AMPLE: Event-Driven Accelerator for Mixed-Precision Inference of Graph Neural Networks

PEDRO GIMENES, Imperial College London, United Kingdom
AARON ZHAO, Imperial College London, United Kingdom
GEORGE CONSTANTINIDES, Imperial College London, United Kingdom

Graph Neural Networks (GNNs) have recently gained attention due to their performance on non-Euclidean data. The use of custom hardware architectures proves particularly beneficial for GNNs due to their irregular memory access patterns, resulting from the sparse structure of graphs. However, existing FPGA accelerators are limited by their double buffering mechanism, which doesn't account for the irregular node distribution in typical graph datasets. To address this, we introduce **AMPLE** (Accelerated Message Passing Logic Engine), an FPGA accelerator leveraging a new event-driven programming flow. We develop a mixed-arithmetic architecture, enabling GNN inference to be quantized at a node-level granularity. Finally, prefetcher for data and instructions is implemented to optimize off-chip memory access and maximize node parallelism. Evaluation on citation and social media graph datasets ranging from 2K to 700K nodes showed a mean speedup of 243× and 7.2× against CPU and GPU counterparts, respectively.

CCS Concepts: • **Hardware → Hardware accelerators**; **Reconfigurable logic applications**.

Additional Key Words and Phrases: FPGA, Graph Neural Networks, Graph Convolutional Networks, Neural Network Quantization, Mixed-Precision Neural Networks, Graph Processing, Network-on-Chip, Hardware Architecture

ACM Reference Format:
Pedro Gimenes, Aaron Zhao, and George Constantinides. 2025. AMPLE: Event-Driven Accelerator for Mixed-Precision Inference of Graph Neural Networks. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3721146.3721963

1 Introduction

Graphs serve as powerful representations for capturing relationships between entities, which are represented as nodes, connected together by edges. This structure enables modeling a wide range of complex systems, including social networks [Alamsyah et al. 2021], biological interactions [Wu et al. 2021], and recommendation systems [Wang et al. 2021]. Graph Neural Networks (GNNs) have emerged as a transformative approach for processing graph data, designed to learn from complex relational information by exploiting the interconnections within the graph [Kipf and Welling 2016; Veličković et al. 2017].

Authors' Contact Information: Pedro Gimenes, pg519@ic.ac.uk, Imperial College London, London, United Kingdom; Aaron Zhao, a.zhao@imperial.ac.uk, Imperial College London, London, United Kingdom; George Constantinides, g.constantinides@imperial.ac.uk, Imperial College London, London, United Kingdom.

Inference on GNN models can be divided into two main computational phases, (1) **Aggregation** and (2) **Transformation** [Gilmer et al. 2017]. In the Aggregation phase, a permutation-invariant function such as summation or mean is applied over the feature embeddings of a node's neighbors. The results of this phase are then utilized in the Transformation phase, which consists of a fully-connected layer used to generate the updated feature embedding for each node. While the Transformation phase presents a *highly regular computational pattern*, which can be effectively accelerated on a parallelized device such as a GPU, the Aggregation phase involves many irregular memory accesses due to the random and sparse nature of typical graph data. Additionally, aggregation latency is a function of a node's degree, which follows a highly non-uniform distribution. As such, an efficiently-designed GNN accelerator needs to alleviate the computational irregularity of the Aggregation phase while leveraging the regularity of the Transformation phase [Yan et al. 2020].

Although CPU memory systems are a mature and highly optimized technology, the sparse structure of graph data renders traditional cache systems less effective, since node aggregation incurs a high number of accesses to non-contiguous memory ranges. Inference on GPUs offers higher performance due to the deep level of parallelism, however, these devices are limited by high-latency memory management mechanisms. Additionally, there is no support for inter-phase pipelining, meaning aggregation results must be stored into off-chip memory before being re-fetched for the transformation phase. Finally, modern devices have limited support for computation with low-precision numerical formats.

