A Unified Engine for Accelerating GNN Weighting/Aggregation Operations, with Efficient Load Balancing and Graph-Specific Caching

Sudipta Mondal, Susmita Dey Manasi, Kishor Kunal, Ramprasath S., Ziqing Zeng, and Sachin S. Sapatnekar

***Abstract*—Graph neural networks (GNN) analysis engines are vital for real-world problems that use large graph models. Challenges for a GNN hardware platform include the ability to (a) host a variety of GNNs, (b) handle high sparsity in input vertex feature vectors and the graph adjacency matrix and the accompanying random memory access patterns, and (c) maintain load-balanced computation in the face of uneven workloads, induced by high sparsity and power-law vertex degree distributions. This paper proposes GNNIE, an accelerator designed to run a broad range of GNNs. It tackles workload imbalance by (i) splitting vertex feature operands into blocks, (ii) reordering and redistributing computations, (iii) using a novel flexible MAC architecture. It adopts a graph-specific, degree-aware caching policy that is well suited to real-world graph characteristics. The policy enhances on-chip data reuse and avoids random memory access to DRAM. GNNIE achieves average speedups of 7197× over a CPU and 17.81× over a GPU over multiple datasets on graph attention networks (GATs), graph convolutional networks (GCNs), GraphSAGE, GINConv, and DiffPool. Compared to prior approaches, GNNIE achieves an average speedup of 5× over HyGCN (which cannot implement GATs) for GCN, GraphSAGE, and GINConv. GNNIE achieves an average speedup of 1.3× over AWB-GCN (which runs only GCNs), despite using 3.4× fewer processing units.**

***Index Terms*—GNN, hardware accelerator, graph-specific caching, load balancing.**

I. Introduction

Deep learning accelerators have largely focused on data with Euclidean embeddings, e.g., audio/video/images/speech. Many real-world problems (e.g., network analysis, embedded sensing, e-commerce, drug interactions) use graphs to model relationships. Inferencing on large, unstructured, and sparse graphs with non-Euclidean embeddings requires specialized graph neural networks (GNNs). Today's GNNs [1]–[4] are based on nearest-neighbor operations, with improved efficiency over early methods [1], [5], [6].

Multilayer GNNs perform two computation steps per layer: ***(a) Weighting*** performs a linear transform of vertex feature vectors through multiplication by a weight matrix.
(b) Aggregation consolidates information from the neighbors of a vertex to compute the feature vectors for the next layer.
The challenges in building GNN accelerators are as follows:
(1) Versatility An accelerator should be able to handle a diverse set of GNNs to cover a wide range of GNN architectures to provide appropriate computation/accuracy tradeoff points for various applications. The achievable accuracy depends on the GNN: graph attention networks (GATs) achieve higher accuracy than other GNNs, but with more computation (Fig. 1).

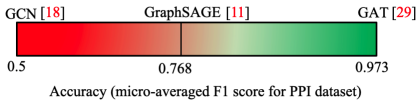

Fig. 1. GNN accuracy comparison (data from [3], PPI dataset).

(2) Adjacency matrix sparsity The graph adjacency matrix encodes vertex neighborhood information required for Aggregation. The adjacency matrix is *highly sparse* ($> 99.8\%$ for all datasets in this paper; in contrast, DNN data shows 10%–50% sparsity). Unlike image/video data, adjacency matrix sparsity patterns typically exhibit *power-law behavior*, with vertex degrees ranging from very low (for most vertices) to extremely high (for very few vertices): in the Reddit dataset, 11% of the vertices cover 88% of all edges.
(3) Input feature vector sparsity The vertex *input feature vectors* are highly sparse, e.g., the 2708 input vertex feature vectors of the Cora dataset have 98.73% average sparsity. In Fig. 2, Region A is sparser than B and requires less computation, leading to load balancing issues during Weighting.

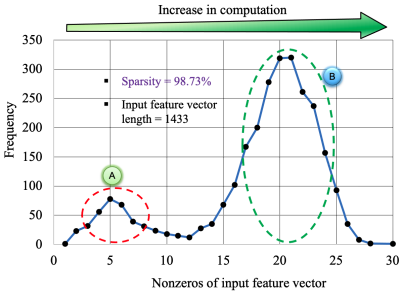

Fig. 2. Nonzero histogram for input vertex feature vectors (Cora).

(4) Memory footprint and random-access patterns Real-world graphs have a large number of vertices and a massive memory footprint (Reddit: 2.3Gb in sparse format). High sparsity and power-law distributions can lead to random memory access patterns and poor data access locality in Aggregation.

Therefore, GNN-specific accelerators must address:
(a) *load balancing during Weighting*, due to the sparsity variations in Fig. 2, and *during Aggregation*, due to the imbalance of computations for high- and low-degree vertices.

S. Mondal, S.D. Manasi, K. Kunal, Ramprasath S., Z. Zeng, and S. S. Sapatnekar are with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN. This work was supported in part by the Semiconductor Research Corporation (SRC).

(b) *lightweight graph-specific caching* of the adjacency matrix for high data access locality and maximal reuse of cached data.

Relation to other acceleration engines: The Weighting step performs matrix-vector multiplication which resembles convolutional neural network (CNN) computations, but CNN accelerators [7]–[13] are inefficient at handling graph data. Aggregation operates on graph neighborhoods and resembles graph analytics, but graph processing accelerators [14]–[16] are designed to perform lightweight computations, significantly lower than the needs of a GNN. Extensions of CNN/graph processing engines are inadequate.

An early GNN accelerator, HyGCN [17], bridges the divide by using two pipelined engines: an Aggregation engine that operates on graph data and consolidates vertex feature vectors from the neighborhood of each vertex, followed by a Combination engine, which uses a multilevel perceptron to weight the aggregated features with the weight matrix. The disparity between engines raises challenges in providing a steady stream of data to keep the Aggregation/Combination engine pipeline busy. The Aggregation engine does not account for power-law behavior while caching partial results, and high-degree vertices may create stalls due to the limited size of on-chip buffers. In the Combination engine, the aggregated feature vectors are both sparse and show high sparsity variations (Fig. 2). Consequently, stalls are required, leading to inefficiency.

AWB-GCN [18] views the GNN computation as two consecutive sparse-dense matrix multiplications (SpMMs). During Weighting, the method is targeted to moderate sparsity of 75% – but input layer vertex feature vectors are ultra-sparse (Fig. 2). During Aggregation, the graph-agnostic SpMM view necessitates numerous expensive off-chip accesses to the adjacency matrix. AWB-GCN addresses workload imbalances issues through multiple rounds of runtime load-rebalancing, but this leads to high inter-PE communication. Finally, SpMM-based approaches face more severe load imbalances for implementing GNNs that involve additional complex computations before Aggregation (e.g., softmax in GATs and DiffPool). In fact, AWB-GCN targets only GCNs and not general GNNs.

Novelty of this work: We propose the GNNIE (pronounced "genie") architecture that uses a *single engine* that efficiently performs both Weighting and Aggregation. The GNNIE framework handles high levels of sparsity in the input vertex feature vectors and the adjacency matrix, with novel approaches for *load balancing* and *graph-specific caching*. It covers a *number of GNN topologies*, from lower accuracy/lower computation (e.g., GCN, GraphSAGE) higher accuracy/higher computation (e.g., GATs), as motivated in Fig. 1, and is *more versatile* than previous methods in handling functions such as softmax over a neighborhood (e.g., as used for attention normalization in GATs; prior work [19] on GATs, skips this crucial step).

Novel methods to mitigate sparsity effects, and overcome load imbalances and compute bottlenecks, include:

- ***Load balancing during Weighting*** based on splitting vertex features into blocks (Section IV-A). Together with load balancing (Section IV-C), this enhances throughput during Weighting by ensuring high PE utilization and skipping unnecessary computations, by (a) Reordering computations on a *flexible MAC (FM) architecture* to address imbalances due to input feature vector sparsity variations. Computations are dynamically mapped to heterogeneous PEs, each with different numbers of MAC units. (b) *Static load redistribution* to nearby PEs, offloading computations from heavily-loaded to lightly-loaded rows, minimizing inter-PE communication.
- ***Load-balanced edge Aggregation*** (Section V) through a mapping scheme that fully utilizes the PEs. For GATs, we further propose a novel linear-complexity computation that implements compute-bound attention vector multiplication similarly as Weighting, and memory-bound attention coefficient computation to maximize reuse of cached data.
- ***Lightweight graph-specific dynamic caching*** (Section VI), fetching vertices in unprocessed degree order; aggregation operates on dynamic subgraphs formed by cached vertices. This new lightweight scheme is effective in avoiding the random DRAM accesses that plague graph computation.

Speedups: On five GNN datasets, results based on an RTL implementation and simulation show that even including all of its preprocessing overheads, GNNIE delivers average speedups of 7197× over CPUs (Intel Xeon Gold 6132 + PyTorch Geometric), 17.81× over GPUs (V100 Tesla V100S-PCI + PyTorch Geometric) and 5× over prior work.

II. Background

In layer l of a GNN, each vertex i in the graph is represented by an F^l-dimensional row vector, $\mathbf{h}_i^l$, called the *vertex feature vector*; $\mathbf{h}_i^0$ is the input vertex feature vector. For each vertex i in a layer, over a set of neighboring vertices j, the GNN aggregates information from vectors $\mathbf{h}_j^{l-1}$ of the previous layer, and processes it to create the output feature vector, $\mathbf{h}_i^l$.

Table I shows the Weighting and Aggregation operations for graph convolution networks (GCNs) [1], GraphSAGE [2], graph attention networks (GATs) [3], and GINConv [4].

Table I: Summary of operations in layer l of various GNNs.

GCN	$\mathbf{h}_i^l = \sigma\left(\sum_{j\in\{i\}\cup N(i)} \frac{1}{\sqrt{d_i d_j}} \mathbf{h}_j^{l-1} W^l\right)$
GraphSAGE	$\mathbf{h}_i^l = \sigma\left(a_k\left(\mathbf{h}_j^{l-1}W^l \forall\, j \in \{i\}\cup S_{N(i)}\right)\right)$
GAT	$\mathbf{h}_i^l = \sigma\left(\frac{\sum_{j\in\{i\}\cup N(i)} \exp(e_{ij})\mathbf{h}_j^{l-1}W^l}{\sum_{j\in\{i\}\cup N(i)}\exp(e_{ij})}\right)$ $e_{ij} = \text{LeakyReLU}(\mathbf{a}^T \cdot [\mathbf{h}_i^{l-1}W^l \| \mathbf{h}_j^{l-1}W^l])$
GINConv	$\mathbf{h}_i^l = \text{MLP}^l\left((1+\epsilon^l)\mathbf{h}_i^{l-1} + \sum_{j\in\mathcal{N}(i)} \mathbf{h}_j^{l-1}, W^l, \mathbf{b}^l\right)$

Weighting multiplies the vertex feature vector, $\mathbf{h}_i^{l-1}$ of each vertex by a weight matrix, W^l, of dimension $F^{l-1} \times F^l$.

