2023 Design, Automation & Test in Europe Conference (DATE 2023) – Best Paper Award Candidate

# PEDAL: A **P**ower **E**fficient GCN Accelerator with Multiple **DA**taf**L**ows

Yuhan Chen, Alireza Khadem, Xin He, Nishil Talati, Tanvir Ahmed Khan, and Trevor Mudge
*Computer Science and Engineering, University of Michigan, Ann Arbor, MI, USA*
{chenyh, arkhadem, xinhe, talatin, takh, tnm}@umich.edu

***Abstract*—Graphs are ubiquitous in many application domains due to their ability to describe structural relations. Graph Convolutional Networks (GCNs) have emerged in recent years and are rapidly being adopted due to their capability to perform Machine Learning (ML) tasks on graph-structured data. GCN exhibits irregular memory accesses due to the lack of locality when accessing graph-structured data. This makes it hard for general-purpose architectures like CPUs and GPUs to fully utilize their computing resources. In this paper, we propose PEDAL, a power-efficient accelerator for GCN inference supporting *multiple dataflows*. PEDAL chooses the best-fit dataflow and phase ordering based on input graph characteristics and GCN algorithm, achieving both efficiency and flexibility. To achieve both high power efficiency and performance, PEDAL features a light-weight processing element design. PEDAL achieves 144.5×, 9.4×, and 2.6× speedup compared to CPU, GPU, and HyGCN, respectively, and 8856×, 1606×, 8.4×, and 1.8× better power efficiency compared to CPU, GPU, HyGCN, and EnGN, respectively.**

***Index Terms*—Accelerator, power-efficient, multiple dataflows**

## I. Introduction

With the rapid development of deep learning in the last decade, neural networks are now widely adopted in many applications such as image recognition [16], object detection [20], and machine translation [2]. However, traditional neural networks are limited to handling Euclidean data [10] such as one-dimensional (1D) text streams and two-dimensional (2D) images, and do not generalize well on non-Euclidean data such as graphs [21] and manifolds [4]. Graph Neural Networks (GNNs) take a further step to explore graph-structured data. Compared to Euclidean data, graphs have better expressiveness, so GNNs can learn from the latent information of nodes and the connections between nodes. This extends the application scope of deep learning to a wider range of applications [22, 25] such as natural science [3].

Among different types of GNNs, Graph Convolutional Networks (GCN) is one of the most prominent algorithm [15]. Motivated by Convolutional Neural Networks (CNNs), GCNs generalize convolution to graph-structured data. GCN solves CNN's limitation of only applicable to regular Euclidean data [25]. GCN is composed of two phases - aggregation and combination. Aggregation collects information from neighboring nodes and/or edges. It works on the input graph and often suffers from irregular memory accesses. Combination uses multi-layer perceptron (MLP) to further process the aggregated results by multiplying them with the trained weight matrices, which have regular memory accesses. GCN has many variations [6, 11, 26], and has developed into a big algorithm family.

Due to the inherent irregular memory accesses in GCNs, CPUs and GPUs are not able to make good use of their massive computing resources. Thus, several works are proposed to enhance resource utilization. HyGCN [23] uses dedicated processing engines for the aggregation and combination phases to alleviate the memory irregularity in the aggregation phase while exploiting the regularity in the combination phase. EnGN [17] applies edge reorganization to compress the sparse adjacency matrix and uses degree-aware vertex cache to store hot nodes. AWB-GCN [8] observes that real-world graph datasets have power-law distributions, and it optimizes Processing Elements' (PE) utilization by performing workload balancing among PEs. ReGNN [5] dynamically computes and reuses the aggregated features of redundant neighbor sets to reduce memory accesses. GCoD [24] and I-GCN [9] both try to improve graph regularity by rearranging the adjacency matrix permutation.

Although prior works performed various optimizations to enhance resource utilization, we observe that they use a fixed dataflow and are not flexible enough to efficiently run different GCNs. Firstly, real-world datasets span a wide range of sizes and densities. They require different aggregation dataflows based on the input dataset characteristics. Secondly, the order of aggregation and combination phases can be altered when the aggregation function is linear (see §II). While this results in better performance, the order must be respected with non-linear aggregation functions. Thus, it is important to support different dataflows and orderings to achieve both efficiency and flexibility.

