GNNIE: GNN Inference Engine with Load-balancing and Graph-specific Caching

Sudipta Mondal, Susmita Dey Manasi, Kishor Kunal, Ramprasath S and Sachin S. Sapatnekar
University of Minnesota, Minneapolis, MN 55455, USA

ABSTRACT

Graph neural networks (GNN) inferencing involves *weighting* vertex feature vectors, followed by *aggregating* weighted vectors over a vertex neighborhood. High and variable sparsity in the input vertex feature vectors, and high sparsity and power-law degree distributions in the adjacency matrix, can lead to (a) unbalanced loads and (b) inefficient random memory accesses. GNNIE ensures load-balancing by splitting features into blocks, proposing a flexible MAC architecture, and employing load (re)distribution. GNNIE's novel caching scheme bypasses the high costs of random DRAM accesses. GNNIE shows high speedups over CPUs/GPUs; it is faster and runs a broader range of GNNs than existing accelerators.

KEYWORDS

GNN, hardware accelerator, graph-specific caching, load balancing

ACM Reference Format:
Sudipta Mondal, Susmita Dey Manasi, Kishor Kunal, Ramprasath S and Sachin S. Sapatnekar . 2022. GNNIE: GNN Inference Engine with Load-balancing and Graph-specific Caching. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530503

1 INTRODUCTION

Deep learning accelerators have largely focused on data with Euclidean embeddings, e.g., audio/video/images/speech. Many real-world problems use graphs to model relationships. Inferencing on large, unstructured, and sparse graphs requires specialized graph neural networks (GNNs) [1–6]. GNNs perform two operations per layer: ***(a) Weighting*** multiplies vertex feature vectors by a weight matrix. ***(b) Aggregation*** consolidates information from neighbors to compute vertex feature vectors for the next layer.

GNN accelerator efficiency is affected by data sparsity. During Weighting, the input node feature vectors can have *high and variable levels of sparsity*, leading to unbalanced computations. In Aggregation, computations over vertex neighborhoods are defined by the graph adjacency matrix, which has *high sparsity* (> 99.8% for datasets in this paper, vs. 10%–50% sparsity for DNN data). Vertex degrees show a *power-law distribution*: most vertices have very low degree, but a small number of vertices have extremely high degree (in the Reddit dataset, 11% of vertices cover 88% of all edges). This leads to random memory access patterns and poor data locality.

This work was supported in part by the Semiconductor Research Corporation (SRC).

Key desirable attributes of a GNN accelerator are the ability to: (1) handle a diverse set of GNNs, from high-accuracy GATs to faster but less accurate GNNs (GCN, GraphSAGE). (2) efficiently access data from DRAM, hiding overheads of random access patterns. (3) load-balance tasks during both Weighting (due to high input node feature vector sparsity) and Aggregation (due to high adjacency matrix sparsity and power-law distribution).

Weighting performs matrix-vector multiplication similar to convolutional neural network (CNN) computations, but CNN accelerators [7–10] are inefficient for graph data. Aggregation operates on graph neighborhoods and resembles graph analytics, but graph processing accelerators [11–13] are designed to perform lightweight computations, significantly lower than the needs of a GNN. Extensions of CNN/graph processing engines are inadequate.

BlockGNN [14] optimizes Weighting by applying an FFT and block-circulant constraint on the weight matrix. HyGCN [15], GNNerator [16], and DyGNN [17] pipeline separate engines for Aggregation and Weighting; DyGNN also employs pruning to reduce vertex/edge redundancy. Such architectures are 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. HyGCN performs Aggregation before Combination, but the opposite order is cheaper [18, 19]. To reduce random memory access, these approaches use sharding. In HyGCN, this has limited parallelism as the sliding window of the current shard depends on the shrinking of the previous shard. None of these methods accounts for power-law behavior. AWB-GCN [18], limited only to GCNs, views the computations as sparse-dense matrix multiplications, but their graph-agnostic view of Aggregation leads to numerous costly off-chip accesses due to cache misses.

In prior work, (1) load balancing schemes incur high communication/control overheads [18, 19]; (2) GAT computations are unoptimized (GNNIE reorders them for efficiency); (3) graph structure is not exploited to manage adjacency matrix sparsity effects.