Aggregation combines the weighted vertex feature vectors neighboring vertex i. If $N(i)$ is the immediate one-hop neighborhood of vertex i, then for GCNs, GATs, and GINConv, $\mathcal{N}(i) = \{i\} \cup N(i)$. For GraphSAGE, $\mathcal{N}(i) = \{i\} \cup S_{N(i)}$, where $S_{N(i)}$ is a random sample of $N(i)$. At vertex i:

GCNs: Each product $\mathbf{h}_j^{l-1}W^l$, $j \in \mathcal{N}(i)$, is multiplied by $1/\sqrt{d_i d_j}$ (d_* is the vertex degree). The result is summed.

GraphSAGE: The products $\mathbf{h}_j^{l-1}W^l$ are combined over $j \in \mathcal{N}(i)$ using aggregator a_k (typically, mean or pooling).

GATs: For each edge (i, j), an inner product with a learned attention vector $\mathbf{a}^l$ finds the normalized attention coefficient

$$\alpha_{ij} = \text{softmax}(\text{LeakyReLU}(\mathbf{a}^{l^T} \cdot [\mathbf{h}_i^{l-1}W^l] \| \mathbf{h}_j^{l-1}W^l]))$$

followed by $\sum_{j\in\{i\}\cup\mathcal{N}(i)} e_{ij}\mathbf{h}_j^{l-1}W^l$, a weighted aggregation. GINConv: The vertex feature vertices of all neighbors of a vertex i are summed and added to ϵ^l times the vertex feature vector of i, where ϵ^l is a learned parameter, using a multilayer perceptron (MLP) with weights W^l and $\mathbf{b}^l$:

$$\mathbf{h}_i^l = \text{MLP}^l\left((1+\epsilon^l)\mathbf{h}_i^{l-1} + \textstyle\sum_{j\in\mathcal{N}(i)}\mathbf{h}_j^{l-1}, W^l, \mathbf{b}^l\right) \quad (1)$$

The activation operator σ (softmax or ReLU), is applied to the aggregated weighted vertex feature vector, yielding the updated $\mathbf{h}_i^l$. For GINConv, activation is built into the MLP.

GINConv concatenates the sum of all vertex feature vectors across all layers to obtain a representation for the graph as

$$\mathbf{h}_G = \Big|\Big|_{l=1}^{L}\left(\textstyle\sum_{i\in G}\mathbf{h}_i^l\right) \quad (2)$$

DiffPool [20] can be combined with any of these GNNs to reduce the volume of data. It uses two GNNs, one to extract vertex embeddings for graph classification, and one to extract embeddings for hierarchical pooling. The embedding GNN at layer l is a standard GNN with Weighting and Aggregation,

$$Z^{l-1} = \text{GNN}_{embed}(A^{l-1}, X^{l-1}); \quad (3)$$

where A^{l-1} is the adjacency matrix of the coarsened graph at level $(l-1)$, and X^{l-1} is the matrix of input cluster features. The pooling GNN generates the assignment matrix:

$$S^{l-1} = \text{softmax}\left(\text{GNN}_{pool}(A^{l-1}, X^{l-1})\right) \quad (4)$$

The number of clusters in layer l is fixed during inference. The coarsened adjacency matrix $A^l = S^{(l-1)T}A^{l-1}S^{l-1}$, and the new embedding matrix $X^l = S^{(l-1)T}\ Z^{l-1}$.

III. Accelerator Architecture

The block diagram of the proposed accelerator is illustrated in Fig. 3, and it consists of the following key components:
(1) **HBM DRAM:** The high-bandwidth memory (HBM) DRAM stores information about the graph. The adjacency matrix of the graph represents its connectivity information and is stored in sparse compressed sparse row (CSR) format. Other formats (CISR [21], C^{2}SR [22], CISS [23]) are not viable candidates as they ignore the underlying graph structure: GNNIE uses adjacency matrix connectivity information to schedule computations and is not a matrix multiplication method.

The sparse input vertex feature vectors are encoded using run-length compression (RLC) [24]. We choose RLC because it is lossless and the decoder has low power/area overhead: this is important because it is only used for the input layer and not thereafter. Alternatives such as CISS have much higher implementation overhead and have been targeted to lock-step systolic arrays, which are unsuitable for Weighting due to the insertion of stalls to handle feature vector sparsity variations.

The DRAM is also used to store intermediate results that do not fit in on-chip memory. High bandwidth options such as HBM or GDDR6 are viable for edge AI [25], [26].
(2) **Memory interface:** The *input buffer* stores vertex features for one pass of the current layer l, i.e., $\mathbf{h}_i^{l-1}$ for vertices i being processed, and the edge connectivity information of the subgraph. Double-buffering is used to reduce DRAM latency: off-chip data is fetched while the PE array computes.

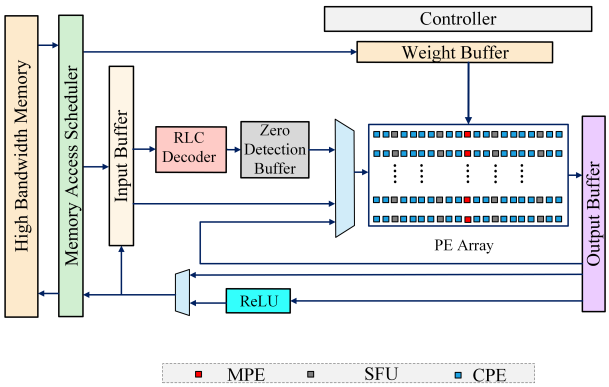

Fig. 3. Block diagram of the proposed architecture.

Sparse data is transmitted from off-chip DRAM to the input buffer using RLC encoding. The input buffer keeps this data in RLC format until it is ready for use, when the data is sent through the *RLC decoder* to the PE array. The RLC decoder is activated for sparse input layer vertex feature vectors, and bypassed for denser feature vectors in later layers.

The *output buffer* caches intermediate results for vertex feature vectors, including the result of multiplication by W^l after Weighting, and the result after Aggregation. The end result is written to off-chip memory. The *weight buffer* holds the values of the weight matrix W^l during Weighting, and, for GAT computations, the attention vector during Aggregation.

The *memory access scheduler* coordinates off-chip memory requests from the input/output/weight buffers.
(3) **An array of processing elements (PEs):** The array consists of an $M \times N$ array of *computation PEs* (CPEs). Each CPE has two scratch pads (spads) and MACs.

Within the array of CPEs, we merge multiple columns of *Special Function Units (SFUs)* (e.g., exp, LeakyReLU, division) [grey blocks], and a row of merge PEs (MPEs) [red blocks]. Interleaved placement allows low latency and communication overhead with CPEs. For exponentiation, we use an accurate, low-area lookup-table-based implementation [27].

Merge PEs (MPEs) are used to aggregate partial results of vertex features sent from the CPE array during Weighting and Aggregation. One MPE is dedicated for each CPE column in the array (Fig. 3), for merging the partial results of vertices. Since these partial results may belong to different vertices we use 16 wires, i.e., one for each CPE, while sending the partial results to the MPEs. A tag is sent along with each partial result to indicate the vertex that the partial result is associated with.

The partial results, along with the tags, are received from the CPEs and stored in the update spad of the MPE. The updated spad can hold 16 such partial results and corresponding tags received from the 16 CPEs of the column. If the tags of partial results match (i.e., if they belong to the same vertex), they are sent to one of the 16 accumulators in the accumulator bank of the MPE to be merged. The result is stored in one of the 16 psum spads with the corresponding tag. The intermediate result stored in the psum spad may be brought into the accumulator again if a partial result with the same tag is found in the updates spad. Following the same procedure, these values are

summed and the result is stored in the psum spad. After merging of the partial results in the update spad is completed, the psum spads send the results and tag to the output buffer. (4) The **Activation unit** performs an activation operation on the vertex features at the final activation stage of computation. (5) The **controller** coordinates operations, including assigning vertex features to the CPE, workload reordering among the CPEs, sending CPE results to the MPEs, sending MPE data to the output buffer, and writing concatenated MPE data.

For a GCN, the layer-wise computation can be written as:

$$\mathbf{h}_i^l = \sigma(\widetilde{A}\mathbf{h}_i^{l-1}W^l) \tag{5}$$

Here, $\widetilde{A} = D^{-1/2}(A+I)D^{-1/2}$ is the normalized adjacency matrix, I is the identity matrix, and $D_{ii} = \sum A_{ij}$. This can be computed either as $(\widetilde{A} \times \mathbf{h}_i^{l-1}) \times W^l$ or $\widetilde{A} \times (\mathbf{h}_i^{l-1} \times W^l)$. The latter requires an order of magnitude fewer computations than the former [18], [28], and we use this approach. Moreover, as $\widetilde{A}$ is highly sparse and shows power-law behavior, we will perform edge-based Aggregation with optimized graph-specific cache replacement policies to limit off-chip accesses.

IV. Mapping Weighting to CPEs

A. Scheduling Operations in the CPEs

We now map the Weighting step, which multiplies the *sparse* feature row vector $\mathbf{h}_i^{l-1}$ with the *dense* weight matrix W^l, to the architecture. The feature vectors are fetched from DRAM, core computations are performed in the CPEs, and the results from the CPEs are assimilated in the MPEs before being written back to DRAM. Our novel scheduling methodology keeps the CPEs busy during the computation, so that Weighting is not memory-bounded. We partition data in two ways (Fig. 5): (1) **Across the vertex feature vector:** We process a **block** of k elements of $\mathbf{h}_i^{l-1}$ at a time, and multiplying it by the corresponding k rows of W^l. This is mapped to a row of the CPE array. With a block size of $k = \lceil F^{l-1}/M \rceil$, the entire feature vector is processed in the CPE array.
(2) **Across vertices:** We process feature vectors for a **set** of s vertices at a time in the PE array, as shown in Fig. 5, where s is constrained by the size of the input buffer. To process all vertices in the graph $G(V,E)$, we process $\lceil |V|/s \rceil$ sets as:

$$\mathbf{h}_i^{l-1}W^l = \Big[\sum_{i=0}^{N-1} \mathbf{h}^{l-1}_{(0:k-1)} W^l_{(0:k-1,i)}, \sum_{i=0}^{N-1} \mathbf{h}^{l-1}_{(k:2k-1)} W^l_{(k:2k-1,i)}, \cdots, \sum_{i=0}^{N-1} \mathbf{h}^{l-1}_{((M-1)k:F^{l-1})} W^l_{((M-1)k:F^{l-1}),i}\Big] \tag{6}$$

where the term in each sum is processed in a separate CPE.