In this paper, we make the following contributions:

- We perform quantitative and qualitative analysis on three widely used GCN algorithms: vanilla GCN, GS-mean, and GS-max with 5 real-world datasets. We show that the GCN algorithms and input dataset characteristics affect the choice of phase ordering and dataflow for the best performance.
- We propose PEDAL, an accelerator for GCN inference. PEDAL features three dataflows, and supports both orderings of the aggregation and combination phases, achieving both efficiency and flexibility.
- We train a decision tree with 400 synthetic datasets to automatically and accurately choose the best dataflow and phase ordering for a GCN algorithm.
- We evaluate the performance of PEDAL using a cycle-accurate simulator and measure its power and area using RTL synthesis. We show PEDAL achieves 144.5×, 9.36×, and 2.55× speedup compared to CPU, GPU, and HyGCN, and also 8856×, 1606×, 8.4× and 1.78× better power efficiency compared to CPU, GPU, HyGCN, and EnGN.

To the best of our knowledge, this is the **first work** that

978-3-9819263-7-8/DATE23/© 2023 EDAA

| Category | Notation & Acronyms | Note |
|---|---|---|
| Dimension | N | Number of nodes in the graph |
| | F1 | The feature dimension for the 1st layer |
| | F2 | The feature dimension for the 2nd layer |
| Matrix | A | Adjacency matrix, dimension N x N |
| | X | Feature matrix, dimension N x F1, each row is a Feature vector (transposed) for the corresponding node. |
| | W | Weight matrix, dimension F1 x F2 |
| Acronyms | **AC** | Short for Aggregation+Combination order |
| | **CA** | short for Combination+Aggregation order |
| | **IP-AC** | Short for Inner-Product, AC order dataflow |
| | **IP-CA** | Short for Inner-Product, CA order dataflow |
| | **RW-AC** | Short for Row-Wise, AC order dataflow |
| | **RW-CA** | Short for Row-Wise, CA order dataflow |
| Others | $\mathcal{N}(v)$ | Neighbors of node v |
| | $d(M)$ | density of matrix M |
| | $NNZ(M)$ | number of non-zeroes of matrix M |

TABLE I: Notation and acronyms used in this paper

explores different dataflows and execution orders for the GCN workload and exploits this knowledge to choose the best dataflow according to the input graph and GCN algorithm.

## II. Background

GCN uses a convolutional layer to collect information for training and inference. While CNN performs convolution on Euclidean data, GCN takes a graph (non-Euclidean data) as the input. Nodes (or edges) of the input graph have a vector of features that contain information for training and inference. For example, in a social network, each node represents a user and with features like age, gender, etc [13].

Unlike the Euclidean data where neighbors are spatially close in the memory (multi-dimensional matrices), neighbors of graph-structured data are located apart. This results in irregular memory accesses and imposes new challenges on the processors. This section gives a brief background on GCN. Table I lists the notations and acronyms used in this paper.

### A. GCN Model

A GCN is composed of multiple layers. In each layer, feature information from the neighbors is aggregated (aggregation phase) and multiplied by a weight matrix (combination phase), and becomes the feature information for the next layer. The aggregation function can be sum, mean, max, min, Long Short Term Memory (LSTM), or other more complicated functions. The combination phase uses an MLP layer with a trainable weight matrix to transform the aggregated features and reduce the dimension of the output features.

Each layer in GCN propagates node or edge information to its one-hop neighborhood. Thus, an N-layer network effectively propagates features to its N-hop neighbors. Usually, a couple of layers is enough as the information from closer neighbors is more important than remote ones. Figure 1 shows an example of a vanilla GCN layer. Other variances of GCN algorithms have a similar model, but with different aggregation functions.

### B. Rich Diversity in GCN Models

We observe that state-of-the-art GCN models and popular input datasets come with a diverse set of aggregation functions and

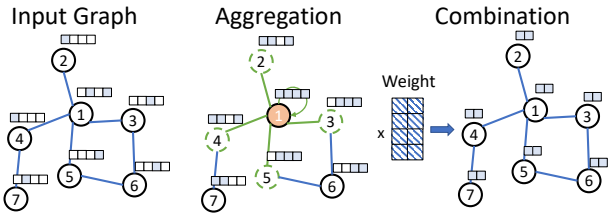



Fig. 1: An example of the Vanilla GCN layer with N=7 nodes, and F1=4 and F2=2 features. White cells are zeros. A self-loop retains the feature vector of the node for aggregation.

input feature densities. An efficient GCN processor must be flexible enough to exploit different characteristics.

