GNNHLS: Evaluating Graph Neural Network Inference via High-Level Synthesis

Supplemental Material

I. Introduction

GNNHLS is an open-source framework for comprehensive evaluation of GNN kernels on FPGA via high-level synthesis (HLS). It includes 6 commonly-used GNN applications: Graph Convolutional Network (GCN) [5], GraphSage (GS) [3], Graph Isomorphism Network (GIN) [14], Graph Attention Network (GAT) [11], Mixture Model Networks (MoNet) [9], and Gated Graph ConvNet (GatedGCN) [1]. This article is the supplementary material for [15]. Section II describes the details of GNN kernels. Section III articulates the experimental methodology. Section IV describes the characterization results of GNN kernels. Section V includes the absolute experiment results in terms of excution time and energy consumption.

II. GNN Kernels

GNN HLS kernels are optimized with several optimization techniques. These optimization techiniques are described here.

- **Pipeline**: Enable instruction-level concurrent execution.
- **Loop Merge**: Optimize the finite state machine (FSM) of nested loops to remove the throughput impact of inner loop latency.
- **Burst Memory Access** & **Memory Port Widening**: access chunks of data in contiguous addresses and increase memory port width.
- **Loop Unroll**: Leverage instruction-level parallelism by executing multiple copies of loop iterations in parallel.
- **Dataflow**: Enable task-level parallelism by connecting multiple functions with FIFOs to form a pipeline-style architecture.
- **Multiple Compute Units (CUs)**: Execute multiple kernel instances as CUs in parallel for different data portions.

Figure 1 illustrates the dataflow diagrams of the GNNHLS kernels, in which memory and computation operations are divided and pipelined based on the complexity of each kernel. To mitigate the cost of dataflow, we also (1) tune the location of FIFO accesses to achieve better throughput, (2) apply vectors for FIFO widening and associated operations, and (3) split loops to optimize the FIFO properties of loop indices.

A. Graph Convolutional Network (GCN)

Graph Convolutional Network (GCN) [5] is one of the earliest GNN models and has a simple structure. It updates node features by aggregating neighboring node features and performing linear projection. The formula is given as follows:

$$h_i^{l+1} = \mathrm{ReLU}\left(U^l \sum_{j \in N_i} h_j^l\right) \quad (1)$$

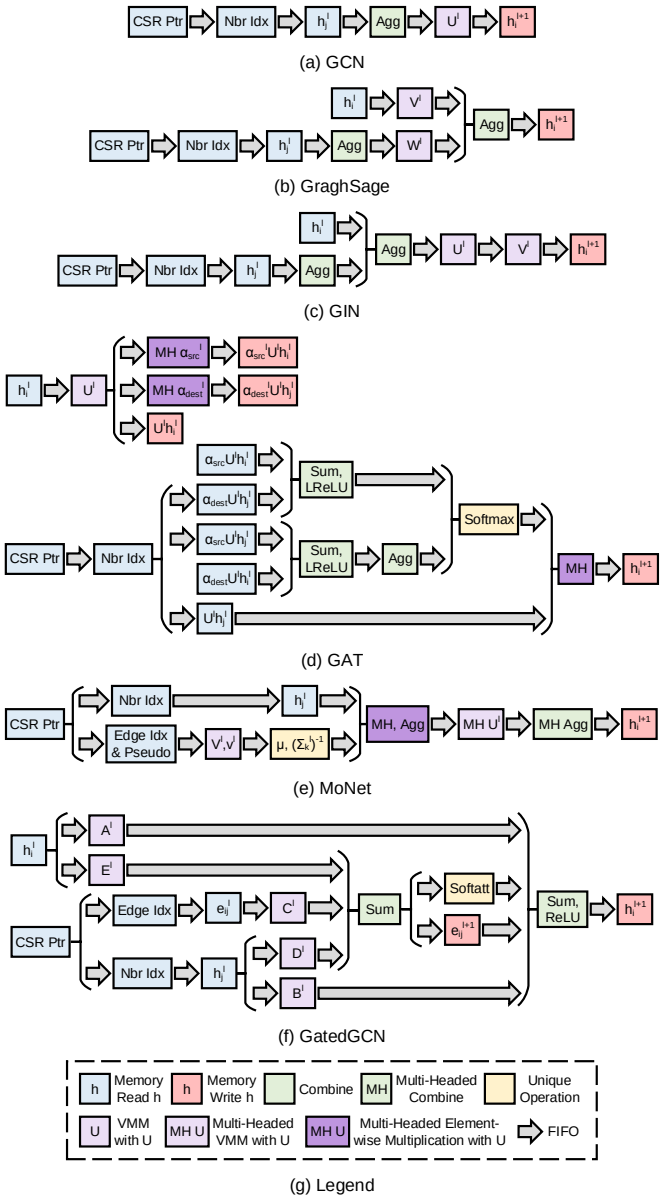

Fig. 1. Dataflow diagrams of GNN HLS kernels in GNNHLS.

Where $U^l \in \mathbb{R}^{d\times d}$ is the learnable weight matrix of the linear projection, which performs vector-matrix multiplication. $h_i^l \in \mathbb{R}^{d\times 1}$ is the feature vector of vertex i in layer l, and N_i represents the neighboring vertices of vertex i.

Based on the above equation, we create the GCN HLS implementation, the dataflow diagram of which is depicted in Figure 1(a). In addition to the memory access modules for input graphs and h, we split the computation operations into two modules: Aggregation of neighbor node vectors h_j and vector-matrix multiplication (VMM) for linear projection. We perform all the optimization techniques described previously to the GCN kernel. The memory burst length vector h is d, limited by the irregularity of the graph topology. The initiation interval (II) of the aggregation module is $4\,|N_i| + 2$. Since Vitis is not good at synthesizing tree-structured floating-point operations, we separate VMM into 2 functions in the dataflow scope for grouped VMM and sum, respectively. The II of VMM is thereby reduced from d^2 to $d+36$. All these modules are reused in the following GNN models. Due to its simplicity, we create 2 CUs to process distinct vertices in parallel.

B. GraphSage (GS)

GraphSage (GS) [3] introduces an inductive framework to improve the scalability over GCN by aggregating information from the fixed-size set of neighbors via uniform sampling, explicitly incorporating feature vectors of both the target vertex and its source neighbors. The mathematical expression of GraphSage with a mean aggregator is formulated as follows:

$$\begin{aligned} h_i^{l+1} &= \mathrm{ReLU}\left(U^l \mathrm{Concat}\left(h_i^l, \frac{1}{|N_i|}\sum_{j\in N_i} h_j^l\right)\right) \\ &= \mathrm{ReLU}\left(V^l h_i^l + W^l \frac{1}{|N_i|}\sum_{j\in N_i} h_j^l\right) \end{aligned} \tag{2}$$

Where N_i is the set of source neighbors of vertex i, and $h_i^l \in \mathbb{R}^{d\times 1}$ is the feature vector of vertex i in layer l. The learnable weight matrix of the linear projection, $U^l \in \mathbb{R}^{d\times 2d}$, is stored in on-chip memory. Given that distinct weight parameters are used for the target vertex and source neighbors, U^l is divided into $V^l \in \mathbb{R}^{d\times d}$ and $W^l \in \mathbb{R}^{d\times d}$, enabling parallel execution of both paths to hide the latency of linear projection for the target vertex. Figure 1(b) illustrates the dataflow structure of GraphSage. The memory read accesses and linear projection of the target feature, and neighbors' feature aggregation are executed simultaneously, and then summed up to update h_i.