We use a weight-stationary scheme (Fig. 4). Each vertex goes through Weighting set by set, placing k-element blocks of the vertex feature vectors for each set into the input buffer.

We fetch N columns of the weight matrix, W^l, from the DRAM to the weight buffer. A **pass** processes all vertex feature vectors (i.e., processing all vertices in all sets). As shown in Fig. 5, we multiply the vertex feature vectors in all sets with N columns of W^l in the pass. At the end of a pass, the next set of N columns of W^l is loaded. After all passes are completed, the current set of weights is replaced by a new set, and the process continues under the weight-stationary scheme. Within each pass, the CPEs are loaded as follows:

- Each column of W^l is loaded to a CPE column in chunks of k rows, i.e., $W_{(ik:(i+1)k-1,j)}$ is loaded into CPE (i,j).
- For a given set of s vertices, the i^{th} subvectors, of size k, of all s vertex feature vectors are broadcast to the entire CPE row i using a bus. This is indicated by $\mathbf{h}$ in Fig. 4. Since the CPEs in a row work independently of each other and CPE rows do not talk to each other during the Weighting phase, we do not require a complex interconnection scheme: since all CPEs in a row are assigned the same feature vector blocks of length k, we use a bus-based interconnection to broadcast this data to a CPE row.

To leverage input data sparsity, a zero detection buffer is used to detect whether a k-element block that is to be broadcast contains zeros only, so that these computations can be skipped. In case such a block is detected we refrain from broadcasting it to the CPE row. We place zero detection circuitry at the output of the RLC decoder (Fig. 3), at a stage after the k-element blocks are created. The zero-detection function uses a set of OR gates and has minimal hardware overhead.

Benefit of using vertex feature subvector blocks: Our use of k-element blocks instead of the entire vector allows a CPE to skip zero subvectors during pipelined execution and immediately move on to a block from the next available subvector. The next block will be fetched from the input buffer, and under the weight-stationary scheme, it can start computation with the already-loaded weights in the CPE.

The proposed weight-stationary dataflow maximizes the reuse of the weights cached in the weight buffer, which in turn reduces the size requirement of the on-chip weight buffer. Though the feature vectors fetched in the input buffer are get reused, for all datasets evaluated, the computation time for vertices cached in the input buffer is seen to be larger than the memory fetch time under the HBM 2.0 off-chip bandwidth.

B. MPE Processing and Weight Updates

The MAC operation within each CPE generates a partial result for an element of the transformed vertex features. This is sent to the MPE in its column for accumulation over the vertex feature subvectors, along with a tag that denotes its vertex. Due to the irregular completion times for the CPEs, the MPE may accumulate partial sums for several vertices at a time. A bank of psum buffers holds the partially accumulated results: when

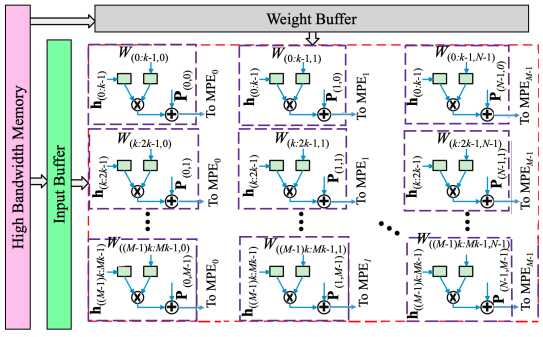

Fig. 4. Weight-stationary linear transformation of vertex features.

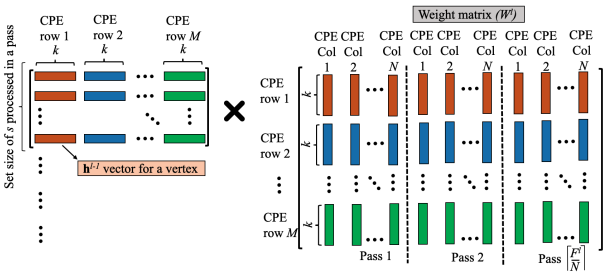

Fig. 5. Mapping Weighting operations to the CPE array.

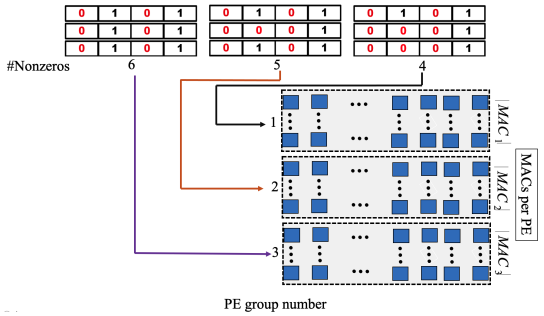

Fig. 6. Workload reordering in flexible MAC (FM) approach.

all partial sums are accumulated for a vertex feature vector, the MPE sends the result to the output buffer, along with the vertex ID i: this is one element of the result of multiplying the feature vector of vertex i and W^l. When all F^l elements are computed, the result is written back to DRAM.

After a CPE column processes all feature blocks for all vertices, the next pass begins. The weights in that column are replaced with the next column of weights from W^l. To overlap computations and keep the CPEs busy, we use double-buffering to fetch the next block of weights from the DRAM to the chip while the CPEs perform their computations.

C. Load Balancing for Weighting

The Weighting computation skips zeros in the vertex feature vector. Vertex feature vectors in the input layer have different sparsity levels (e.g., in Regions A and B of Fig. 2), and this is also true of the k-subvectors. Hence, some k-subvectors are processed rapidly ("rabbits") while others take longer ("turtles"). This causes workload imbalance in the CPE array.

The MPEs that accumulate the results of the CPEs must keep track of psums from a large number of vertices, but they have only limited psum slots for accumulating information. The rabbit/turtle disparity implies that stalls may have to be introduced to stay within the limits of available psum memory in the MPE. As results are accumulated in the output buffer, a larger number of vertex feature vectors must be stored within the buffer, waiting to be completed and written to the DRAM, to account for the disparity between rabbits and turtles.

Flexible MAC (FM) Architecture: We can avoid stalls and speed up computation with more MACs per CPE. Increasing the number of MACs per CPE uniformly throughout the array overcomes the bottleneck of "turtles," but is overkill for "rabbits." Our flexible MAC architecture uses a heterogeneous number of MAC units per CPE in different rows of the array. The CPE array is divided into g row groups, each with an equal number of rows; the number of MACs per CPE, $|MAC|_i$, is monotonically nondecreasing from the first row to the last, i.e., $|MAC|_1 \leq |MAC|_2 \leq \cdots \leq |MAC|_g$. The input buffer has a scheduler that assigns vertex feature vectors to CPE rows. The scheduler uses information about the total nonzero workload for each k-element block of the vertex feature vector to assign the workload to CPE rows. The workloads for the k-element blocks are first binned based on the number of nonzeros, where the number of bins equals the number of CPE groups. Workload binning is carried out as a preprocessing step in linear time on a CPU. The bin with fewest nonzeros is sent to the first CPE group with fewest MACs, and so on; the bin with the most nonzeros is sent to the last CPE row group with the most MACs. After workload binning, each block in a bin is assigned an ID that denotes the CPE row to which it should be broadcast. The scheduler receives these block IDs for the k-element blocks of each feature vector from the host CPU. The input buffer is connected to the embedded scheduler through one port which fetches the block ID information for each feature vector as they are sent over to RLC decoder and eventually the k-element feature vector blocks are broadcast to a CPE row according to their IDs. Since the assignment of IDs to k-element blocks is computed as a preprocessing step, the scheduler does not require any runtime information. The total preprocessing overheads (which include the preprocessing overheads for the linear time binning of k-element blocks) for the four datasets used in our experiment are shown in Table IV. For the Cora, Citeseer, Pubmed, and Reddit datasets, the preprocessing times required for the binning of k-element blocks are, respectively, 5.5%, 4.8%, 3.4%, and 0.7% of the total inference time. It should also be noted that this percentage overhead is lower for the larger datasets (Reddit (233K vertices) has a lower percentage overhead than Cora (2.7K vertices)), indicating the scalability of the solution.

An example of workload reordering among CPE rows is shown in Fig. 6. The CPE array is divided into three groups, Group 1, 2, and 3, where Group i is equipped with $|MAC|_i$ MACs per CPE, where $|MAC|_1 < |MAC|_2 < |MAC|_3$. The vertex feature blocks are binned into three bins that will be assigned to each group. Each bin has several vertex feature blocks: the vertex feature blocks in the left-most bin have the most nonzeros (six), and those of the right-most bin have the fewest of nonzeros (four). We see that the least populated bin is assigned to the group with the fewest MACs, the next to the group with the next number of MACs, and so on.

Load Redistribution (LR): The FM approach does not completely balance the workload. For greater uniformity, we redistribute loads among nearby CPEs. Based on workload distribution in CPE rows, the controller selects pairs of CPE rows to perform workload redistribution, offloading a portion of workload from heavily loaded to lightly loaded CPE rows.

To perform computation on the offloaded workloads, the weights must be transferred with the data. To minimize communication overhead, we first finish the computation in FM, to the point where the current weights are no longer needed, before applying LR. The spads for weights in these CPE rows

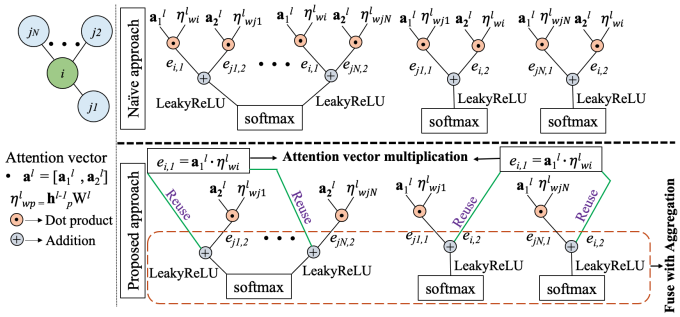

Fig. 7. Reordering of GAT computations.

are loaded with weights for the offloaded workloads.

V. Aggregation Computations

For most GNNs in Section II, Aggregation is a simple summation over the neighbors of the vertex, but GATs require significantly more computation in determining attention coefficients, which are used for weighted aggregation. The first two subsections focus on GAT-specific computations. We then consider Aggregation operations that affect all GNNs.