Table II shows the aggregation and combination functions of the three GCN models used in this work. Note that we call the original GCN proposed in [15] vanilla GCN. Vanilla GCN takes the mean value of all neighboring nodes' features and multiplies the aggregated results with a weight matrix through an MLP layer. GraphSAGE [11] introduced neighbor sampling to vanilla GCN. GS-mean and GS-max are two variations of GraphSAGE that use *mean* and *max* for aggregation functions, respectively. In this work, we use a sampling number of 25 to be consistent with the original GraphSAGE algorithm [11].

| Algorithm | Aggregation | Combination |
|---|---|---|
| Vanilla GCN [15] | $B = mean(\mathcal{N}(H^l))$ | $X = ReLU(BW)$ |
| GS-mean [11] | $B = mean(\mathcal{N}(H^l))$ | $\sigma(W_l \cdot Concat(B, h^l))$ |
| GS-max [11] | $B = max_{j \in \mathcal{N}(h^l)} \sigma(W_l^1 \cdot h_j^l)$ | $\sigma(W_l^2 \cdot Concat(B, h^l))$ |

TABLE II: Aggregation and combination operations of GCN models [1].

We observe a variety of input datasets with different input sizes and feature matrix densities. Table III shows the information of these datasets. Cora and CiteSeer have relatively small input graphs with a sparse feature matrix. PubMed has medium size input graph with a 10% dense feature matrix, and Reddit and Ogbn-products have a large input graph with a dense matrix.

| Dataset Name | #Vertices | #Edges | F1 | $d(X)$ | X size |
|---|---|---|---|---|---|
| Cora (CR) [15] | 2708 | 10566 | 1433 | 1.27% | 385KB |
| CiteSeer (CS) [15] | 3327 | 9104 | 3703 | 0.85% | 820KB |
| PubMed (PB) [15] | 19717 | 88648 | 500 | 10% | 7.5MB |
| Reddit (RD) [11] | 232965 | 114.6M | 602 | 100% | 535MB |
| Ogbn-Prod (OP) [12] | 2449029 | 123.7M | 100 | 99% | 925MB |

TABLE III: Datasets information. All datasets contains a single graph, and all graphs are unweighted, undirected, and symmetrical. Non-zeros in the feature matrix are stored in 32-bit fixed point.

### C. Phase Orderings

The original GCN model performs the aggregation phase before the combination phase. This is similar to CNNs, where convolution is performed before feeding the results to fully-connected layers. However, prior works [8, 17] have observed that reordering the phases - that is, performing combination before aggregation - can sometimes greatly reduce the operation count. This is because the combination reduces the feature dimension, and by executing the combination phase first, the matrix multiplication in the aggregation is performed on a smaller dimension. In this work, we refer to the original order

of performing **A**ggregation phase before **C**ombination phase as the **AC** order, and the reverse order as the **CA** order.

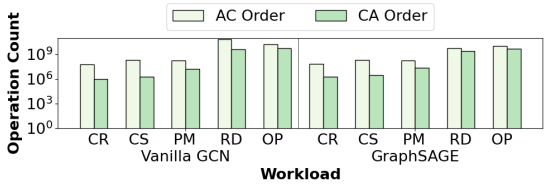



Fig. 2: Operation count for vanilla GCN, GS-mean, and GS-max models in **AC** and **CA** orders. **CA** order is not applicable to GS-max.

Figure 2 shows the total number of arithmetic operations for **AC** order and **CA** order for vanilla GCN and GraphSAGE with different datasets. On average, **CA** order achieves 93% operation count reduction for GCN and GS-mean. **CA** order does not apply to GS-max, which uses a non-linear function for aggregation. Lower operation count makes **CA** order preferable, however, it is only applicable when the alternation of the order does not affect the correctness of the output.

To ensure the correctness, aggregation function must be linear, meaning that $Agg(a,b)\times c == Agg(a\times c, b\times c)$, where $\times c$ is the combination operation. For example, addition and mean functions are linear operations, because $(a+b)\times c == (a\times c + b\times c)$, while max and min functions are not linear, because $max(a,b)\times c != max(a\times c, b\times c)$.

### D. Aggregation Dataflow

The aggregation phase is essentially the multiplication of the adjacency matrix and the feature matrix. We have three widely-used matrix multiplication methods: inner-product (IP), outer-product (OP), and row-wise (RW) as our candidates.