C. Graph Isomorphism Network (GIN)

Graph Isomorphism Network (GIN) [14] employs the Weisfeiler-Lehman Isomorphism Test [13] as its foundation to investigate the discriminative ability of GNNs. The formula of GIN is described as follows:

$$h_i^{l+1} = \mathrm{ReLU}\left(U^l \mathrm{ReLU}\left(V^l\left((1+\epsilon)h_i^l + \sum_{j\in N_i} h_j^l\right)\right)\right) \tag{3}$$

where ϵ is a learnable scalar weight, U^l and $V^l \in \mathbb{R}^{d\times d}$ denote learnable weight matrices of cascaded VMM modules, $h_i^l \in \mathbb{R}^{d\times 1}$ again refers to the feature vector of vertex i in layer l, and N_i is again the source neighbors of vertex i. In contrast to GraphSage, GIN illustrated in Figure 1(c) first sums up the aggregated vector of neighbors h_j and the target vertex vector h_i, hiding the latency of reading h_i, then performs two cascaded VMM modules with weight matrices U^l and V^l, respectively. This framework avoids the generation of long critical paths and achieves a higher clock frequency.

D. Graph Attention Network (GAT)

Graph Attention Network (GAT) [11] is an anisotopic GNN model that uses self-attention mechanisms to weight and learn representations of neighbor vertices unequally. The equation is described as follows:

$$h_i^{l+1} = \mathrm{Concat}_{k=1}^{K}\left(\mathrm{ELU}\left(\sum_{j\in N_i} \alpha_{ij}^{k,l} U^{k,l} h_j^l\right)\right) \tag{4}$$

$$\alpha_{ij}^{k,l} = \mathrm{Softmax}(e_{ij}^{k,l}) = \frac{\exp(e_{ij}^{k,l})}{\sum_{j'\in N_i}\exp(e_{ij'}^{k,l})} \tag{5}$$

$$\begin{aligned} e_{ij}^{k,l} &= \mathrm{LeakyReLU}(\vec{a}^T Concat(U^{k,l}h_i^l, U^{k,l}h_j^l)) \\ &= \mathrm{LeakyReLU}(a_{src}^{k,l}U^{k,l}h_i^l + a_{dest}^{k,l}U^{k,l}h_j^l) \end{aligned} \tag{6}$$

where $\alpha_{ij}^l \in \mathbb{R}^K$ is the attention score between vertex i and vertex j of layer l, $U^{k,l} \in \mathbb{R}^{d\times d}$ and $\vec{a} \in \mathbb{R}^{2d}$ are learnable parameters. Note that the weight parameter $\vec{a}^T$ is decomposed into a_{src}^l and $a_{dest}^l \in \mathbb{R}^d$ in the DGL library, because it is more efficient in terms of performance and memory footprint by transferring VMM between $U^{k,l}$ and h^l from edge-wise to node-wise operations, especially for sparse graphs where the edge number is larger than the vertex number.

Figure 1(d) depicts the dataflow framework of GAT. Due to the unbalanced workload of the numerator and the denominator in (5), the results of $\exp(e_{ij})$, size $O(|N_i|)$, need to be temporarily stored prior to being accumulated. Considering the irregularity and large maximum $|N_i|$ of graphs, we divide the GAT model into 2 HLS kernels linked to the same memory banks for shared intermediate results: kernel 1 is designed to perform VMM with U and h, and multi-headed element-wise multiplication (MHEWM) with a_{src} and a_{dest}, respectively, in (6). After being optimized, the II of MHEWM is $k+112$. The intermediate results are written back to memory and then read by kernel 2 to implement (4) and (5). Note that e_{ij} is computed twice in parallel to avoid performance degradation and deadlock issues. The II of aggregation, softmax, and MHEWM are $k\cdot|N_i| + 2k + 38$, $k\cdot|N_i| + k + 17$, and $k\cdot|N_i| + k + 14$, respectively.