The GNNIE inference engine handles sparsity in both the input feature vectors and adjacency matrix. Our contributions are:

- Hardware efficiency, using a *single engine* for regular-structured Weighting and irregular-structured Aggregation computations.
- *Versatility* in efficiently executing of a range of GNNs, from lower accuracy/lower computation (GCN, GraphSAGE) to higher accuracy/higher computation (GATs).
- *Load-balanced Weighting* for high PE utilization by (i) splitting vertex features into blocks, (ii) reordering computations on a flexible MAC (FM) architecture, and (iii) static load redistribution.
- *Load-balanced Aggregation* through a mapping scheme to maximize PE utilization. A novel decomposition for GAT computation results in linear-complexity attention vector multiplication.
- *Lightweight dynamic caching*, with random memory accesses only to on-chip cache, avoiding costly random DRAM accesses.

2 BACKGROUND

In layer l of a GNN (layer 0 is the input), the *vertex feature vector* for vertex i is an F^l-dimensional row vector, $\mathbf{h}_i^l$. Over neighboring vertices j, vectors $\mathbf{h}_j^{l-1}$ of layer $l-1$ are aggregated to create $\mathbf{h}_i^l$. Table 1 lists the operations for various GNNs. Specifically:

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 over $\mathcal{N}_i$ for vertex i. For GCN/GATs/GINConv, $\mathcal{N}(i) = \{i\} \cup N(i)$, where $N(i)$ is the one-hop neighborhood of 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 [2]: 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 [3]: The products $\mathbf{h}_j^{l-1} W^l$ are combined over $j \in \mathcal{N}(i)$ using aggregator a_k (typically, mean, max, or pooling).

GATs [4]: 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 $\sum_{j \in \{i\} \cup \mathcal{N}(i)} \alpha_{ij} \mathbf{h}_j^{l-1} W^l$, a weighted aggregation.

GINConv [5]: Feature vectors in $\mathcal{N}_i$ are summed and added to ϵ^l times the 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.

DiffPool [20] can be combined with any GNN to reduce data volume. It uses two GNNs: one extracts vertex embeddings for classification, and one extracts embeddings for hierarchical pooling.

GCN computation can be written as $\mathbf{h}_i^l = \sigma(\widetilde{A} \mathbf{h}_i^{l-1} W^l)$, where $\widetilde{A} = D^{-1/2}(A + I) D^{-1/2}$ 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 $\widetilde{A} \times (\mathbf{h}_i^{l-1} \times W^l)$. We use the latter, which requires an order of magnitude fewer computations [18, 19].

3 ACCELERATOR ARCHITECTURE

The block diagram of the proposed accelerator is illustrated in Fig. 1, 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 is stored in sparse compressed sparse row (CSR) format and run-length compression (RLC) is used for encoding sparse input feature vectors.

(2) *Memory interface:* The **input buffer** stores vertex features for the current layer l, i.e., $\mathbf{h}_i^{l-1}$, 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. The **output buffer** caches intermediate results, including the result of multiplication by W^l after Weighting, and the result after Aggregation. The

Table 1: 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	$\mathbf{h}_i^l = \sigma\left(\frac{\sum_{j \in \{i\} \cup N(i)} \exp(e_{ij}) \mathbf{h}_j^{l-1} W^l}{\sum_{j \in \{i\} \cup N(i)} \exp(e_{ij})}\right)$; $e_{ij} = \text{LeakyReLU}(\mathbf{a}^T \cdot [\mathbf{h}_i^{l-1} W^l \Vert \mathbf{h}_j^{l-1} W^l])$
GINConv	$\mathbf{h}_i^l = \text{MLP}^l \left((1 + \epsilon^l) \mathbf{h}_i^{l-1} + \sum_{j \in \mathcal{N}(i)} \mathbf{h}_j^{l-1}, W^l, \mathbf{b}^l\right)$

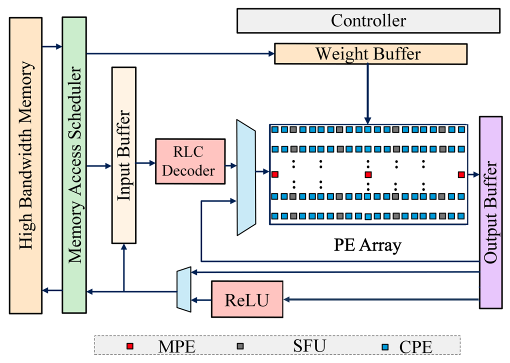


Fig. 1. Block diagram of the proposed architecture.

weight buffer holds the values of W^l during Weighting, and, for GAT computations, the attention vector during Aggregation. The **memory access scheduler** coordinates off-chip memory requests.