**Inner-product** performs element-wise operation with a row and a column of two matrices on matching indices. It exploits output data reuse because each output is written only once, but suffers from bad input reuse due to repeated reading of the second matrix for each row in the first matrix. Besides, for very sparse matrices, the odds of having matching indices are very low and can become a major overhead.

**Outer-product** performs pair-wise multiplication with a column and a row of two matrices and generates a partial matrix of the same size as the final result. Outer-product enjoys input data reuse because both input matrices are read only once, but it generates N partial result matrices and needs element-wise merging of all the partial matrices to get the final result.

**Row-wise** takes a row from the first matrix, uses its indices to retrieve the corresponding rows from the second matrix, and reduces multiple rows to one using the aggregation function. Row-wise has good output data reuse and avoids the index matching overhead in the inner-product. The downside of row-wise is bad input data reuse as the second matrix will be repetitively read, and its access pattern depends on the first matrix, causing irregular memory accesses.

Out of the three aggregation dataflows, outer-product is not suitable as it requires merging partial results to get final results, which impedes the pipelining of the aggregation and combination phases. The choice between inner-product and row-wise is explained in §III-C.

General-purpose architectures are ill-suited for efficiently executing GCN workloads, and prior accelerators are unable to adapt to a large design space of GCN workloads. Therefore, it is crucial to design an accelerator architecture that can support diverse GCN models, different phase orderings, and aggregation dataflows in order to optimize performance and power efficiency.

## III. Proposed Design

### A. PEDAL Architecture

In this section, we present the architecture design of PEDAL.

**Top-level.** Figure 3(a) shows the top-level PEDAL architecture. PEDAL has two types of Processing Elements (PEs) - Aggregation Processing Elements (APEs) for aggregation and MAC Processing Elements (MPEs) for combination. PEDAL has 32 APEs and 16 MPEs, an 8 MB feature buffer, a scheduler, and a backend HBM memory system. APEs and MPEs are connected to the scheduler, which controls the task assignment and intermediate result movement among APEs and MPEs. The feature buffer is connected to all APEs. It has 32 banks and can be used as a user-managed scratchpad or a user-transparent cache. The scheduler has a 2 MB edge buffer and a 512 KB partial result buffer. Each MPE has a 32 KB weight buffer. All MPEs, the scheduler, and the feature buffer are connected to the backend HBM memory system.

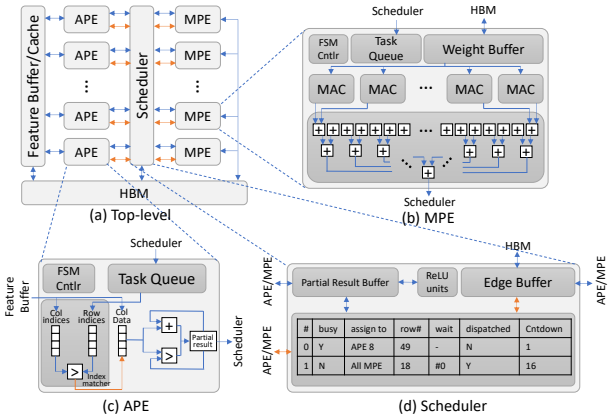



Fig. 3: PEDAL architecture. (a) is the top-level architecture, (c), (b), and (d) are the details inside APE, MPE, and the scheduler module, respectively. The blue lines in the figures are the data path, orange lines are the control path.

**APE.** APEs are used to execute aggregation operations. Figure 3(c) shows the architecture of an APE. It has a task queue to receive tasks from the scheduler, a Finite-State Machine (FSM) controller to execute the tasks, an index matcher that matches row indices with column indices for the inner product, and an accumulator and a comparator. In this work, the computing units in APEs are simplified to only an adder and a comparator for addition and max/min operations. This minimizes the area and power consumption while still allowing APEs to perform a handful of the most popular algorithms like GCN, GS-mean, and GS-max. If desired, other computing units can be added to APEs to enable other operations.

**Index matcher.** The index matcher finds the intersection of two sorted arrays. It keeps two circular queues of 32 Column and Row indexes. A naive implementation compares the top element of queues and returns them if they are equal. Otherwise, it pops the smaller one. In the worst case, this implementation needs to pop all elements of the queues sequentially. To decrease this overhead, we equip the index matcher with $2\times8$ comparators. They compare the top elements with the 8 top indices of the other queues. In one cycle, the index matcher pops as many indexes from one queue as its top element becomes smaller than or equal to the other. Compared to the naive design, we increase the performance by $3.97\times$ while we only add 6% area overhead for 16 parallel comparators.