These considerations have motivated the design of several GNN accelerators. HyGCN leverages a set of Single Instruction Multiple Data (SIMD) cores for aggregation, and a systolic array for node transformation [Yan et al. 2020]. Meanwhile, GenGNN was proposed as a model-agnostic framework for GNN acceleration, addressing the gap between the development pace of GNN models and custom accelerators [Abi-Karam et al. 2022] through High-Level Synthesis tools. Table 1 summarizes the characteristics of available GNN hardware platforms. Despite the benefits of previously proposed GNN accelerators, (i) the double-buffering mechanism deployed in HyGCN is not well suited for graph computation due to the non-uniform distribution of node degrees. Under this paradigm, low degree nodes must wait for higher degree nodes before computation can proceed, causing a high number of pipeline gaps. This highlights the need for an **event-driven programming flow**, where nodes are independently allocated resources and scheduled onto the accelerator. Additionally, (ii) neither accelerator offers hardware support for model quantization. As observed by Tailor et al. [Tailor

Table 1. Summary of graph processing features across hardware platforms. Although CPU parallelization is possible, multi-threaded CPUs have limited core count compared to parallelized accelerators. Event-driven programming is possible in GPUs, however computation does not follow a node-centric flow. Although pre-fetching is possible in GenGNN, all incoming messages for nodes in flight are required to be stored on-chip. Finally, no existing accelerators support arbitrary multi-precision computation.

Hardware Platform	Parallelization	Event-Driven Programming	Node Pre-Fetching	Multi-Precision
CPU (Intel Xeon)	✗	✓	✗	✗
GPU (RTX A6000)	✓	✓	✗	✗
HyGCN [Yan et al. 2020]	✓	✗	✗	✗
GenGNN [Abi-Karam et al. 2022]	✓	✗	✓	✗
AMPLE	✓	✓	✓	✓

et al. 2020], the accuracy cost of quantization in GNNs is predominantly due to the aggregation phase and directly correlated to a node's degree. As such, casting low-degree nodes to lower-precision formats while preserving high-degree nodes in high precision leads to reduced memory cost and resource usage at a low cost to model accuracy. Finally, (iii) existing accelerators require on-chip buffering of node embeddings for the entire input graph. As such, these have limited applicability for inference on large graphs ($> 100k$ nodes) where embeddings cannot feasibly be stored on-chip, highlighting the need for a **node-centric pre-fetching system** to hide memory access latency while the accelerator is busy.

We address these shortcomings by introducing a novel GNN accelerator, AMPLE, and contribute the following:

- We showcase an event-driven programming model for GNN acceleration, by enabling the host to program nodes asynchronously through memory-mapped registers.
- We propose an architecture featuring a heterogeneous pool of multi-precision Aggregation Cores connected through a Network-on-Chip, which are dynamically allocated to nodes according degree and precision.
- We evaluate AMPLE on large-scale graph datasets ranging from 2K to 700K nodes, achieving an average speedup of 243× and 7.2× compared to CPU and GPU baselines, respectively.

The body of this paper is structured as follows. Section 2 covers background on GNNs and neural network quantization. Section 3 explains the architecture of the AMPLE accelerator, including how each high-level feature is achieved at the circuit level. Finally, Section 4 explains the testing methodology and experimental results against CPU/GPU baselines.

2 Background

2.1 Graph Representation

A graph $G = (\mathcal{V}, \mathcal{E})$ is a set of nodes/vertices $\mathcal{V}$ and edges $\mathcal{E}$. The set of feature representations at layer l is denoted by matrix $X^{(l)} \in \mathcal{R}^{N \times D}$, where $N = |\mathcal{V}|$ is the number of nodes and D is the feature size. An element $e_{i,j} = (v_i, v_j)$ present in $\mathcal{E}$ indicates that there is a connection between nodes v_i and v_j, meaning node v_j is contained in the set of i's neighbors, $\mathcal{N}_i$, and v_i is contained in $\mathcal{N}_j$. In an undirected graph, the edge element $e_{i,j}$ corresponds to $e_{j,i}$. The connections in a graph can be represented using an $N \times N$ adjacency matrix, where each element $A_{i,j}$ represents an edge between nodes i and j.