(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)** (exp, LeakyReLU, division) [grey blocks], and a row of merge PEs (MPEs) [one for each CPE column; shown in red blocks in (Fig. 1)]. Interleaved placement allows low latency and communication overhead with CPEs. For exponentiation, we use an accurate, low-area lookup-table-based implementation. **Merge PEs (MPEs)** aggregate partial results of vertex features sent from its designated CPE column during Weighting. A bank of partial sum (psum) spads holds intermediate results from the MPE until they are sent to the output buffer.

(4) The *controller* coordinates operations: assigning vertex features to CPEs; workload reordering among CPEs; sending CPE results to MPEs; sending MPE data to the output buffer; writing to DRAM.

4 MAPPING WEIGHTING TO CPES

4.1 Scheduling Operations in the CPEs

The Weighting step multiplies the *sparse* feature row vector $\mathbf{h}_i^{l-1}$ with the *dense* weight matrix W^l. 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. Our novel scheduling methodology keeps the CPEs busy during the computation, so that Weighting is not memory-bounded. We partition data in two ways:

(1) **Across the vertex feature vector:** At a time, we multiply a **block** of k elements of $\mathbf{h}_i^{l-1}$ with N columns of W^l (Fig. 2). To process the entire feature vector in the CPE array, $k = \lceil F^{l-1}/M \rceil$.

(2) **Across vertices:** A **set** of s feature vectors is processed in the CPE array (Fig. 2), where s is constrained by the input buffer size. Our weight-stationary scheme processes one feature vector of a set at a time (Fig. 3), until all $\lceil |V|/s \rceil$ sets are processed.

In this scheme, we fetch N columns of the weight matrix, W^l, from the DRAM to the weight buffer and load the CPEs as:

- 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 each feature vector in a set, each ith subvector, of size k, is broadcast to the entire CPE row i. This is indicated by $\mathbf{h}_{(ik:(i+1)k-1)}$ in Fig. 3.

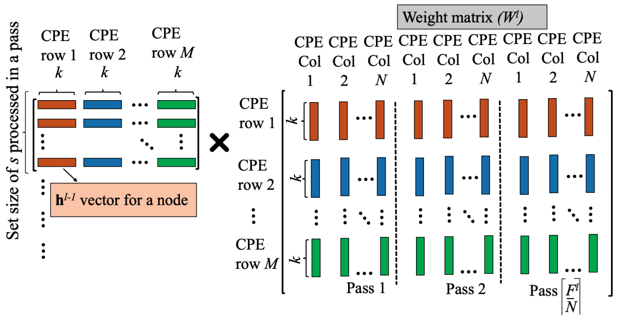


Fig. 2. Mapping Weighting operations to the CPE array.

After loading CPEs the multiplication between a feature vector and the N columns of W^l is carried out. If a block of feature vector is all-zeros, the CPE skips this computation. The sparsity of input vectors allows skipping, but if the entire vector is processed at a time (as in prior work), it is nonzero and skipping is not possible.

A **pass** (Fig. 2) is completed when all vertex feature vectors across all sets have finished multiplication with the N columns of W^l. Thus $\mathbf{h}_i^{l-1}W^l$ for a pass is carried out as follows:

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

At the end of a pass, the next set of N columns of W^l is loaded. Double-buffering is used so that fetching the next blocks of weights overlaps computation. To leverage input data sparsity, a zero detection buffer is used to skip CPE computations involving zeros.

The partial results for an element of the transformed feature vectors are sent from CPEs in each column to its designated MPE for accumulation, along with a tag which denotes their vertex id. As stated earlier, psum spads hold the partially accumulated results. The final result, i.e., one element of $\mathbf{h}_j^{l-1}W^l$, is written to the output with its vertex id, and eventually written back to DRAM.