**MPE.** MPEs are used to execute the combination phase, which is the matrix multiplication of the feature and the weight matrices. Figure 3(b) shows the architecture of an MPE. It contains a task queue that receives jobs from the scheduler and an FSM Controller for controlling the execution; instead of accessing a unified weight buffer for all MPEs, each MPE has a private weight buffer. The columns of the weight matrix are evenly distributed to MPEs, and each MPE will compute with the assigned portion of the weight matrix. Each MPE has 64 Multiply-Accumulate (MAC) units, and a hierarchical adder tree with 63 adders to reduce the MAC results. Finally, the result is sent back to the partial result buffer in the scheduler.

**Scheduler.** Scheduler is responsible for assigning tasks to APEs and MPEs and keeping track of the status of each task. For example, tasks that are assigned to multiple APEs or MPEs can retire only when all PEs finished the task. The scheduler also monitors the dependencies between aggregation and combination phases. It only dispatches tasks that have no outstanding dependencies. Finally, the scheduler takes care of the data movement between APEs and MPEs when the results of one phase are needed in another phase. Figure 3(d) shows the architecture of the scheduler. It has an edge buffer for the adjacency matrix, a partial result buffer for the intermediate results from APEs and MPEs, and a schedule table that keeps track of the status of each task.

### *B. PEDAL Dataflows*

Decoupling Aggregation and MAC PEs enables PEDAL to support diverse dataflows. In this work, we feature two ways of performing the aggregation phase: Inner-product (IP) and Row-Wise (RW), as well as two different computation orders: **AC** order and **CA** order as discussed in § II. In total, it gives us four different dataflows, namely **IP-AC**, **IP-CA**, **RW-AC**, and **RW-CA**. We describe the four dataflows in detail below:

**IP-AC**. In **IP-AC**, the feature matrix is assigned to APEs column-wisely. Each APE is equipped with a portion of the feature buffer of equal size (256KB). When the feature matrix is too big to be loaded into the feature buffer, it will be split into chunks of columns, and PEDAL loads the next chunk once the previous one is done. We assign as many columns to fill up the feature buffer of APEs, eliminating workload unbalance from uneven distribution of non-zeros. The weight matrix is dense, so we assign an equal number of columns to each MPE.

Each row of the adjacency matrix is an aggregation task, and each row of the aggregated feature matrix is a combination task. The scheduler is responsible for scheduling, dispatching, tracking, and retiring tasks. Each aggregation task is broadcast to all APEs, and each APE will perform aggregation on the adjacency matrix row with the assigned feature matrix columns using inner-product. Once an APE finishes a task, it will send the task id along with the partial results to the scheduler. When all APEs finish on a task, the scheduler retires the aggregation task and dispatches the corresponding combination task to MPEs. MPEs perform inner-product with the rows of the aggregated feature matrix and columns of the weight matrix.

**IP-CA**. **IP-CA** reverses the order of aggregation and combination to reduces the operation count. However, performing in CA order leads to a crucial issue: the combination phase generates the intermediate matrix row by row, while aggregation in the inner-product requires all rows indicated by the adjacency matrix at once, which will not be available at the time needed. Thus, we forfeit the **IP-CA** dataflow as it impedes the pipelining of aggregation and combination and hurts the performance.

**RW-AC**. **RW-AC** is an alternative way of performing GCN algorithms in AC order. In this dataflow, the feature buffer is used as a unified cache that is transparent to users. Similar to **IP-AC**, each row of the adjacency matrix is an aggregation task, but instead of broadcasting to all APEs, row-wise assigns each task to one APE. The APE retrieves the rows from the feature matrix based on the column indices of non-zeros in the adjacency matrix row. Each APE works independently from the other APEs and receives a new task upon finishing one, thus dynamically achieving workload balancing. The combination phase is the same as in **IP-AC** and is pipelined with the aggregation phase.

**RW-CA**. **RW-CA** performs the combination phase first to shrink the feature dimension, and then performs aggregation in row-wise manner. While **IP-AC** and **RW-AC** dataflow *pull* the neighboring nodes' features, **RW-CA** *pushes* the feature of a node to all its neighbors. This is because the combination phase generates the intermediate feature matrix row by row, and it is not feasible to *pull* features from neighbors as they may not be ready yet. In **RW-CA**, the combination is performed the same way as in **IP-AC** and **RW-AC**. When a row of the intermediate feature matrix is generated, it is broadcast to all APEs. Each row of the adjacency matrix is an aggregation task, and it is evenly split into slices and assigned to all APEs. Each APE is responsible for aggregating the feature to the slice assigned, so there is no memory contention across APEs. The feature buffer is evenly assigned to each APE (256KB) and used as a cache.