E. Mixture Model Networks (MoNet)

Mixture Model Networks (MoNet) [9] is a general anisotopic GNN framework designed for graph and node classification tasks using Baysian Gaussian Mixture Model (GMM) [2]. The model is formulated as follow:

$$h_i^{l+1} = \mathrm{ReLU}\left(\sum_{k=1}^{K}\sum_{j\in N_i} w_k(u_{ij})U^{k,l}h_j^l\right)$$
$$= \mathrm{ReLU}\left(\sum_{k=1}^{K} U^{k,l}\sum_{j\in N_i} w_k(u_{ij})h_j^l\right) \quad (7)$$

$$w_k(u_{ij}) = \exp\left(-\frac{1}{2}(u_{ij}^l-\mu_k^l)^T(\textstyle\sum_k^l)^{-1}(u_{ij}^l-\mu_k^l)\right) \quad (8)$$

$$u_{ij}^l = \mathrm{Tanh}(V^l pseudo_{ij}^l + v^l) \quad (9)$$

$$pseudo_{ij}^l = \mathrm{Concat}(deg_i^{-0.5}, deg_j^{0.5}) \quad (10)$$

where $v^l \in \mathbb{R}^2$, $V^l \in \mathbb{R}^{2\times2}$, $\mu \in \mathbb{R}^{K\times2}$, $(\sum_k^l)^{-1} \in \mathbb{R}^{K\times2}$, and $U^l \in \mathbb{R}^{d\times d}$ are learnable parameters of GMM. v^l and V^l represent the pseudo-coordinates between the target vertex and its neighbors, $\mu \in \mathbb{R}^{K\times2}$ and $(\sum_k^l)^{-1} \in \mathbb{R}^{K\times2}$ denote the mean vector and covariance matrix. $U^{k,l}$ is the weight matrix.

The dataflow diagram of MoNet is depicted in Figure 1(e). In our HLS implementation, $pseudo_{ij}$ of each edge is processed by a small VMM module with V^l and v^l in (9) and the Gaussian Weight Computation module with μ and $(\sum_k^l)^{-1}$ in (8). Meanwhile, h_j is read from memory for the subsequent MHEWM with aggregation, MHVMM with U, and MH Aggregation modules. Note that we perform the MH VMM with U after aggregation in (7), transferring it from an edge-wise to node-wise operation to reduce its occurrence. After optimization, the II of the VMM for u_{ij}, Gaussian computation, MHEWM with aggregation, MHVMM with U, and MH Aggregation are 1, 1, 4, $d+k+28$, and $7k+10$, respectively. We create 2 CUs for the HLS kernel to process vertices with distinct indices.

F. Gated Graph ConvNet (GatedGCN)

The Gated Graph ConvNet (GatedGCN)) [1] is a type of anisotropic graph neural network (GNN) model that employs a gating mechanism to regulate the flow of information during message passing, allowing the model to emphasize relevant information and filter out irrelevant data. The gating mechanism utilizes gate functions (e.g., sigmoid) to control the flow of messages at each layer. The mathematical expression for GatedGCN is provided below:

$$h_i^{l+1} = \mathrm{ReLU}\left(A^l h_i^l + \frac{\sum_{j'\in N_i} B^l h_{j'}^l \odot \sigma(e_{ij'}^{l+1})}{\sum_{j'\in N_i}\sigma(e_{ij'}^{l+1})+\epsilon}\right) \quad (11)$$

$$e_{ij}^{l+1} = E^l h_i^l + D^l h_j^l + C^l e_{ij}^l \quad (12)$$

where A^l, B^l, D^l, E^l and $C^l \in \mathbb{R}^{d\times d}$ are learnable matrix parameters, $e_{ij}^l \in \mathbb{R}^{1\times d}$ denote the edge features from vertex i to j layer l, h_i^l represents node features of vertex i in layer l, $\odot$ denotes Hadamard product, σ denotes the sigmoid function, and ϵ is a constant for numerical stability.

