HPCA*, 2020.
[3] T. Geng *et al.*, "AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing," in *MICRO*, 2020.
[4] J. Stevens *et al.*, "GNNerator: A Hardware/Software Framework for Accelerating Graph Neural Networks," in *DAC*, 2021.
[5] Z. Zhou *et al.*, "BlockGNN: Towards Efficient GNN Acceleration Using Block-Circulant Weight Matrices," in *DAC*, 2021.
[6] C. Chen *et al.*, "DyGNN: Algorithm and Architecture Support of Dynamic Pruning for Graph Neural Networks," in *DAC*, 2021.
[7] B. Zhang *et al.*, "BoostGCN: A Framework for Optimizing GCN Inference on FPGA," in *FCCM*, 2021.
[8] W. Hu *et al.*, "Open Graph Benchmark: Datasets for Machine Learning on Graphs," in *NeurIPS*, 2020.
[9] Y. Wang *et al.*, "GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs," in *OSDI*, 2021.
[10] Z. Jia *et al.*, "Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc," in *MLSys*, 2020.
[11] L.Ma *et al.*, "NeuGraph: Parallel Deep Neural Network Computation on Large Graphs," in *USENIX ATC*, 2019.
[12] Z. Lin *et al.*, "PaGraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching," in *SoCC*, 2020.
[13] H. You *et al.*, "GCoD: Graph Convolutional Network Acceleration via Dedicated Algorithm and Accelerator Co-Design," in *HPCA*, 2022.
[14] H. Zeng *et al.*, "GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms," in *FPGA*, 2020.
[15] X. Chen *et al.*, "Rubik: A Hierarchical Architecture for Efficient Graph Neural Network Training," *IEEE T. Comput. Aid. D.*, 2021.
[16] Z. Zhou *et al.*, "GNNear: Accelerating Full-Batch Training of Graph Neural Networks with Near-Memory Processing," in *PACT*, 2021.
[17] G. Sun *et al.*, "Multi-Node Acceleration for Large-Scale GCNs," *IEEE Transactions on Computers*, 2022.
[18] S. Tailor *et al.*, "Degree-Quant: Quantization-Aware Training for Graph Neural Networks," in *ICLR*, 2021.
[19] S. Mondal *et al.*, "GNNIE: GNN Inference Engine with Load-Balancing and Graph-Specific Caching," in *DAC*, 2022.
[20] G. Karypis *et al.*, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Sci. Comput.*, 1998.
[21] S. Gandhi *et al.*, "P^3: Distributed Deep Graph Learning at Scale," in *OSDI*, 2021.
[22] "CACTI 6.5." https://github.com/Chun-Feng/CACTI-6.5.
[23] J. Nan *et al.*, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *ISPASS*, 2013.
[24] A. B. Kahng *et al.*, "ORION3.0: A Comprehensive NoC Router Estimation Tool," *IEEE Embedded Sys. Lett.*, 2015.
[25] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Comp. Arch. Lett.*, vol. 15, no. 1, 2015.
[26] M. O'Connor *et al.*, "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems," in *ISCA*, 2017.
[27] M. Wang *et al.*, "Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs," in *ICLR*, 2019. https://github.com/dmlc/dgl/.