2.2 Graph Neural Networks (GNNs)

Within a GNN, graph data is transformed over several layers to perform classification and/or regression tasks on the full graph or individual nodes/edges. GNNs can be represented through the Message Passing Mechanism [Gilmer et al. 2017], which generalizes the node update law as follows.

$$\mathbf{x}_i^{l+1} = \gamma(\mathbf{x}_i^l, \mathcal{A}_{j \in \mathcal{N}(i)}(\phi(\mathbf{x}_i^l, \mathbf{x}_j, e_{i,j}^l))) \tag{1}$$

It can be seen that in the general case, each node aggregates incoming messages represented as an arbitrary function ϕ, which is equivalent to aggregating neighboring embeddings when $\phi = \mathbf{x}_j^l$. Messages are aggregated through an arbitrary permutation-invariant aggregation function $\mathcal{A}_{j \in \mathcal{N}(i)}$ over the neighborhood of node i, and and an arbitrary transformation function $\gamma(\mathbf{x}_i^l, \mathbf{m}_i^l)$, where $\mathbf{m}_i^l$ is the result of aggregation (i.e. $\mathbf{m}_i = \mathcal{A}_{j \in \mathcal{N}(i)} \phi(\mathbf{x}_i^l, \mathbf{x}_j^l, e_{i,j}^l)$).

2.2.1 *Graph Convolutional Networks (GCN).*

GCNs emerged as a solution analogous to Convolutional Neural Networks in the computer vision domain [Kipf and Welling 2016]. The element-wise node update law for a single GCN layer is shown in Equation 2.

$$\mathbf{x}_i^{l+1} = W \left(\sum_{j \in \mathcal{N}_i \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j^l \right) \tag{2}$$

The normalization factors are given by $\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$. It can be seen that $\mathcal{A}$ is taken as the summation $\mathcal{A} = \sum_{j \in \mathcal{N}_i} \phi(\mathbf{x}_j, e_{i,j})$, with $\gamma(\mathbf{x}_i, \mathbf{m}_i) = W\mathbf{m}_i$.

2.2.2 *Graph Isomorphism Networks (GIN).*

GIN was proposed as a model that can provably generate distinct feature updates for two graphs that can be shown to be non-isomorphic through the Weisfeiler-Lehman test [Leman 2018], thus maximizing its representational capacity [Xu et al. 2018]. The update law is given by the following, where ϵ is a small scalar for numerical stability.

$$\mathbf{x}_i^{l+1} = MLP \left((1 + \epsilon) \cdot \mathbf{x}_i^l + \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j^l \right) \tag{3}$$

The same aggregation $\mathcal{A}$ is used as in GCN. In contrast to GCN, GIN does not make use of normalization factors in aggregation (i.e. $\phi = \mathbf{x}_j$), and a residual connection is added after aggregation, which is equivalent to a self-connection in the graph's adjacency matrix.

Table 2. Example state of the Node Scoreboard at runtime, with nodeslots are found in various states. The adjacency list and updated feature pointers indicate the memory address from which to fetch the list of neighbouring node IDs and write updated feature embeddings, respectively. The precision field dictates which arithmetic cores are allocated at runtime.