Since the soft attention of GatedGCN shown in (11) is distinct from GAT, performing accumulation operations for e_{ij} on both the numerator and denominator, we implement a single pipeline to build the HLS kernel. Figure 1(f) illustrates the dataflow framework of GatedGCN. To hide the latency of multiple VMM modules in GatedGCN, we perform all of them in parallel with parameters A, B, D, E, and C, respectively. Then the soft attention module is implemented to update h_i. After optimization, the II of the soft attention and sum modules to generate h_i^{l+1} are $10\cdot|N_i|+72$ and 31, respectively.

TABLE I
GRAPH DATASETS.

Dataset	Node #	Edge #	Max. Deg.	Avg. Deg.
OGBG-MOLTOX21 (MT)	145459	302190	6	2.1
OGBG-MOLHIV (MH)	1049163	2259376	10	2.2
OGBN-ARXIV (AX)	169343	1166243	13155	6.9
OGBN-PROTEINS (PT)	132534	79122504	7750	597.0

III. EXPERIMENTAL METHODOLOGY

Datasets: Table I shows the graph datasets used in our evaluation. All these graphs are collected from Open Graph Benchmark [4] and have a wide range of fields and scales. These graphs represent two classes of graphs with distinct topologies: MH and MT consist of multiple small dense graphs, while AX and PT each consist of one single sparse graph. The maximum and average degree shown in Table I indicates their varying distributions ranging from regular-like to powerlaw-like. In addition, we set feature dimensions for the kernels: GCN, GraphSage, and GIN have the same input and output dimensions at 128. The input, head, and output dimensions of GAT and MoNet are (128, 8, 16) and (64, 2, 64), respectively. All the dimensions of GatedGCN are 32.

Evaluation methods: To perform evaluation, we use a Xilinx Alveo U280 FPGA card, provided by the Open Cloud Testbed [6], to execute the HLS kernels. This FPGA card provides 8 GB of HBM2 with 32 memory banks at 460 GB/s total bandwidth, 32 GB of DDR memory at 38 GB/s, and 3 super logic regions (SLRs) with 1205K look-up tables (LUTs), 2478K registers, 1816 BRAMs, and 9020 DSPs. We adopt 32-bit floating point as the data format. We use Vitis 2020.2 for synthesis and hardware linkage with the power-profile option enabled to perform power profiling during runtime, and Vitis Analyzer to view resource utilization, execution time and power consumption. We compare our HLS implementation with CPU and GPU baselines with PyTorch and the highly-optimized DGL library. We perform CPU baseline runs on an Intel Xeon Silver 4114 at 2.2 GHz with 10 cores, 20 threads, and 13.75 MB L3 cache. The GPU baseline is implemented on an Nvidia RTX 2080 Ti with 2994 CUDA cores at 1.5 GHz and 8 GB GDDR6 at 448 GB/s total bandwidth. We measure the energy consumption of the CPU and GPU baselines using the same technique as prior work [8].

IV. CHARACTERIZATION

To capture insight into the properties of GNNHLS, we first characterize the GNN kernels using instruction mix,

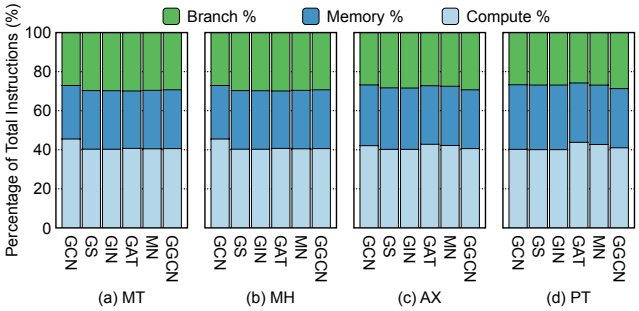

Fig. 2. Instruction breakdown of all the HLS kernels.