A. Reordering for Linear Computational Complexity

We present a new method for reordering GAT computations for efficient hardware implementation (Fig. 7). We define the weighted vertex attention vector for vertex p as $\boldsymbol{\eta_w}_p^l = \mathbf{h}_p^{l-1} W^l$. The first step in finding the attention coefficient α_{ij} for neighboring vertices i and j, is to multiply the $2F^l$-dimensional attention vector, $\mathbf{a}^l$, by a concatenation of two F^l-dimensional weighted vertex feature vectors, ($\boldsymbol{\eta_w}_i^l$, $\boldsymbol{\eta_w}_j^l$).

Rewriting $\mathbf{a}^l = [\mathbf{a}_1^l \ \mathbf{a}_2^l]$, where $\mathbf{a}_q^l$ is the subvector that multiplies $\boldsymbol{\eta_w}_q^l$, we can denote this inner product as

$$e_{ij} = \mathbf{a}_1^{l\,T} \cdot \boldsymbol{\eta_w}_i^l + \mathbf{a}_2^{l\,T} \cdot \boldsymbol{\eta_w}_j^l = e_{i,1} + e_{j,2} \tag{7}$$

where $e_{i,1} = \mathbf{a}_1^{l\,T} \cdot \boldsymbol{\eta_w}_i^l$, $e_{j,2} = \mathbf{a}_2^{l\,T} \cdot \boldsymbol{\eta_w}_j^l$, This goes through a LeakyReLU and then a softmax over all neighbors of i to find the normalized attention coefficient,

$$\alpha_{ij} \quad = \quad \text{softmax}\left(\text{LeakyReLU}(e_{ij})\right) \tag{8}$$

As shown in Fig. 7 a naïve approach would fetch $\boldsymbol{\eta_w}_j^l$ from each neighbor j of i, compute e_{ij} using (7), and perform softmax to find α_{ij}. However, since $e_{j,2}$ is required by every vertex for which j is a neighbor (not just i), this would needlessly recompute its value at each neighbor of j. To avoid redundant calculations, we propose to reorder the computation (Fig. 7): for each vertex i, we compute
(a) $e_{i,1} = \mathbf{a}_1^{l\,T} \boldsymbol{\eta_w}_i^l$, used to compute α_{i*} at vertex i.
(b) $e_{i,2} = \mathbf{a}_2^{l\,T} \boldsymbol{\eta_w}_i^l$, used by all vertices j for which i is a neighbor, to compute α_{j*} at vertex j.
Since $\mathbf{a}^l = [\mathbf{a}_1^l \ \mathbf{a}_2^l]$ is identical for each vertex, we calculate $e_{i,2}$ just once at vertex i, and transmit it to vertices j.

For $|V|$ vertices and $|E|$ edges, the naïve computation performs $O(|E|)$ multiplications and memory accesses to $\boldsymbol{\eta_w}_i^l$) per vertex, for a total cost of $O(|V||E|)$. Our reordered computation is $O(|V|+|E|)$, with $O(|E|)$ accumulations over all vertices, i.e., *latency and power are linear in graph size.*

B. Mapping Attention Vector Multiplication

As in Weighting, we use a block strategy to distribute computation in the CPE array. The vector $\boldsymbol{\eta_w}_i$ is distributed across all N columns of a row, so that the size of each block allocated to a CPE for vertex i is $G = \lceil F^l/N \rceil$. Each CPE column processes V_a vertices. Here, V_a depends on the number of columns N in the CPE array, and also depends on the size of the output buffer $|OB|$, i.e., the size of the set of vertices that can be cached in the output buffer: $V_a = |OB|/N$.

This dot product computation is very similar to the weight-stationary scheme used in the Weighting step, i.e., the attention vectors remain stationary until a pass through all the vertices. The F^l-dimensional subvector $\mathbf{a}_1^l$ is divided into N blocks of size G and distributed columnwise to one of the spads in each CPE. Vertex feature blocks for V_a vertices at a time, divided into chunks of size G, are loaded into the other spad, and the inner product computation proceeds. Since $\boldsymbol{\eta_w}_j^l$ and $\mathbf{a}^l$ are dense, load balancing in the CPE array is unnecessary.

As the CPEs in a column finish computation for a vertex, the partial results are sent to the corresponding MPE for Aggregation. We overlap the computation in a CPE column and with the Aggregation in the corresponding MPE: as the MPE aggregates partial results for the current vertex, the blocks of the next weighted vertex features are loaded into the CPE. Thus, all CPEs and MPEs remain busy.

After all V_a vertices in the row are processed, the spad that contains $\mathbf{a}_1^l$ is loaded with $\mathbf{a}_2^l$, and the second inner product computation for the V_a vertices is performed, reusing $\boldsymbol{\eta_w}$. The computed $e_{i,1}$ and $e_{i,2}$ are written back to the output buffer and are appended to the feature vector of vertex i.

C. Mapping Edge-based Computations

The last step requires edge aggregation from each neighbor of a vertex. All GNNs, perform edge-based summations followed by an activation function; for GATs, the weights for this summation are computed using methods in the above subsections.

Typical graphs are too large for the on-chip buffers. We use a dynamic scheme (Section VI) to process a subgraph of the graph at a time, processing edges in parallel in the CPE array.
Load Distribution: The Aggregation computation brings data into the input buffer. For each vertex in the subgraph corresponding to the vertices in the buffer, it accumulates edge data by pairwise assignment to CPE spads.

Due to power-law behavior, the vertex degrees in the subgraph may have a large range. To distribute the load, the Aggregation summations are divided into unit pairwise summations and assigned to CPEs. For instance, accumulation of a sum effectively implements an adder tree in which the number of CPEs required to process Aggregation for each vertex depends on its degree in the subgraph. Thus, the number of CPEs assigned for Aggregation of a vertex in a subgraph is proportional to its degree. The degree-dependent assignment of CPEs to vertices tackles imbalance in workload that might occur due to the power-law behavior.
GATs: The final step in computing the attention coefficient α_{ij} involves edge-based computations (Equation (8)):

- the addition, $e_{ij} = e_{i,1} + e_{j,2}$

- a LeakyReLU step, LeakyReLU(e_{ij})
- a softmax step, $\exp(e_{ij})\boldsymbol{\eta_w}_j / \sum_{k\in\{i\}\cup N(i)} \exp(e_{ik})$

Each edge from a neighbor j to vertex i contributes an e_{ij} to the numerator of the softmax, and one to the denominator. These computations are parallelized in the CPEs among incoming edges of a vertex using pull-based aggregation [29].

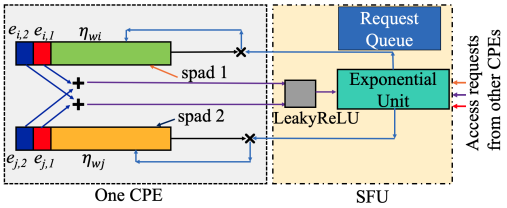

Fig. 8. Data flow corresponding to computation of an edge.

The computation of numerator in the softmax step is shown in Fig. 8. For a target vertex i connected to a neighbor j by edge (i, j), $\boldsymbol{\eta_w}_i$, $e_{i,1}$, and $e_{i,2}$, are loaded into one spad of a CPE, and the corresponding data for j into the other spad. For vertex i, the result $e_{i,1} + e_{j,2}$ is sent to the SFU to perform LeakyReLU followed by exponentiation. The output returns to the CPE and is multiplied with $\boldsymbol{\eta_w}_j^l$. A similar operation is performed for vertex j to compute $\exp(e_{ji})\boldsymbol{\eta_w}_i^l$.

Other GNNs: The Aggregation step for GCN, GraphSAGE, GAT and GINConv involves a sum of weighted vertex feature vectors over all neighbors j (or a sample of neighbors for GraphSAGE) of each vertex i. This computation is similar to but simpler than that in Fig. 8: just addition is performed.

As before, a subgraph of the larger graph is processed at a time. In processing vertex i, the data for all neighbors j is processed in an adder tree, placing operands in spad1 and spad2 of a CPE, and storing the result in spad1. The partial results for a vertex (partial sum for a general GNN, or the summed numerator and softmax denominator for a GAT) are written to the output buffer after each edge computation. For a GAT, the values of $\exp(e_{ik})$ are also added over the neighborhood to create the denominator for the softmax. Finally, the accumulation over neighbors is divided by the denominator, in the SFU to obtain the result. Similarly, in the another round of accumulation the partial results of the vertices are sent form the output buffers to CPEs to compute the final result. When all components of the sum for vertex i are accumulated, the result is sent through the Activation unit and written to DRAM.

VI. Graph-Specific Caching

Aggregation operations intensively access the graph adjacency matrix. Computational efficiency requires graph-specific caching techniques to transfer data to/from on-chip input and output buffers, maximizing data reuse and minimizing off-chip random memory accesses. A notable feature of our proposed policy is a guarantee that *all random-access patterns are confined to on-chip buffers* and *off-chip fetches are sequential*.

As stated earlier, the adjacency matrix is stored in the CSR format. Our input is a graph represented by three arrays: (i) the coordinate array lists the incoming/outgoing neighbors of each vertex, (ii) the offset array contains the starting offset of each vertex in the coordinate array, and (iii) the property array with the weighted vertex feature, $\boldsymbol{\eta_w}_i^l$ (see Section V-A), for each vertex i; for GATs, this is concatenated with $\{e_{i,1}, e_{i,2}\}$.

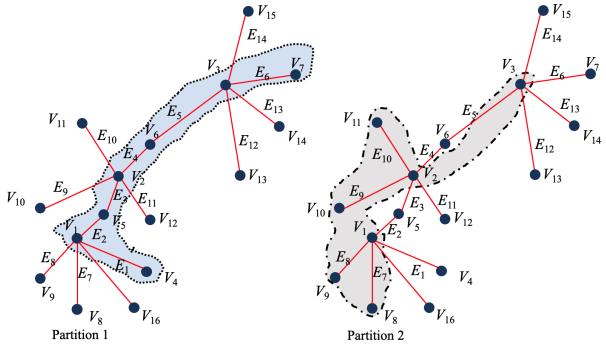

Fig. 9. Example illustrating the subgraph in the input buffer (left) and its evolution after cache replacement (right).

Subgraph in the Input Buffer: Edge-mapped computations involve a graph traversal to aggregate information from neighbors. At any time, a set of vertices resides in the input buffer: these vertices, and the edges between them, form a subgraph of the original graph. In each iteration, we process edges in the subgraph to perform partial Aggregation operations (Section V-C) for the vertices in the subgraph. Under our proposed caching strategy, ultimately all edges in the graph will be processed, completing Aggregation for all vertices.