4.2 Load Balancing for Weighting

High and variable input feature vector sparsity results in nonuniform distribution of non-zeros among k-subvector blocks: sparse blocks compute rapidly ("rabbits") while dense blocks take longer ("turtles"), causing workload imbalance. Since MPEs have a limited number of psum slots, this imbalance requires stalls, contributing to inefficiency. We combat this by proposing two mechanisms:

(1) Flexible MAC (FM) Architecture: To avoid stalls and speed up computation the number of MACs per CPE can be increased uniformly; this helps "turtles" but is overkill for "rabbits." We divide

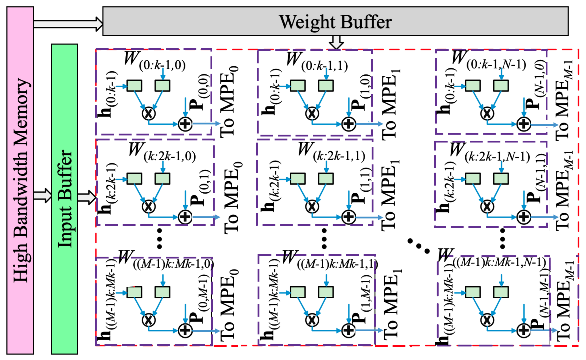


Fig. 3. Weight-stationary linear transform of vertex features.

the CPE array into g row groups, each with an equal number of rows. The number of MACs per CPE, $|MAC|_i$, is monotonically nondecreasing along the rows: $|MAC|_1 \leq |MAC|_2 \leq \cdots \leq |MAC|_g$.

The input buffer statically assigns vertex feature blocks to CPE rows based on their nonzero count. In a linear-time preprocessing step, the k-element blocks are binned based on the number of nonzeros, where the number of bins equals the number of CPE groups. The bins map to CPE groups: blocks with the fewest nonzeros are statically scheduled to go to the first (low-MAC) rows, those with the next most nonzeros go to the next CPE row group, and so on.

(2) Load Redistribution (LR): The static FM-based workload distribution does not achieve full balance. We pair CPE rows to redistribute loads, 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 are loaded with weights for the offloaded workloads.

5 AGGREGATION COMPUTATIONS

For most GNNs, Aggregation sums over the neighbors of a vertex, but GATs must also compute attention coefficients. We first address this and then consider Aggregation operations for all GNNs.

5.1 Optimizing GAT Computations

Reordering GAT Computations: Our method reorders GAT computations for efficiency. The first step in finding attention coefficient α_{ij} for vertices i, j is to multiply the $2F^l$-dimensional attention vector, $\mathbf{a}^l = [\mathbf{a}_1^l \ \mathbf{a}_2^l]$, by $(\boldsymbol{\eta}_{\mathbf{w}_i}{}^l, \boldsymbol{\eta}_{\mathbf{w}_j}{}^l)$, two concatenated F^l-dimensional weighted vertex feature vectors (here, $\boldsymbol{\eta}_{\mathbf{w}}{}^l = \mathbf{h}^{l-1}W^l$). Then

$$e_{ij} = \mathbf{a}_1^{l\,T} \cdot \boldsymbol{\eta}_{\mathbf{w}_i}{}^l + \mathbf{a}_2^{l\,T} \cdot \boldsymbol{\eta}_{\mathbf{w}_j}{}^l = e_{i,1} + e_{j,2} \tag{3}$$

where $e_{i,1} = \mathbf{a}_1^{l\,T} \cdot \boldsymbol{\eta}_{\mathbf{w}_i}{}^l$, $e_{j,2} = \mathbf{a}_2^{l\,T} \cdot \boldsymbol{\eta}_{\mathbf{w}_j}{}^l$. The normalized attention coefficient takes this through a LeakyReLU and softmax over $N(i)$

$$\alpha_{ij} \quad = \quad \text{softmax}\left(\text{LeakyReLU}(e_{ij})\right) \tag{4}$$

A naïve method would fetch $\boldsymbol{\eta}_{\mathbf{w}_j}{}^l$ from each neighbor j of i, compute e_{ij} using (3), 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. Reordering to avoid redundancy, for each vertex i, we compute
(a) $e_{i,1} = \mathbf{a}_1^{l\,T}\boldsymbol{\eta}_{\mathbf{w}_i}{}^l$, used to compute α_{i*} at vertex i.
(b) $e_{i,2} = \mathbf{a}_2^{l\,T}\boldsymbol{\eta}_{\mathbf{w}_i}{}^l$, used to compute α_{j*} at vertex $j \in N(i)$.
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 $j \in N(i)$.