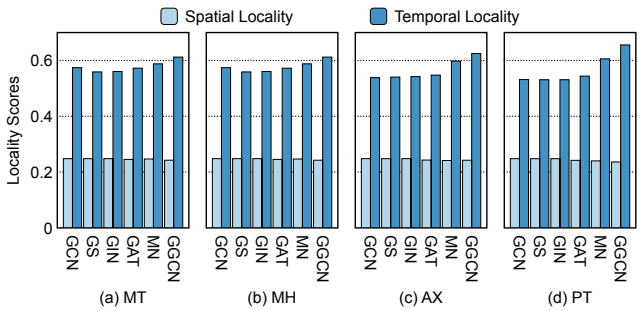

Fig. 3. Memory locality scores of HLS kernels.

spatial locality, and temporal locality. We use Workload ISA-Independent Characterization (WIICA) [10], a workload characterization tool, to capture ISA-independent properties by generating and parsing a dynamic trace of runtime information. Due to the limits of disk and processing time, profiling the full trace is impractical. Thus we use uniform random node sampling [7] to select a sequence of 500 nodes for evaluation.

A. Instruction Mix

We first take a look at the dynamic instruction mix, partitioning instructions into 3 classes: branch, memory and compute. Figure 2 shows the instruction mix of the HLS kernels on the 4 datasets. We observe that the instruction breakdown shows a consistent tendency: (1) The computation instructions make up the largest fraction (about $40\%-50\%$) of total instructions, implying that these pipeline-style GNN HLS kernels are computation-intensive. (2) Memory instructions consume the second largest fraction (about 30% 35%), indicating the total number of memory accesses is still nontrivial even if all the kernels are in a pipeline style. (3) While branch instructions take $25\%-30\%$ of the total, most of them are due to conditional statements of for loops and irregularity of graphs. We also observe that denser graphs (e.g., AX and PT) induce a higher fraction of compute instructions for anisotropic kernels (i.e., GAT, MN, and GGCN) due to their edge-wise operations. In contrast, denser graphs induce a higher fraction of memory instructions for isotropic kernels (i.e., GCN, GS, and GIN) because their edge-wise operations are less computation intensive than node-wise update.

B. Spatial and Temporal Locality

We use spatial locality and temporal locality scores developed by Weinberg et al. [12] to quantitatively measure the memory access patterns. Spatial locality characterizes the closeness of memory references among consecutive memory accesses. For HLS accelerators, it represents the potential opportunity to optimize the efficiency of prefetching and memory burst transfer. Temporal locality measures the frequency of memory instructions accessing the same memory address. It represents the latent efficiency of cacheing data elements so that they can be accessed repetitively with lower cost. Therefore, the higher the temporal locality, the more performance improvement due to cacheing mechanisms in the accelerators. Both return a score in the range $[0, 1]$.

Figure 3 illustrates the spatial and temporal locality scores. Focusing first on the spatial locality, we observe the score stays consistently low (about $0.23-0.25$) across all the kernels and datasets. It is because the irregularity of graph topology induces non-contiguous memory references, limiting memory burst transfer and prefetching to the length of feature sizes. Next examining the temporal locality, we observe that the score stays in the range of $0.5-0.7$, indicating the potential performance benefit of cacheing mechanisms, regardless of the graph topology. In addition, we observe anisotropic kernels show a higher temporal locality than isotropic kernels, due to them having more edge-wise operations.

V. Evaluation

In this section, we present the absolute experiment results in terms of performance and energy. Table II shows the execution time of the CPU and GPU baselines and the HLS kernels. Table III shows the energy consumption of the baselines and the HLS kernels.

References

[1] X. Bresson and T. Laurent, "Residual gated graph convnets," *arXiv preprint arXiv:1711.07553*, 2017.
[2] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *J. Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977.
[3] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Adv. Neural Inf. Process. Syst.*, vol. 30, 2017.
[4] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *Adv. Neural Inf. Process. Syst.*, vol. 33, 2020.
[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. of Int'l Conf. on Learning Rep.*, 2017.
[6] M. Leeser, S. Handagala, and M. Zink, "FPGAs in the cloud," *Computing in Science & Engineering*, vol. 23, no. 6, pp. 72–76, 2021.
[7] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proc. of 12th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, 2006, pp. 631–636.
[8] Y. C. Lin, B. Zhang, and V. Prasanna, "GCN inference acceleration using high-level synthesis," in *Proc. of IEEE High Performance Extreme Computing Conf.*, 2021.
[9] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein, "Geometric deep learning on graphs and manifolds using mixture model CNNs," in *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition*, 2017, pp. 5115–5124.