We illustrate the concept through an example in Fig. 9, showing a graph with vertices V_1 through V_{16}. The highest degree vertices are first brought into the cache, i.e., the input buffer: vertices V_1, V_2, and V_3 of degree 5, vertices V_5 and V_6 of degree 2, and then two vertices of degree 1, V_4 and V_7. The subgraph, Subgraph 1, consists of these vertices and edges E_1 to E_6 which connect them. After edges E_1 through E_6 are processed, vertices V_4 through V_7 have no unprocessed edges and may be replaced in the cache by V_8 through V_{11} in Iteration 2. This creates Subgraph 2, the subgraph with edges E_7 through E_{10}), which is processed next, and so on.

Cache Replacement Policy: As vertices are replaced after computation of each subgraph, a replacement policy is necessary. Our policy prioritizes vertices with the most *unprocessed* edges for retention in the input buffer. Since such vertices appear more frequently in the list of neighbors for other vertices in the coordinate array, this increases the likelihood of finding both the source and destination of edges in the cache.

The policy requires inexpensive preprocessing to sort vertices in order of their degrees. In practice, it is enough to sort vertices into bins based to their degrees, differentiating high-degree vertices from medium-/low-degree vertices to prioritize higher-degree vertices. After preprocessing, vertices of the input graph are stored contiguously in DRAM in descending degree order of the bins. Ties are broken in dictionary order of vertex IDs. *The key to avoiding random-access fetches from DRAM is the preprocessing step and the replacement policy.*

We track the number of unprocessed edges, α_i for vertex i, decrementing it as each neighbor is processed. Initially α_i is the vertex degree; when $\alpha_i = 0$, $\mathbf{h}_i^l$ is fully computed. Tracking α_i requires minimal hardware overhead (a decrementer

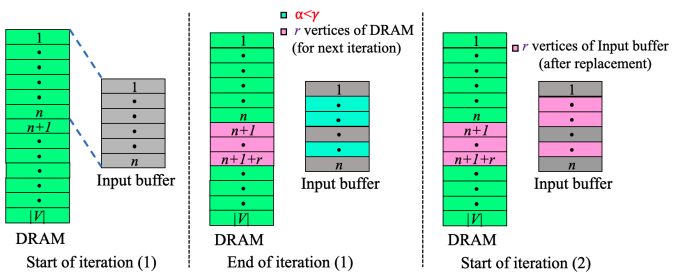

Fig. 10. Input buffer replacement policy during Aggregation.

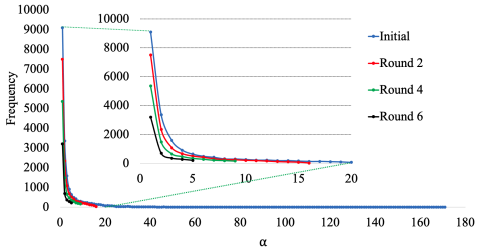

Fig. 11. Histogram of α through various Rounds (Pubmed). The inset shows a magnified view.

and one word of storage per vertex), and its tracking enables GNNIE to maximize edge processing in each iteration.

Fig. 10 illustrates our policy, managed by a cache controller using a 4-way set associative cache. Graph vertices are stored contiguously in DRAM in descending degree order, where vertex 1 has the highest degree. If the input buffer capacity is n vertices, initially data (i.e., feature vector, connectivity information, α_i) for vertices 1 to n are loaded from DRAM.

The algorithm processes each such set of vertices in the input buffer in an iteration. We track α_i for vertex i, decrementing it as each neighbor is processed. Tracking α_i requires minimal hardware overhead (a decrementer and one word of storage per vertex). Initially α_i is the vertex degree. At the end of iteration 1 (after finishing computation of the subgraph of the first n vertices), if $\alpha_i < \gamma$ for any vertex, where γ is a predefined threshold, it is replaced from the cache. We replace r vertices in each iteration using dictionary order. These vertices are replaced in the input buffer by vertices $(n+1)$ to $(n+1+r)$ from DRAM: these have the next highest vertex degrees. For each such vertex i, we write back the α_i value into DRAM. When all vertices are processed once, we have completed a **Round**.

Similarly, the partial sums for the vertex feature vector in the output buffer are updated as more edges in the subgraphs are processed. Any $\mathbf{h}_i^l$ for which all accumulations are complete is written back to DRAM. Due to limited output buffer capacity, and only a subset of partial vertex feature vector sums can be retained in the buffer, and the rest must be written to off-chip DRAM. To reduce the cost of off-chip access, we use a degree-based criterion for prioritizing writes to the output buffer vs. DRAM. As partial Aggregation results for softmax are written to DRAM, the numerator and denominator components for a vertex are stored nearby, for locality during future fetches.

How our policy avoids random-access DRAM fetches: Our policy makes random accesses only to the input buffer; all DRAM fetches are sequential. In the first Round, data is fetched from consecutive DRAM locations. In the CPE array, aggregation of each vertex fetches the vertex feature data of its neighbors in the current subgraph in the cache. Each vertex feature vector may be thus fetched by the CPE array multiple times according to the graph neighborhood structure, but all such random accesses are limited to the cache, which has much better random-access bandwidth than the off-chip memory.

Vertices evicted from the cache, with $\alpha_i < \gamma$, may be fetched again in a subsequent Round. Even in these Rounds, data blocks are brought into cache in serial order from DRAM: there are no random accesses from DRAM. During DRAM fetches, a cache block is skipped if all of its vertices are fully processed. The total unprocessed edges in a cache block is tracked through inexpensive hardware, similar to tracking α_i.

The effectiveness of the approach is illustrated in Fig. 11, which shows the histogram of α_i distributions in the input buffer after each Round. The initial distribution corresponds to the power-law degree distribution, and in each successive Round, the histogram grows flatter – with both the peak frequency and maximum α becoming lower, thus mitigating the problems of power-law distribution. In contrast, HyGCN ignores the power-law problem, and AWB-GCN overcomes it using high inter-PE communication. Moreover, our approach is shown to be effective even for much more intensive GAT computations (prior accelerators do not address GATs).

Fig. 12 shows the impact of γ on DRAM accesses for three datasets during Aggregation of first layer. For the calculation we use the weighted feature vector size at first layer to 128 B. As γ increases, more vertices are evicted and may have to be brought back to the cache, resulting in more DRAM accesses. However, if γ is too low, vertices may not be evicted from the cache, resulting in deadlock as new vertices cannot be brought in. In our experiments, we use a static value $\gamma = 5$, but in practice, γ may have to be changed dynamically when deadlock arises. Deadlock detection is inexpensive and is based on the number of total unprocessed edges in the partition, which is monitored by a counter, and this dynamic scheme will be inexpensive in hardware.

VII. Related Work

There has been much work on CNN accelerators [7]–[13], but these are not efficient for processing GNNs. Graph analytics accelerators include ASIC-based (Graphicionado [14], GraFBoost [16]), FPGA-based (FPGP [15]) and in-memory (GraphPIM [30]) platforms. However, graph accelerators target lightweight operations, do not focus on data reuse, and would be challenged by computation-intensive GNNs.

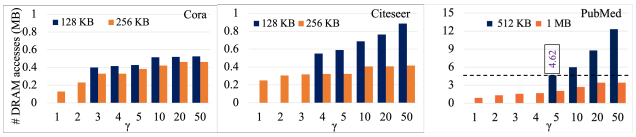

Fig. 12. Ablation study on γ: (a) Cora (b) Citeseer (c) Pubmed.

Software frameworks for GNNs include Deep Graph Library, AliGraph, and TensorFlow. Some hardware accelerators have been proposed, but we know of no prior work that can handle networks that require softmax nonlinearities on graphs such as GATs. Although some GAT computations are addressed in [19], the crucial attention normalization step is left out. To our knowledge, no methods handle extreme input feature vector sparsity using graph-specific methods.

HyGCN [17] uses an Aggregation engine for graph processing and a Combination engine for neural operations. This requires separate on-chip buffers for each engine, which are not fully utilized due to with workload imbalance at different stages of computation. HyGCN must arbitrate off-chip memory access requests coming from on-chip buffers of two different engines, which involves complicated memory access control. Using a single hardware platform optimized to handle both the irregular graph computation and compute-intensive, albeit regular, DNN computation, GNNIE achieves performance gains over HyGCN. Moreover, HyGCN uses sharding with window sliding/shrinking to reduce random memory access during Aggregation: this has (1) limited efficacy for highly sparse adjacency matrices as the number of overlapping neighbors of vertices is a small farction of the total number of vertices in a shard, undermining its efficacy; moreover, no specific effort is made to address power-law degree distributions. (2) limited parallelism, as the sliding window of the current shard depends on the shrinking of the previous shard; HyGCN does not fully leverage data reuse opportunities of high-degree vertices during Aggregation, performing $(\tilde{A}\mathbf{h}_i^{l-1})W^l$, instead of the cheaper $\tilde{A}(\mathbf{h}_i^{l-1}W^l)$ [18], [28]. Input feature vector sparsity is not addressed and can result in inefficiency due to stalls. These factors explain GNNIE's speedups over HyGCN.

BlockGNN [31] optimizes Weighting by applying an FFT and block-circulant constraint on the weight matrix. Similar to HyGCN [17], GNNerator [32] and DyGNN [33] pipeline separate engines for Aggregation and Weighting. Thus susceptible to stalls due to (i) unbalanced loads between engines (ii) workload variations in each engine due to variable input feature vector sparsity and power-law degree distributions. DyGNN also employs pruning to reduce vertex/edge redundancy.

AWB-GCN [18], which is limited only to GCNs and not general GNNs, views the problem as a set of matrix operations. It does not specifically try to reduce random memory accesses due to the highly sparse graph adjacency matrix. Its dynamic scheduling scheme in AWB-GCN for workload redistribution among PEs may incur high inter-PE communication, degrading energy efficiency. The GCNAX/SCGNAX approaches [34], [35] also propose a matrix-multiplication-based approach and handle only GCNs, and their results show reducing speedups as the dataset sizes increase. EnGN [28] uses a ring-edge-reduce (RER) dataflow for Aggregation, where each PE broadcasts its data to other PEs in the same column. To reduce communication, EnGN reorders the edges, but this is an energy-intensive step, undermined by high sparsity in the adjacency matrix, that occurs frequently as the limited number of cached edges are replaced. The scheme has large preprocessing costs. In the literature, reconfigurable PE array designs (e.g., Planaria [36], RecPipe [37]) have been proposed for DNN acceleration. However, focus of these works are orthogonal to GNNIE's approach. For instance, Planaria targets dynamic architecture fission for spatial multi-tenant execution and RecPipe focuses on optimizing multi-stage recommendation inference.