For $|V|$ vertices and $|E|$ edges, the naïve computation performs $O(|E|)$ multiplications per vertex, for a cost of $O(|V||E|)$. Our reordering has $O(|E|)$ operations over all vertices for $O(|V|+|E|)$ computation, i.e., *latency/power are linear in graph size.*

Mapping Attention Vector Multiplication: As in Weighting, we use a block strategy to distribute computation in the CPE array. Vector $\boldsymbol{\eta}_{\mathbf{w}_i}$ 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, where 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., $V_a = |OB|/N$.

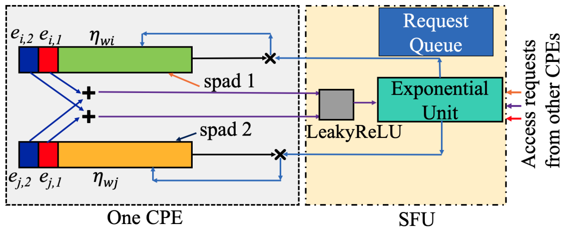

$e_{i,2}$ $e_{i,1}$ η_{wi}
spad 1
spad 2
$e_{j,2}$ $e_{j,1}$ η_{wj}
Request Queue
Exponential Unit
LeakyReLU
Access requests from other CPEs
One CPE
SFU

Fig. 4. Data flow corresponding to computation for an edge.

This dot product computation is similar to the weight-stationary scheme in Weighting. Attention vectors remain stationary until a pass through all vertices. Unlike Weighting, $\boldsymbol{\eta_w}^l_j$ and $\mathbf{a}^l$ are dense, and CPE load balancing is unnecessary. After all V_a vertices in the row are processed, the spad that contains $\mathbf{a}^l_1$ is loaded with $\mathbf{a}^l_2$, 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 appended to the feature vector of vertex i and written back to the output buffer.

5.2 Mapping Edge-based Computations

The last step requires aggregation from each neighbor of a vertex. We process edges in parallel in the CPE array. All GNNs perform edge-based summations followed by an activation function; for GATs, the weights for summation are computed as described above.
Load Distribution: During Aggregation, for the subgraph of vertices in the input buffer, edge data is accumulated by pairwise assignment to CPE spads. Due to power-law behavior, vertex degrees in the subgraph may have a large range. To distribute the load, aggregation summations over all vertices are divided into pairwise summations assigned to CPEs, with results accumulated in MPEs.
GATs: The final step in computing the attention coefficient α_{ij} involves edge-based computations (Equation (4)):

- the addition, $e_{ij} = e_{i,1} + e_{j,2}$ (Equation (3)
- 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 between vertices i, j contributes one e_{ij} to the numerator and denominator of softmax. These computations are parallelized in the CPEs over all vertex neighbors using pull-based aggregation.

The computation of numerator in the softmax step is shown in Fig. 4. 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 $\boldsymbol{\eta_w}^l_j$. A similar operation is performed for vertex j.
Other GNNs: For GCN, GraphSAGE, and GINConv, Aggregation sums weighted vertex feature vectors over all neighbors (or a sample of neighbors for GraphSAGE) of each vertex. This is similar to but simpler than the computation in Fig. 4: just addition is performed. 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 softmax denominator. Finally, the accumulation over neighbors is divided by the denominator, in the SFU to obtain the result. When all components of the sum for vertex i are accumulated, the result is sent through activation in the SFU and written to DRAM.

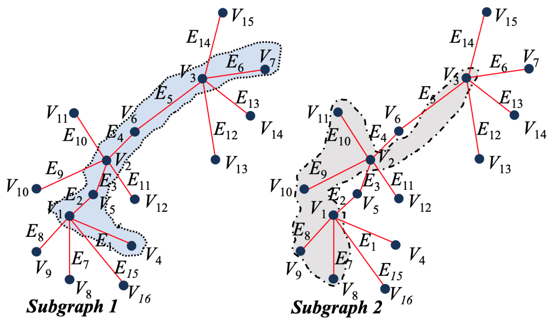


Fig. 5. Example illustrating the subgraph in the input buffer (left) and its evolution after cache replacement (right).