### *C. Choosing the Right Dataflow*

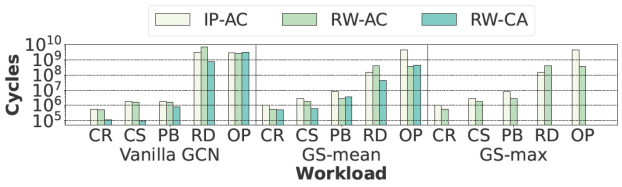


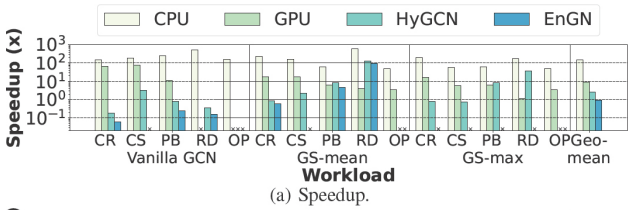

Fig. 4: Performance of **IP-AC**, **RW-AC**, and **RW-CA**.

Figure 4 shows the performance of each dataflow for vanilla

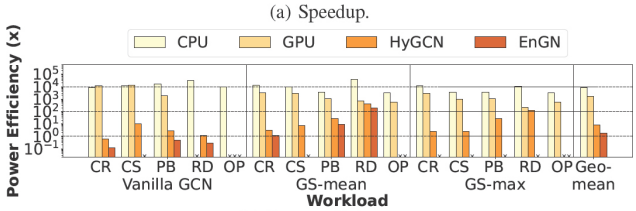
GCN, GS-mean, and GS-max with 5 real-world datasets. The choice of dataflow (**IP-AC**, **RW-AC**, or **RW-CA**) significantly affects the execution time. We need to pick the best dataflow for each GCN algorithm and dataset pair. A simple approach is to find the number of arithmetic operations and compare them for different dataflows. However, this approach does not take memory stalls into account. For example, while the operation count of vanilla GCN and Reddit dataset is the same for **IP-AC** and **RW-AC** dataflows, the high cache miss rate of **RW-AC** dataflow results in a longer execution time compared to **IP-AC**. The simple approach chooses the right dataflow in only 73% of the evaluated GCN model and input dataset pairs. A better approach is needed to make educated decisions based on the dataset characteristics and the GCN model. We use N, NNZ(A), NNZ(X), and F1 as dataset characteristics, which are the input dataset metadata.

With the complexity of so many dataset parameters, GCN variances, and execution orders, we need a decision tree to choose the best dataflow. We created 400 synthetic datasets where N ranges from 1K to 1M, NNZ(A) from 2K to 200M, F1 from 100 to 3K, and NNZ(X) from 1K to 3B. We pick these parameters because they reflect the sizes and densities of the input graphs and input features. These ranges are large enough to cover all the real-world datasets we evaluate in this paper. Besides, the synthetic datasets are generated such that non-zeros in adjacency matrices have power-law distribution, and non-zeros in feature matrices have Gaussian distribution based on our observation from the real-world datasets. We build a decision tree using scikit-learn [19], which uses an optimized version of CART (Classification and Regression Trees). We use the synthetic datasets to train the decision tree and use it to predict the best dataflow on real-world datasets.

## IV. Evaluation




(a) Speedup.




(b) Power efficiency.

Fig. 5: Speedup and power efficiency of PEDAL compared to CPU, GPU, HyGCN and EnGN. Power efficiency is measured by power-delay product. × markers mean missing data points due to GPU Out-Of-Memory or prior accelerators not reporting for some datasets or not supporting some GCN models.

### A. Experimental Setup

**Baseline.** We evaluate the CPU performance on Intel Xeon Gold 6230 CPU, and GPU performance on NVIDIA GPU with Ampere architecture. We implement the baseline on the state-of-the-art PyTorch Geometric [7] library. We also compare PEDAL with two prior GNN accelerators: HyGCN [23] and EnGN [17] using the reported performance numbers. Table IV shows the configurations of PEDAL and baseline architectures.