Prior accelerators have not fully explored load balancing. Methods that offload tasks to idler PEs (ring-edge-reduce [28], multistage networks [18]) involve high communication and control overheads. GNNIE bypasses such approaches and uses the flexible MAC architecture for load balancing, using heterogeneous PEs, and assigning computation according to need. The idea is simple, effective, and easily implemented. In addition, as stated in Section IV-C the load redistribution scheme of GNNIE results in low inter-PE communication, low control overhead, and high speedup gain for the hardware overhead (Fig. 19). Preprocessing is cheap and involves linear-time binning of vertex features blocks into groups.

Frequency-based caching techniques for graph data have been proposed in [38] using a programming interface. However, [38] is a purely software-based framework, agnostic to the underlying hardware, for traditional graph analytics and uses a static approach. GNNIE uses a hardware-centric *dynamic* frequency-based caching scheme that tracks the α value for each vertex with minimal hardware overhead, and ensures serial access to DRAM. Other schemes are also static and more computational than GNNIE: they use hashing functions [39] or perform more computation [40], [41] in finding static communities/partitions that do not specifically address cache size. On the other hand, GNNIE's computationally cheap dynamic scheme automatically adapts to the cache size using subgraphs built from vertices in the cache. GRASP [42], another cache management scheme for graph analytics, employs a most-recently-used (MRU) approach. However, this scheme is based on past history, while GNNIE's use of the unprocessed vertex count measures future potential for a vertex.

VIII. Evaluation

A. Experimental Setup

Accelerator Simulator: We develop a simulator to measure the execution time in terms of the number of cycles required. The simulator models each module of GNNIE and integrated with Ramulator [43] to model the memory access to the off-chip HBM with 256 GB/s bandwidth.

Each module was implemented and synthesized in Verilog and the synthesized design was verified through RTL simulations. Synopsys Design Compiler was used to synthesize the accelerator at 32nm technology node with standard VT cell library. The chip area, critical path delay, and dynamic/static power, extracted from Design Compiler, are used for evaluating performance and energy. CACTI 6.5 is used to estimate the area, energy consumption, and access latency of on-chip buffers. The energy of HBM 2.0 is 3.97 pJ/bit [44]. The chip area is 15.6mm^2 and its frequency is 1.3 GHz.

Benchmark GNN Datasets and Models: For evaluation of the performance of GNNIE, we used the benchmark graph

Table II: Dataset information [45]

Dataset	Vertices	Edges	Feature Length	Labels	Sparsity
Cora (CR)	2708	10556	1433	7	98.73%
Citeseer (CS)	3327	9104	3703	6	99.15%
Pubmed (PB)	19717	88648	500	3	90%
Reddit (RD)	232965	114.6M	602	41	48.4%

Table III: Convolution layer configurations (len[$\mathbf{h}_i^l$] = length of $\mathbf{h}_i^l$)

GNN Model	Weighting	Aggregation	Sample size
GAT	len[$\mathbf{h}_i^l$] , 128	Sum	−−
GCN	len[$\mathbf{h}_i^l$] , 128	Sum	−−
GraphSAGE	len[$\mathbf{h}_i^l$] , 128	Max	25
GINConv	len[$\mathbf{h}_i^l$] , 128 / 128	Sum	−−
DiffPool (GCN_{pool})	len[$\mathbf{h}_i^l$] , 128	Sum	−−
DiffPool ($GCN_{embedding}$)	len[$\mathbf{h}_i^l$] , 128	Sum	−−

datasets listed in Table II. We used five GNN models for evaluations, i.e., GAT, GCN, GraphSAGE, GINConv, and DiffPool. The convolution layer configurations are shown in Table III. All preprocessing costs are included in the evaluation.

Configurations for Baseline/Cross-Platform Comparison: We first compare GNNIE against two baseline architectures, i.e., a general-purpose CPU and a GPU. The CPU platform is equipped with Intel Xeon Gold 6132@2.60GHz and 768 GB DDR4. The GPU platform is equipped with V100 Tesla V100S-PCI @1.25GHz and 32 GB HBM2.

For GNNIE, the sizes of output and weight buffers are 1MB and 128KB, respectively. The input buffer size is 256KB for the smaller datasets (CR, CS) and 512KB for the larger datasets (PB, RD). The area and power numbers reported later correspond to the larger input buffer size. The output buffer is larger since it must cache many partial results before they are aggregated, particularly for high-degree vertices. For a 1-byte weight, for the dataset with the largest feature vector (∼4K for CS), to keep 16 CPE columns occupied, the buffer size is 4K×16×2 (for double-buffering) = 128KB.

The 16×16 CPE array consists of four MAC units for CPE row number 1 to 8, five MAC units for CPE row number 9 to 12 and six MAC units for CPE row number 13 to 16. The heterogeneous CPE array is blockwise regular and is friendly to back-end physical design. The number of MACs per CPE was chosen through design space exploration, optimizing the cost-to-benefit ratio (speedup gain : hardware overhead).

B. Baseline Platform Comparisons

Performance comparisons with CPU and GPU: To make a fair performance comparison with the general-purpose CPU and GPU, we implement the GNN models with the PyTorch Geometric (PyG) software framework. The PyG-based implementations for CPU and GPU used in our experiment are denoted as PyG-CPU and PyG-GPU, respectively. Neighborhood sampling for GraphSAGE is based on cycling through a pregenerated set of random numbers. Table IV shows the absolute run time including the preprocessing overheads for the five datasets across the GNN models used in our experiment. The total preprocessing time (including degree-based vertex reordering (Aggregation: Section VI) and workload reordering (Weighting: Section IV-A) and neighborhood sampling time

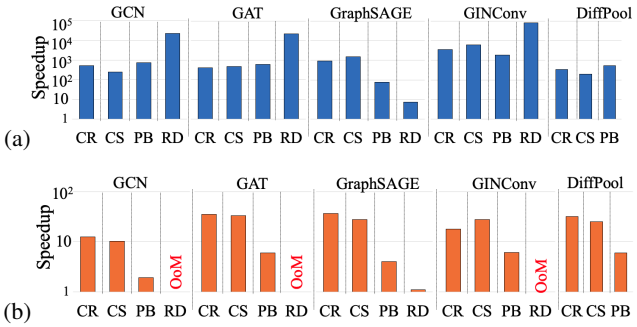

Fig. 13. GNNIE performance vs. (a) PyG-CPU (b) PyG-GPU.

for GraphSAGE) is shown in the parenthesis along with the run time. It can be seen that the total preprocessing time is very negligible for smaller datasets (CR and CS) and a small percentage of run time for larger datasets (RD).

Table IV: Absolute run time of inference

Dataset	GCN	GAT	GraphSAGE	GINConv
CR	45.80 μs (27.50 μs)	53.80 μs (27.50 μs)	54.70 μs (34.7 μs)	56.01 μs (27.5 μs)
CS	49.50 μs (26.40 μs)	62.60 μs (26.40 μs)	61.30 μs (37.40 μs)	66.40 μs (26.40 μs)
PB	0.25 ms (94.60 μs)	0.39 ms (94.60 μs)	0.36 ms (0.21 ms)	0.34 ms (94.60 μs)
RD	10.31 ms (0.31 ms)	12.32 ms (0.31 ms)	83.12 ms (74.9 ms)	11.31 ms (0.31 ms)

As shown in Fig. 13(a), the average speedup of GNNIE over the PyG-CPU across the datasets used in our experiment for GCN, GAT, GraphSAGE, GINConv, and DiffPool are 6229×, 5894×, 625×, 22878×, and 359×, respectively. According to Fig. 13(b) the average speedup of GNNIE over the PyG-GPU across the datasets used for GCN, GAT, GraphSAGE, GINConv, and DiffPool are 8.25×, 24.67×, 17.53×, 17.37×, and 21×, respectively. The speedup calculations take into account the total preprocessing times mentioned in Table IV.

The speedup comes from several GNNIE optimizations: (i) The segmentation of vertex feature vectors and their assignment in our FM architecture tackles the feature vector sparsity challenge. (ii) Our degree-aware cache replacement policy avoids random memory accesses to DRAM. (iii) During Weighting, distributed computation across multiple batches enables weight reuse. Note that PyG-CPU and PyG-GPU do not allow our dynamic caching scheme to be implemented within their purely software based frameworks. The speedup of GNNIE on GINConv is further enhanced because of PyTorch Geometric executes Aggregation before Weighting: as described in Section III, this requires more computation than the reverse order of computation used in GNNIE.

For the GraphSAGE speedup calculations, the neighborhood sampling time on PyG-CPU/PyG-GPU is excessive and is excluded (for RD it is 13s whereas the execution time is 0.35s for PyG-CPU and 0.003s for PyG-GPU), but GNNIE runtimes include neighborhood sampling times. This results in lower speedup compared to PyG-GPU for RD. However, the GPU is much more power-hungry than GNNIE, e.g., it requires 98.5× more energy for GraphSAGE/RD than GNNIE. GNNIE is scalable on PyG-CPU: for GCN, GAT, and GINConv, the speedups generally increase with benchmark size. GraphSAGE bucks this trend for the above reasons, but while its sampling scheme improves scalability, it reduces accuracy [3], [46].

On PyG-GPU, the speedups do not monotonically improve with the number of nodes. This is because larger datasets (e.g., PB) reap greater benefit from GPU parallelization: for these datasets, GNNIE vs. PyG-GPU speedup decreases whereas GNNIE vs. PyG-CPU speedup increases. It is important to note that the GPU comparison is not entirely fair to GNNIE's lightweight accelerator with low on-chip memory, targeted to edge applications. In contrast, this GPU has a $\sim$20$\times$ larger on-chip memory than GNNIE and its power-hungry nature makes it impractical for the edge. Nevertheless, GNNIE shows speedups over even this powerful GPU.

C. Cross-platform Comparisons

We conduct cross-platform performance comparisons with HyGCN and AWB-GCN. Neither computes exponentiation for softmax, required by GATs, and AWB-GCN only implements GCN. Thus, for GCN we perform a comparison with HyGCN and AWB-GCN. For GraphSAGE and GINConv we also show a comparison with HyGCN. Unlike the original implementations, HyGCN uses 128 channels for hidden layers of all the GNN models, and therefore we have also configured the hidden layers similarly (Table III). To compare with HyGCN, AWB-GCN runs the customized GCN model with 128 channels for hidden layers on a E5-2680v3 CPU with PyG and reports relative speedup and inference latency. We leverage inference latency data from AWB-GCN for our comparison.