6 GRAPH-SPECIFIC CACHING

Aggregation intensively accesses the adjacency matrix. Computational efficiency requires graph-specific caching to maximize the reuse of the cached data in the input buffer and minimize off-chip random memory accesses. We propose a novel frequency-based caching policy that tracks the number of unprocessed neighbors of each vertex. The scheme ensures that *all random-access patterns are confined to on-chip buffers* and *off-chip fetches are sequential.*

Frequency-based caching for graphs has been used in Cagra [21], a software framework, using a programming interface. In contrast, GNNIE uses a hardware-centric *dynamic* scheme that tracks frequencies with minimal hardware overhead. Another cache management scheme for graph analytics, GRASP [22], employs a most-recently-used (MRU) approach, but this is based on past history, while GNNIE's approach measures future potential for a vertex.
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 (Section 5.2) 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.

Consider the example graph, with vertices V_1 through V_{16}, in Fig. 5. The highest degree vertices are first cached into the input buffer: vertices V_1, V_2, and V_3 of degree 5, vertices V_5 and V_6 of degree 2, and V_4 and V_7 with degree 1. 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: Vertices are replaced after computation of each subgraph using a replacement policy that prioritizes retention of vertices with the most *unprocessed* edges (denoted by α_i for vertex i). Such vertices appear more frequently in the list of neighbors for other vertices, this increases the likelihood of finding both the source and destination of edges in the cache.

The policy requires inexpensive preprocessing that bins vertices in order of their degrees, differentiating high-degree vertices from medium-/low-degree vertices to prioritize higher-degree vertices. After preprocessing, vertices of the graph are stored contiguously in DRAM in descending degree order of the bins. Ties are broken

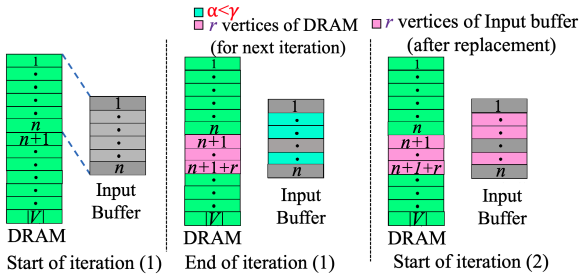


Fig. 6. Input buffer replacement policy during Aggregation.

in dictionary order of vertex IDs. This preprocessing step and the replacement policy enable GNNIE to avoid random DRAM accesses.

Fig. 6 illustrates our policy, managed by a cache controller using a 4-way set associative cache. Vertices are stored contiguously in DRAM in descending degree order. If the input buffer capacity is n vertices, data for vertices 1 to n are initially 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, i.e., after finishing computation of the subgraph corresponding to 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 if fewer (or more) than r candidates are available. 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 vertex data with α_i value into DRAM. When all vertices are processed once, we have completed a **Round**. This continues until for all vertices $\alpha_i = 0$.

Similarly, the partial sums for the vertex feature vector in the output buffer are updated after each iteration and eventually written back to DRAM if all accumulations are complete for any $\mathbf{h}_i^l$. Due to limited output buffer capacity, we use degree-based criterion for prioritizing writes to the output buffer vs. DRAM.

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. While performing aggregation for each vertex of the current subgraph the feature data of its neighbors are fetched from the cache into the CPE array. Consequently, a vertex feature maybe fetched multiple times. Though this process may incur random accesses, they are limited to the cache, which has far better random-access bandwidth than DRAM.

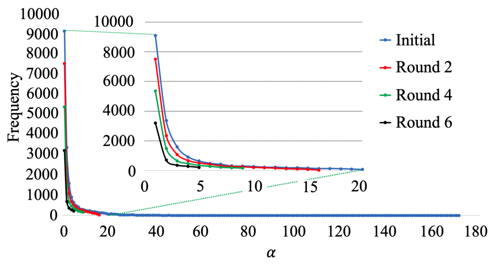


Fig. 7. Histogram of α through various Rounds (Pubmed). The inset shows a magnified view.

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. Tracking of unprocessed edges in a cache block is similar to tracking α_i.

The effectiveness of the approach is illustrated in Fig. 7, 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.

7 EVALUATION

Experimental Setup: We develop a simulator to measure the number of GNNIE execution cycles. We use Ramulator [23] to model off-chip access to HBM (256 GB/s, 3.97pJ/bit [24]), and CACTI 6.5 for the area, energy consumption, and access latency of on-chip buffers. A Verilog description of GNNIE is synthesized using Synopsys Design Compiler with a 32nm standard VT cell library, resulting in a chip area of 15.6mm^2 and frequency of 1.3GHz.