**PEDAL simulation.** We build a cycle-accurate simulator in python and C++ to measure the computation cycles of PEDAL. Our simulator is event-based and controlled by a state machine to enforce dependencies. The memory access trace is recorded and fed to Ramulator [14] for memory access latency. Ramulator includes both cache and HBM memory as a hierarchical memory system. We implement our design in RTL and use Design Compiler to synthesize with a commercial 12nm CMOS library at 1GHz clock frequency. We use eDRAM for on-chip memory of PEDAL, HyGCN, and EnGN, and analyze with CACTI [18].

**GCN algorithms and datasets.** We evaluate the vanilla GCN algorithm and two variations of GraphSAGE: GS-mean and GS-max in this work. The details of the algorithms can be found in Table II. The datasets used in this work can be found in Table III. We cover graph size from small to large, and with the feature matrix from sparse to dense to have a thorough comparison between PEDAL and prior works. We use the same hidden dimension (128) as in HyGCN for layer 1.

### B. Decision tree accuracy

We use 80% of the synthetic datasets to train the decision tree and test on the remaining 20% and achieved 90% accuracy. Then we apply the decision tree to the real-world datasets, and it selects the best dataflow with 93.3% accuracy. The only mistake happens on the vanilla GCN and OP dataset in which the mispredicted dataflow (**IP-AC**) is only 9% slower than the best dataflow (**RW-AC**). We calculate the ratio of execution time between the decision tree selected dataflow and the best dataflow on the synthetic test set. The average ratio is 1.047, meaning that the decision tree selected dataflow has an execution time expectation less than 5% higher than the best dataflow.

### C. Speedup and power efficiency

Figure 5a shows the performance of PEDAL compared to CPU, GPU, and prior accelerators. We select the best dataflow for different GCN algorithm and dataset pairs.

On average, PEDAL outperforms CPU and GPU by 144.5× and 9.36×. Compared to prior accelerators, despite having less computing resources, PEDAL achieves 2.55× speedup over HyGCN. Compared to EnGN, PEDAL also supports non-linear aggregation functions (e.g., GS-max) while achieving similar performance for linear aggregation functions.

Compared to prior works, where thousands of PEs are used for better performance, PEDAL uses only 32 APEs and 16 MPEs to achieve similar or better performance in most of cases. Figure 5b shows power efficiency of PEDAL. Power efficiency is measured using the power-delay product. On average, PEDAL achieves 8856×, 1606×, 8.4× and 1.78× better power efficiency than CPU, GPU, HyGCN, and EnGN respectively. PEDAL is conservative on adding an excessive amount of PEs because a) too many APEs to access the feature buffer will cause serialization issue, b) too many APEs will cause cache thrashing to the capacity-limited feature buffer, c) an

| | Compute Unit | On-chip Memory | Off-chip Memory | Area($mm^2$) | Power(W) |
|---|---|---|---|---|---|
| CPU | 80 cores @ 2.1GHz | 96MB | 256 GB/s DDR4 | - (14nm) | 125 |
| GPU | 10496 Shading Units @ 1.4GHz | 16.25MB | 936.2 GB/s | 628 (8nm) | 350 |
| HyGCN | 16 SIMD cores @ 1GHz, and 32x128 systolic array | 22.1MB | 256 GB/s HBM | 7.8 (12nm) | 6.7 |
| EnGN | 128x16 arrays @ 1GHz | 1.6MB | 256 GB/s HBM | 3.54 (14nm) | 3.87 |
| PEDAL | 32 APEs and 16 MPEs @ 1GHz | 11MB | 256 GB/s HBM | 4.05 (12nm) | 2.04 |

TABLE IV: Architecture configuration comparison of CPU, GPU, HyGCN, EnGN, AWB-GCN and PEDAL

appropriate ratio of APEs and MPEs is important to load balance between aggregation and combination. By employing a lower number of PEs, PEDAL achieves lower power consumption while keeping a comparable performance, thus having better power efficiency.