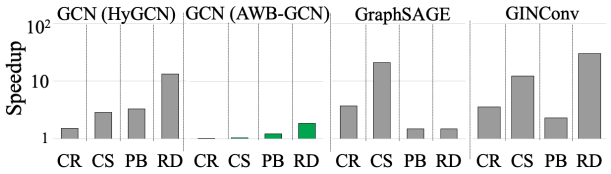

Fig. 14. Performance comparison with HyGCN and AWB-GCN.

To compute speedup over HyGCN for GraphSAGE, GINConv, and DiffPool we run the GNN models on Intel Xeon Gold 6132@2.60GHz CPU, which has similar performance as the E5-2680v3@2.50GHz CPU, and determine the relative speedup of GNNIE. We then take a ratio of the computed relative speedup with the relative speedup of HyGCN compared to E5-2680v3 CPU. It should be noted that PyG framework is used to optimize both the baseline CPUs of HyGCN and GNNIE. Fig. 14 shows compared to HyGCN, GNNIE achieves average speedup of 5.23$\times$, 6.81$\times$, and 3.1$\times$ for GCN, GraphSAGE, GINConv, respectively. A comparison for DiffPool is not possible: HyGCN does not report results on the widely used datasets that we evaluate. As before, these speedup comparisons include GNNIE preprocessing costs. Even though the on-chip buffer size of HyGCN (24 MB + 128 KB) is much larger than GNNIE (1.7 MB), GNNIE shows an average speedup of 5.05$\times$.

AWB-GCNs scatter-based-aggregation requires 3x larger on-chip buffers than GNNIEs gather-based-aggregation. Sparse matrix-vector-multiplication-based AWB-GCN loses graph-adjacency view, sacrificing efficiency. GNNIEs caching scheme VI specifically leverages graph adjacency to reduce expensive random DRAM accesses. For GCNs, GNNIE (with

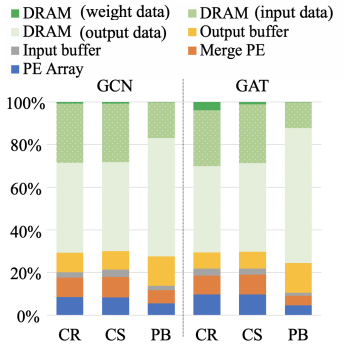

Fig. 15. Energy breakdown for GCN and GAT.

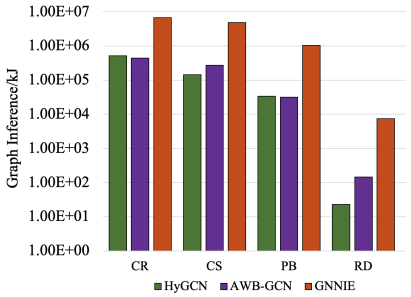

Fig. 16. Energy efficiency: GNNIE vs. HyGCN, AWB-GCN.

3.4x fewer PEs) shows 1.3x speedup (Fig. 14) over AWB-GCN and 15-51x higher (Fig. 16) inferences/kJ. We note that AWB-GCN results are reported on an FPGA, which is likely to be slower and more power-hungry than an ASIC.

D. Throughput and Energy Comparisons

Table V shows the throughput for various datasets for our configuration of GNNIE. The table shows that the throughput degrades only moderately as the graph size is increased.

The power dissipation of GNNIE is 3.9W in 32nm, lower than HyGCN (6.7W in 12nm), similar to recent CNN edge inference engines (Edge TPU, Hailo-8, InferX1). Fig. 15 shows the energy breakdown for GNNIE for GAT and GCN across three datasets, including DRAM energy required to supply the output, input, and weight buffers. The output buffer has the most of transactions with DRAM due to psum storage. On-chip weight buffer energy is negligible and not shown.

Fig. 16 compares GNNIE's energy efficiency with prior works. The efficiency ranges from ranges from 2.3×10^1 – 5.2×10^5 inferences/kJ for HyGCN and 1.5×10^2 – 4.4×10^5 inferences/kJ for AWB-GCN. GNNIE clearly outperforms the others, going from 7.4×10^3 – 6.7×10^6 inferences/kJ.

E. DRAM Access Analysis

To illustrate the efficiency of the proposed graph-specific caching scheme we compare the number of DRAM accesses

Table V: Throughput for various datasets for GNNIE.

Peak	Cora (CR)	Citeseer (CS)	Pubmed (PB)	Reddit (RD)
3.16 TOPS	2.88 TOPS	2.69 TOPS	2.57 TOPS	2.52 TOPS

required by GNNIE with those in the widely used 2-D graph partitioning method (employed by GridGraph [47], HyGCN [17], Marius [48]). In 2-D graph partitioning, vertices of the graph are divided into u equal-sized disjoint partitions and stored in DRAM. Edges are then grouped into u^2 blocks that can be viewed as a grid. In this grid, each edge block (p, q) contains the edges for which source nodes belong to the p^{th} vertex partition and destination nodes belong to q^{th} partition. In this scheme, except for the self-edge blocks (e.g., edge block (p, p)) vertex partition p and q must be in the cache (input buffer) together at least once to process the corresponding edge block (p, q).

If the input buffer can hold v vertex partitions at a time ($u \geq v$), a lower bound on the number of DRAM block accesses for processing the graph using 2-D partitioning is [48]):

$$\left\lceil \left(\frac{u(u-1)}{2} - \frac{v(v-1)}{2} \right) \Big/ (v-1) \right\rceil \tag{9}$$

To compare the caching schemes of GNNIE and 2-D graph partitioning we evaluate the DRAM accesses required for executing Aggregation of the first layer for the Pubmed dataset. In our experiment we use a 512 KB input buffer and the size of each vertex feature vector is set to 128 B. For the 2-D partitioning scheme, we vary the number of vertex partitions in DRAM (u) from 2 to 100 in steps of of 1 and compute the corresponding lower bound on the number of DRAM access for 2-D partitioning using (9). The lower number is multiplied with the size of each vertex partition in the input buffer to determine the DRAM accesses in MB. To calculate the each vertex partition size the input buffer size is divided by v. In Fig. 17, the x-axis denotes the number of vertex partitions in DRAM (u) and the y-axis shows the corresponding lower bound for 2-D partitioning on the DRAM access required (in MB) to process the graph. From Fig. 17 we can see that initially, the lower bound on the DRAM accesses decreases with the number of partitions and plateaus eventually for higher values of u. For $u = 100$, the lower bound is 5.59MB.

The static caching scheme proposed in 2-D graph partitioning must go through all the vertex pair combinations to process all the edges. Due to the power-law behavior and sparsity of real-world graphs, not all vertices in a vertex partition are used to process the edges of its corresponding edge blocks. However, processing of an edge block requires all vertices of the corresponding vertex partition to be cached in this scheme. Since this approach does not make any effort to distinguish between the useful vertices of a vertex partition to process the edge blocks, it incurs redundant DRAM access and provides suboptimal performance in reducing DRAM accesses.

On the other hand, as shown in Fig. 12(c) and Fig. 17 for $\gamma = 5$ GNNIE requires 4.62 MB of DRAM access to execute the first layer Aggregation of Pubmed dataset. In GNNIE the number of vertex feature vectors that get replaced after each iteration dynamically varies according to the α of cached vertices and γ. In each iteration, GNNIE tries to maximize the number of edges being processed by retaining the vertices with a higher potential of being reused in the next iteration. Thus, by dynamically tuning the retentivity of cached vertices at each iteration to maximize their reuse the proposed graph-specific caching scheme leads to lower DRAM accesses compared to calculated lower bound of 2-D partitioning.

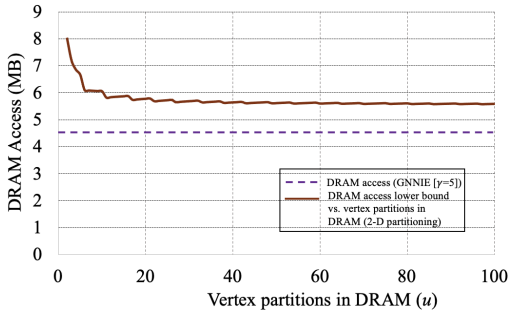

Fig. 17. Comparison of DRAM access of GNNIE with the lower bound on DRAM access vs vertex partitions in DRAM of 2-D graph partitioning.

F. Optimization Analysis

We analyze key optimization techniques applied in GNNIE. To evaluate these techniques we select a baseline design (**Design A**) which uses four MACs per CPE uniformly. Parameters for the flexible MAC architecture and on-chip buffer sizes for all designs are as described at the end of Section VIII-A. The dimension of the PE array in all cases is 16×16.

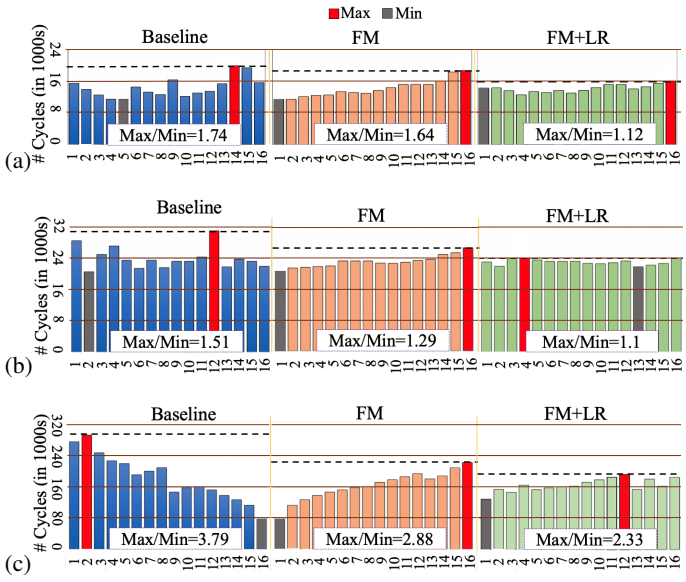

Fig. 18. CPE row workload in Weighting: (a) Cora (b) Citeseer (c) Pubmed.

Optimizing Weighting Time: We first analyze the performance improvement of applying flexible MACs (**FM**) on the baseline design during Weighting. For the Cora, Citeseer, and Pubmed datasets, the workload distribution among the CPE rows for the baseline (without load-balancing) and FM designs are shown in Figs. 18(a), (b), and (c), respectively. Due to vertex feature sparsity, the CPE rows in the baseline design suffer from workload imbalance. The FM design smooths the workload distribution among the CPE rows results in 6% (Cora), 14% (Citeseer), and 24% (Pubmed) reduction in the number of cycles required to compute 16 elements of

the output vertex features during Weighting. The imbalance between the maximum and minimum is also reduced by FM.