TABLE II
EXECUTION TIME (SEC) OF DGL-CPU, DLG-GPU, AND GNN HLS IMPLEMENTATION ON 4 GRAPH DATASETS.

	MT			MH			AX			PT		
	DGL-CPU	DGL-GPU	HLS	DGL-CPU	DGL-GPU	HLS	DGL-CPU	DGL-GPU	HLS	DGL-CPU	DGL-GPU	HLS
GCN	0.11	0.28	0.05	0.69	0.35	0.39	0.31	0.34	0.21	16.09	6.29	14.85
GS	0.21	0.30	0.13	1.42	0.38	0.98	0.43	0.42	0.52	16.45	5.68	34.29
GIN	0.15	0.29	0.13	0.93	0.35	0.98	0.34	0.41	0.52	16.11	5.15	34.29
GAT	0.91	0.12	0.21	6.52	0.24	1.51	3.10	0.27	0.67	186.93	OoM	28.28
MN	0.32	0.11	0.05	2.37	0.18	0.32	1.18	0.21	0.05	89.71	OoM	1.77
GGCN	0.12	0.11	0.17	0.62	0.26	1.26	0.36	0.26	0.54	38.93	OoM	33.55

TABLE III
ENERGY CONSUMPTION (J) OF DGL-CPU, DGL-GPU, AND GNN HLS IMPLEMENTATION ON 4 GRAPH DATASETS.

	MT			MH			AX			PT		
	DGL-CPU	DGL-GPU	HLS	DGL-CPU	DGL-GPU	HLS	DGL-CPU	DGL-GPU	HLS	DGL-CPU	DGL-GPU	HLS
GCN	9.06	59.67	0.80	58.38	75.25	5.85	25.93	73.38	3.10	1367.75	1352.67	208.77
GS	17.95	64.60	1.68	120.97	80.63	12.73	36.74	89.54	6.69	1397.99	1221.69	439.91
GIN	13.12	63.20	1.77	79.25	75.04	13.40	29.10	89.11	7.10	1369.04	1107.06	464.29
GAT	77.45	25.37	2.79	554.10	50.53	20.50	263.09	57.74	8.83	15889.04	OoM	344.14
MN	27.46	24.32	0.80	201.19	38.70	6.48	100.59	45.59	0.75	7625.48	OoM	17.22
GGCN	9.84	23.82	1.62	53.12	55.47	12.05	30.76	55.32	5.00	3309.16	OoM	323.44

[10] Y. S. Shao and D. Brooks, "ISA-independent workload characterization and its implications for specialized architectures," in *Proc. of IEEE Int'l Symp. on Perf. Analysis of Systems and Software*, 2013, pp. 245–255.

[11] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," in *Proc. of Int'l Conf. on Learning Rep.*, 2017.

[12] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely, "Quantifying locality in the memory access patterns of HPC applications," in *Proc. of ACM/IEEE Conf. on Supercomputing*, 2005.

[13] B. Weisfeiler and A. Leman, "The reduction of a graph to canonical form and the algebra which appears therein," *Nauchno-Technicheskaya Informatsia*, vol. 2, no. 9, pp. 12–16, 1968.

[14] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *Proc. of Int'l Conf. on Learning Rep.*, 2019.

[15] C. Zhao, Z. Dong, Y. Chen, X. Zhang, and R. D. Chamberlain, "GNNHLS: Evaluating Graph Neural Network Inference via High-Level Synthesis," in *Proc. of 41st IEEE Int'l Conf. on Computer Design*, Nov. 2023.