### D. Power and area breakdown

| Module | Components | Power | Area |
|---|---|---|---|
| APE | Accumulator | 0.2% | 0.07% |
| | Index Matcher | 7.7% | 1.80% |
| | Controller | 0.7% | 0.20% |
| | TaskQueue | 1.04% | 2.98% |
| MPE | Adder Tree | 2.4% | 6.42% |
| | MAC | 42.8% | 11.41% |
| | Controller | 0.45% | 0.31% |
| | Weight Buffer | 0.9% | 12.91% |
| | Task Queue | 0.53% | 1.48% |
| Feature Buffer | Buffer | 40% | 49.93% |
| Scheduler | Partial Results | 0.07% | 0.40% |
| | Edge Buffer | 2.9% | 11.83% |
| | Controller | 0.06% | 0.07% |
| | ReLU | 0.15% | 0.00% |

TABLE V: Power and Area breakdown

PEDAL has an average power consumption of 2.04W, which is 69.6% and 47.3% lower than HyGCN and EnGN, respectively (Table IV). Compared to HyGCN with general-purpose SIMD units, PEDAL customizes processing elements and requires less computing power. Besides, the limited feature buffer size of EnGN increases the miss rate drastically for large datasets, for example, Reddit. This results in higher eDRAM power consumption compared to PEDAL that uses 8MB of feature buffer. The total area of PEDAL is 4.05 $mm^2$, which is 48.1% smaller than HyGCN, and 14.4% higher than EnGN, respectively. Table V lists the power and area breakdown of each component.

### E. Discussion

PEDAL supports multiple dataflows and phase orderings. Figure 2 shows that operation distributions varies in different dataflows, making either APEs or MPEs the bottleneck; solely adding more resources to the architecture can only help certain dataflow but not all. Besides, PEDAL also needs to support Row-Wise mode, where the feature buffer is used as a unified cache. Adding too many APEs requires increasing the size of the feature buffer to ensure access latency. Either of the solutions is too expensive for the potential performance gain of this design.

## V. Conclusion

In this work, we present PEDAL, a power-efficient accelerator for GCN inference supporting multiple dataflows. In order to accommodate different input graph sizes and densities, as well as GCN variants with different aggregation functions, PEDAL features multiple dataflows, namely **IP-AC**, **RW-AC**, and **RW-CA** to support performing GCN inference in both phase orderings efficiently. We evaluate the performance of PEDAL using a cycle-accurate simulator and do RTL synthesis to get power and area. PEDAL achieves 144.5×, 9.36×, and 2.55× speedup compared to CPU, GPU, and HyGCN respectively, and 8856×, 1606×, 8.4× and 1.78× better power efficiency compared to CPU, GPU, HyGCN and EnGN respectively.

## References


[1] S. Abadal *et al.*, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Comput. Surv.*, vol. 54, no. 9, oct 2021.
[2] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2016.
[3] P. W. Battaglia *et al.*, "Interaction networks for learning about objects, relations and physics," *CoRR*, vol. abs/1612.00222, 2016.
[4] R. Chakraborty *et al.*, "Manifoldnet: A deep neural network for manifold-valued data with applications," *IEEE TPAMI*, pp. 1–1, 2020.
[5] C. Chen *et al.*, "Regnn: A redundancy-eliminated graph neural networks accelerator," in *2022 HPCA*, 2022, pp. 429–443.
[6] J. Chen *et al.*, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *ICLR*, 2018.
[7] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *CoRR*, vol. abs/1903.02428, 2019.
[8] T. Geng *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *MICRO*, 2020, pp. 922–936.
[9] ——, "I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO*, 2021.
[10] A. Graves *et al.*, "Multi-dimensional recurrent neural networks," in *Artificial Neural Networks - ICANN*, 2007.
[11] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018.
[12] W. Hu *et al.*, "Open graph benchmark: Datasets for machine learning on graphs," 2020. [Online]. Available: https://arxiv.org/abs/2005.00687
[13] M. Kim and J. Leskovec, "Modeling social networks with node attributes using the multiplicative attribute graph model," 2011.
[14] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, pp. 45–49, 2016.
[15] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017.
[16] Y. Lecun *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.
[17] S. Liang *et al.*, "Engn: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE TC*, 2021.
[18] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches."
[19] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.
[20] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016.
[21] F. Scarselli *et al.*, "The graph neural network model," *IEEE Transactions on Neural Networks*, 2009.
[22] N. Talati *et al.*, "A deep dive into understanding the random walk-based temporal graph learning," in *IISWC*, 2021.
[23] M. Yan *et al.*, "Hygcn: A gcn accelerator with hybrid architecture," in *HPCA*, 2020.
[24] H. You *et al.*, "Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design," in *HPCA*, 2022.
[25] J. Zhou *et al.*, "Graph neural networks: A review of methods and applications," *AI Open*, 2020.
[26] C. Zhuang and Q. Ma, "Dual graph convolutional networks for graph-based semi-supervised classification," in *WWW*, 2018.