GNNIE is evaluated on the graph datasets in Table 2, whose size is representative of edge applications, using five GNNs (GAT, GCN, GraphSAGE, GINConv, DiffPool). All GNNs have one hidden layer, except GINConv which has two; each hidden layer has 128 channels. The GAT hidden layer uses eight 16-dimensional attention heads.

GNNIE Configuration: We configure GNNIE as follows:
Buffers: Output buffer: 1MB; Weight buffer: 128KB; Input buffer: 256KB for smaller datasets (CR, CS), 512KB for larger datasets (PB, RD); reported area/power correspond to the larger input buffer.
CPE array with flexible MACs: 16 × 16 array; 4 MACs (rows 1–8), 5 MACs (rows 9–12), 6 MACs (rows 13–16). The number of MACs per CPE was chosen through design space exploration.

Performance comparisons with CPU and GPU: We compare all GNNs against PyTorch Geometric (PyG) [25], implemented as:
(a) *PyG-CPU*: Intel Xeon Gold 6132@2.60GHz CPU, 768GB DDR4.
(b) *PyG-GPU*: V100 Tesla GPU, V100S-PCI@1.25GHz, 32GB HBM2.

As shown in Fig. 8, the average speedup of GNNIE over the PyG-CPU and PyG-GPU are 7202.80× and 17.76×, respectively. *Speedup calculations include the total preprocessing times, which are inexpensive.* 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, distributing the computation across multiple batches enables weight reuse, increasing efficiency. The speedup of GNNIE on GINConv is further enhanced because of PyTorch Geometric executes Aggregation before Weighting: as described in Section 2, this requires more computation than the reverse order of computation used in GNNIE.

For 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

Table 2: Dataset information

Dataset	Vertices	Edges	Feature Length	Labels	Feature Vector 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%

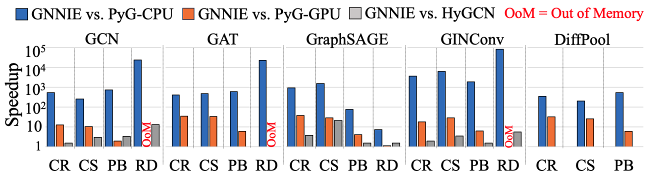


Fig. 8. Performance: GNNIE vs. PyG-CPU, PyG-GPU, HyGCN

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 [4, 26].

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 ~10× 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.

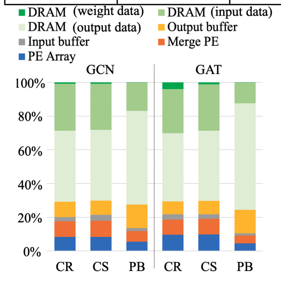
HyGCN comparison: The HyGCN configuration, like ours, uses 128 channels for the hidden layers of all GNN models. Since HyGCN reports results using a baseline CPU with similar performance to our baseline CPU, we take the ratio of the relative speedup compared to the baseline CPU to compute speedup over HyGCN. Fig. 8 shows an average GNNIE speedup of 5.07× over HyGCN, even though on-chip HyGCN buffers (>24MB) are much larger than GNNIE (<1.7MB). The improvements are attributable to differences between GNNIE and HyGCN described in Section 1. As HyGCN does not implement GAT and does not report results on the widely used datasets that we evaluate for DiffPool, GAT/DiffPool comparisons cannot be shown.

Throughput/Energy Comparisons: Table 3 shows the GNNIE throughput for 3 datasets: it degrades only slightly with graph size.

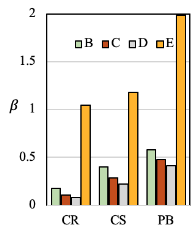
The power dissipation of GNNIE is 3.9W in 32nm, lower than HyGCN (6.7W in 12nm). Fig. 9(a) shows the energy breakdown for GNNIE for GAT and GCN across three datasets, including DRAM energy required to transfer data to the output, input, and weight buffers. On-chip weight buffer energy is negligible and not shown.

Table 3: Throughput for various datasets for GNNIE.

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



(a)



(b)

Fig. 9. (a) Energy breakdown for GCN and GAT. (b) Cost/benefit ratio for adding MACs in Designs B–E.

Optimization Analysis: We study key GNNIE optimizations, relative to a baseline **Design A** (4 MACs/CPE, 1024 total MAC units).