For all datasets, the last four CPE rows require more cycles than others (heavily loaded CPE rows) and the first four CPE rows finish computation earlier (lightly loaded rows) in FM. We perform load redistribution (**LR**) between "LR pairs" of heavily loaded and lightly loaded CPE rows, offloading a portion of the workload from the heavily loaded CPE row to the lightly loaded one. The figure shows that applying LR on FM further smooths the workload distribution, reducing the imbalance between the maximum and minimum significantly, and also further reduces the number of cycles.

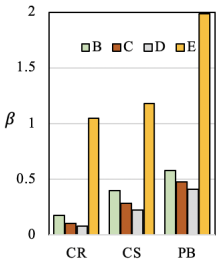

Fig. 19. Cost/benefit ratio for adding MACs in Designs B–E.

Cost/Benefit Ratio: We introduce a metric, the cost/benefit ratio, β, relative to Design A with 1024 MACs (4 MACs/CPE)

$$\beta = (\%\text{ reduction in Cycles})/(\%\text{ increase in MACs}) \quad (10)$$

The percentage reduction in cycles required is measured for Weighting for various choices of MAC counts. The additional hardware overhead is measured in terms of percentage increase in MACs compared to the baseline design. We compute β for four designs. These design choices are as follows: (i) 5 MACs per CPE (**i.e., Design B**, 1280 MACs in all), (ii) 6 MACs per CPE (**i.e., Design C**, 1536 MACs in all), (iii) 7 MACs per CPE (**i.e., Design D**, 1792 MACs in all), (iv) flexible MAC architecture for GNNIE, described at the end of Section VIII-A (**i.e., Design E**, 1216 MACs in all).

Fig. 19 plots β on the three datasets used in our experiment for the four design choices. As MAC units are added uniformly to the baseline design β drops and is lowest for Design D across all datasets. β drops for Designs B, C, and D as the high sparsity and sparsity variation among vertex features yield low speedup gains as more MACs are added. By employing MACs among CPE rows as needed, the FM approach tackles input vertex feature sparsity, achieving high β across all datasets.

Optimizing Aggregation Time: Our baseline design has 4 MACs/row (no FM), no load balancing (i.e., no degree-dependent load distribution in Aggregation), and no graph-specific caching (i.e., vertices are processed in order of ID).

We first evaluate our degree-aware graph reordering and our proposed cache replacement policy (**CP**). We measure the execution time of the baseline during Aggregation with and without CP. Fig. 20(left) shows that CP reduces Aggregation time by 11% (Cora), 35% (Citeseer), and 80% (Pubmed). This is due to reduced random off-chip memory accesses as more edges in a subgraph are processed under degree-aware caching.

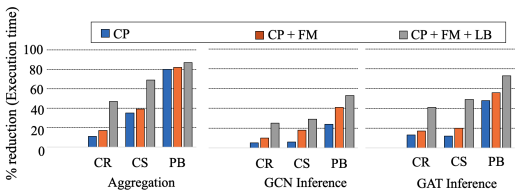

Fig. 20. Effectiveness of GNNIE's optimization methods.

Next, we apply CP over FM to measure their combined effect. From Fig. 20(left), the added MACs in CP + FM yield gains of 17% (Cora), 39% (Citeseer), and 82% (Pubmed).

We add our approach for load-balancing (**LB**) during Aggregation, using the load distribution approach in Section V-C, on top of CP+FM. The combined effect (CP+FM+LB) is shown in Fig. 20(left) to reduce Aggregation time cumulatively by 47% (Cora), 69% (Citeseer), and 87% (Pubmed).

Optimizing Inference Time: We evaluate our techniques on GCN and GAT inference time. We first analyze the effect of CP on inference time. Next, we incrementally add FM and LR optimization to CP and measure their combined effect on inference time. Finally, we add all load-balancing (LB) methods: the LR technique for Weighting as well as load distribution during Aggregation. Figs. 20(middle) and (right) shows the reduction in the GCN and GAT inference time, respectively for CP, CP+FM, and CP+FM+LB. The reduction in inference time is higher for Pubmed (19717 vertices) than Cora (2708 vertices), indicating the scalability of GNNIE.

Customizing GNNIE for specific GNNs: GNNIE is specifically designed to support a wide variety of GNNs. The baseline architecture used for GNNs can be used without any change for GraphSAGE; for GINConv, a larger PE array can be used to overlap some additional computations (e.g., multiplication by $1+\epsilon$), but these PEs are not well utilized for other parts of the computation and the speedup is not worth the hardware cost. For GAT, where one could increase the number of SFUs to one per CPE to achieve 17.6% higher speedup, at the cost of 1.0% area increase and 21.1% higher power.

IX. Conclusion

We have proposed GNNIE, a versatile GNN acceleration platform for a wide degree of GNNs, including GATs. GNNIE efficiently works with unstructured data, input vertex feature vector sparsity, and adjacency matrix sparsity, and "power-law" vertex degree distribution. It mitigates load balancing issues, computational bottlenecks, and irregular/random data accesses using multiple methods: splitting the computation into blocks to leverage sparsity; optimized caching strategies; employing a flexible MAC architecture in the CPE array. Substantial improvements over prior work are shown.

References

[1] T. N. Kipf *et al.*, "Semi-Supervised Classification with Graph Convolutional Networks," in *International Conference on Learning Representations*, 2017.

[2] W. Hamilton *et al.*, "Inductive Representation Learning on Large Graphs," in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 1025–1035, 2017.

[3] P. Veličković *et al.*, "Graph Attention Networks," in *International Conference on Learning Representations*, 2018.

[4] K. Xu *et al.*, "How Powerful are Graph Neural Networks?," in *International Conference on Learning Representations*, 2019.

[5] J. Bruna *et al.*, "Spectral Networks and Locally Connected Networks on Graphs," in *International Conference on Learning Representations*, 2013.

[6] M. Defferrard *et al.*, "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering," in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 3844–3852, 2016.

[7] S. Han *et al.*, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pp. 243–254, 2016.

[8] Y.-H. Chen *et al.*, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[9] A. Aimar *et al.*, "Nullhop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 3, pp. 644–656, 2019.

[10] N. P. Jouppi *et al.*, "In-datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the International Symposium on Computer Architecture*, pp. 1–12, 2017.

[11] A. Parashar *et al.*, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pp. 27–40, 2017.

[12] H. Sharma *et al.*, "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pp. 764–775, 2018.

[13] N. P. Jouppi *et al.*, "A Domain-Specific Supercomputer for Training Deep Neural Networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

[14] T. J. Ham *et al.*, "Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pp. 1–13, 2016.

[15] G. Dai *et al.*, "FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 105–110, 2016.

[16] S.-W. Jun *et al.*, "GraFBoost: Using Accelerated Flash Storage for External Graph Analytics," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pp. 411–424, 2018.

[17] M. Yan *et al.*, "HyGCN: A GCN Accelerator with Hybrid Architecture," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pp. 15–29, 2020.

[18] T. Geng *et al.*, "AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, 2020.

[19] A. Auten *et al.*, "Hardware Acceleration of Graph Neural Networks," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 1–6, 2020.

[20] R. Ying *et al.*, "Hierarchical Graph Representation Learning with Differentiable Pooling," in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 4805–4815, 2018.

[21] J. Fowers *et al.*, "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp. 36–43, 2014.

[22] N. Srivastava *et al.*, "MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pp. 766–780, 2020.

[23] N. Srivastava *et al.*, "Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pp. 689–702, 2020.

[24] D. Salomon, *Data Compression: The Complete Reference*. London, UK: Springer Science & Business Media, 4th ed., 2007.

[25] J. Hruska, "HBM2 vs. GDDR6: New Video Compares, Contrasts Memory Types." https://www.extremetech.com/computing/289391-hbm2-vs-gddr6-new-video-compares-contrasts-memory-types, 4/11/2019.

[26] S. Ward-Foxton, "Memory Technologies Confront Edge AI's Diverse Challenges." https://www.eetimes.com/memory-technologies-confront-edge-ais-diverse-challenges, 9/18/2020.

[27] P. Nilsson *et al.*, "Hardware Implementation of the Exponential Function using Taylor Series," in *NORCHIP*, pp. 1–4, 2014.

[28] S. Liang *et al.*, "EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks," *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1511–1525, 2021.

[29] J. Malicevic *et al.*, "Everything You Always Wanted to Know about Multicore Graph Processing but Were Afraid to Ask," in *Proceedings of the USENIX Annual Technical Conference*, pp. 631–643, 2017.

[30] L. Nai *et al.*, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pp. 457–468, 2017.

[31] Z. Zhou *et al.*, "BlockGNN: Towards Efficient GNN Acceleration Using Block-Circulant Weight Matrices," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 1009–1014, 2021.

[32] J. Stevens *et al.*, "GNNerator: A Hardware/Software Framework for Accelerating Graph Neural Networks," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 955–960, 2021.

[33] C. Chen *et al.*, "DyGNN: Algorithm and Architecture Support of Dynamic Pruning for Graph Neural Networks," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 1201–1206, 2021.

[34] J. Li *et al.*, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, pp. 775–788, 2021.

[35] J. Li *et al.*, "SGCNAX: A scalable graph convolutional neural network accelerator with workload balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2834–2845, 2022.

[36] S. Ghodrati *et al.*, "Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pp. 681–697, 2020.

[37] U. Gupta *et al.*, "RecPipe: Co-designing Models and Hardware to Jointly Optimize Recommendation Quality and Performance," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pp. 870–884, 2021.

[38] Y. Zhang *et al.*, "Making Caches Work for Graph Analytics," in *Proceedings of the IEEE International Conference on Big Data*, pp. 293–302, 2017.

[39] X. Chen *et al.*, "Rubik: A Hierarchical Architecture for Efficient Graph Neural Network Training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 4, pp. 936–949, 2022.

[40] J. Arai *et al.*, "Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pp. 22–31, 2016.

[41] Y. Wang *et al.*, "GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 515–531, 2021.

[42] P. Faldu *et al.*, "Domain-Specialized Cache Management for Graph Analytics," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pp. 234–248, 2020.

[43] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2015.

[44] M. O'Connor *et al.*, "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pp. 41–54, 2017.

[45] P. Sen *et al.*, "Collective classification in network data," *AI magazine*, vol. 29, no. 3, pp. 93–93, 2008.

[46] V. P. Dwivedi *et al.*, "Benchmarking Graph Neural Networks," *arXiv preprint arXiv:2003.00982*, 2020.

[47] X. Zhu *et al.*, "GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning ," in *Proceedings of the USENIX Annual Technical Conference*, pp. 375–386, 2015.

[48] J. Mohoney *et al.*, "Marius: Learning Massive Graph Embeddings on a Single Machine," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 533–549, 2021.