Cost/benefit analysis for FM We consider: **Design B** (5 MACs/CPE, 1280 total MAC units), **Design C** (6 MACs/CPE, 1536 total MACs), **Design D** (7 MACs/CPE, 1792 total MACs), and the GNNIE FM architecture, **Design E** (4/5/6 MACs/CPE as described earlier, 1216 total MACs). The cost/benefit ratio, β, relative to Design A is:

$$\beta = (\text{\% reduction in Cycles})/(\text{\% increase in MACs}) \tag{5}$$

Fig. 9(b) shows that β drops from Design B–D which add MAC units uniformly to all CPEs, because the added MACs are underutilized. By selectively adding MACs, Design E with FM achieves high β.

GNNIE Optimizations for Weighting We analyze the improvement from flexible MACs (**FM**) over Design A for PB. Fig. 10 shows the imbalanced workload distribution among the CPE rows for the baseline design, caused by variations in vertex feature sparsity. This is mitigated by the FM design, which reduces the number of Weighting cycles by 24.0%. Next, we add the load redistribution (**LR**), moving computations from the heavily-loaded CPE rows to lightly-loaded rows: this brings the reduction to 28.3%.

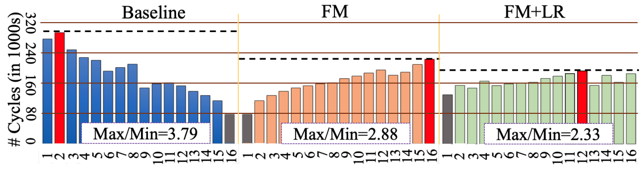


Fig. 10. CPE row workload in Weighting for Pubmed.

8 CONCLUSION

GNNIE's edge inference accelerator covers a variety of GNNs, including GATs. Its novel methods mitigate random memory access and load imbalance: k-blocking during Weighting, FM/LR schemes, computation reordering (for GATs), and graph-specific caching.

REFERENCES

[1] S. Han et al. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*, June 2016.
[2] T. Kipf et al. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*, 2017.
[3] W. Hamilton et al. Inductive Representation Learning on Large Graphs. In *NeurIPS*, 2017.
[4] P. Veličković et al. Graph Attention Networks. In *ICLR*, 2018.
[5] K. Xu et al. How Powerful are Graph Neural Networks? In *ICLR*, 2019.
[6] H. Sharma et al. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *ISCA*, 2018.
[7] H. Genc et al. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-stack Integration. In *DAC*, 2021.
[8] Y. Chen et al. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *JSSC*, 52(1), 2017.
[9] N. P. Jouppi et al. In-datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, June 2017.
[10] A. Parashar et al. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proc. ISCA*, pages 27–40, 2017.
[11] Tae Jun Ham et al. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *MICRO*, 2016.
[12] G. Dai et al. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *FPGA*, 2016.
[13] Sang-Woo Jun et al. GraFBoost: Using accelerated flash storage for external graph analytics. In *ISCA*, 2018.
[14] Z. Zhou et al. BlockGNN: Towards Efficient GNN Acceleration Using Block-Circulant Weight Matrices. In *DAC*, 2021.
[15] M. Yan et al. HyGCN: A GCN Accelerator with Hybrid Architecture. In *HPCA*, 2020.
[16] J.R. Stevens et al. GNNerator: A Hardware/Software Framework for Accelerating Graph Neural Networks. In *DAC*, 2021.
[17] C. Chen et al. DyGNN: Algorithm and Architecture Support of Dynamic Pruning for Graph Neural Networks. In *DAC*, 2021.
[18] T. Geng et al. AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. In *MICRO*, 2020.
[19] S. Liang et al. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. *IEEE Transactions on Computers*, 2020.
[20] R. Ying et al. Hierarchical Graph Representation Learning with Differentiable Pooling. In *NeurIPS*, 2018.
[21] Y. Zhang et al. Making Caches Work for Graph Analytics. In *IEEE BigData*, 2017.
[22] P. Faldu et al. Domain-Specialized Cache Management for Graph Analytics. In *HPCA*, 2020.
[23] Y. Kim et al. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Comp. Arch. Letters*, 15(1), 2015.
[24] M. O'Connor et al. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *ISCA*, 2017.
[25] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. https://github.com/pyg-team/pytorch_geometric.
[26] V. P. Dwivedi et al. Benchmarking Graph Neural Networks. *arXiv preprint arXiv:2003.00982*, 2020.