Slot	Node ID	Precision	State	Neighbors	Adjacency List Pointer	Updated Feature Pointer
0	267	float	Transformation	32	0x3BC90188	0x4FE8B774
1	268	float	Aggregation	8	0xCAF5C03F	0xE672109F
…	…	…	…	…	…	…
63	330	int4	Prefetch	1	0x78E26A27	0xA4D89ED9

2.2.3 *GraphSAGE.*

GraphSAGE was proposed as an inductive framework to generate feature embeddings with high representational capacity for unseen nodes and/or sub-graphs [Hamilton et al. 2017].

$$\mathbf{x}_i^{l+1} = W_1 \mathbf{x}_i + W_2 \cdot \left(\underset{j \in \mathcal{N}(i)}{\text{mean}} \, \sigma(W_3 \mathbf{x}_j^l + \mathbf{b}) \right) \tag{4}$$

It can be seen that the message passing function ϕ is taken as a fully-connected layer with activation σ over the neighbouring embeddings $\mathbf{x}_j$, $\mathcal{A}$ is taken as the mean, and the transformation $\gamma(\mathbf{x}_i, \mathbf{m}_i) = W_1 \mathbf{x}_i + W_2 \mathbf{m}_i$ where W_1, W_2 are linear projection matrices. The projection parameterized by W_1 can be seen as a scaled residual connection.

2.3 Neural Network Quantization

Quantization has been widely explored as a method for reducing model complexity and computational latency in neural networks. Quantization-Aware Training (QAT) enables minimizing accuracy loss at low-precision representations by quantizing activations in the forward pass, making use of the Straight-Through Estimator (STE) in the backwards pass to estimate the non-differentiable quantization gradients. In general, activations are quantized following Equation 5, where q_{min}, q_{max} form the chosen range of representable values, s is the scaling factor to place x into the required range, z is the zero-point (floating point equivalent of the value 0 in the quantized space) and the brackets represent the rounding operation.

$$x_q = \min(q_{max}, \max(q_{min}, \left\lfloor \frac{x}{s} + z \right\rceil)) \tag{5}$$

The min and max functions are in place to show that any values beyond the specified range assume the fixed-point value at the limit. Following this, activation can be de-quantized by $\hat{x} = (x_q - z)s$, where $\hat{x}$ is an approximation of the original floating-point value.

2.3.1 *Quantization-Aware Training for Graph Neural Networks.*

Degree-Quant, proposed by Tailor *et al.*, was one of the first approaches applying Quantization-Aware Training to Graph Neural Networks [Tailor et al. 2020]. Firstly, Tailor *et al.* suggest that the aggregation phase of GNN inference is the predominant source of quantization error. This effect is observed more heavily in nodes with higher in-degrees, which can be intuitively understood since the absolute magnitude of aggregation grows with the number of neighbors. The growth in expected aggregation for high-degree nodes affects the q_{max} and q_{min} values, reducing the quantization resolution due to these outliers in the distribution of aggregation results.

The authors of Degree-Quant address the issue of quantization error by stochastically applying a protection mask at each layer following the Bernoulli distribution [Tailor et al. 2020]. Protected nodes operate in floating-point, while non-protected nodes operate in fixed-point. A node's probability of protection is a function of its degree, interpolated within a parametrizable range $[p_{min}, p_{max}]$, where the graph nodes with minimum/maximum neighbor counts are assigned the limit probabilities.

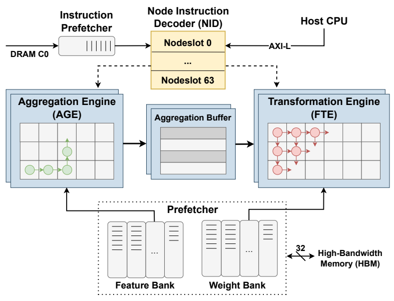

Fig. 1. **AMPLE** Top Level Diagram. Packets propagate through dimension-order routing in the Aggregation Engine's Network-on-Chip (shown in green), and are driven diagonally into the Transformation Engine's systolic array (shown in red). Dashed lines represent control flow interfaces, while solid lines represent data flow between units. Node embeddings are fetched through HBM, while instructions are stored in DRAM.

3 Architecture

As shown in Figure 1, AMPLE is composed of the following functional units, with their respective functions.

- **Node Instruction Decoder (NID)**: communication with the host device and driving other functional units to schedule work onto the accelerator.
- **Prefetcher**: fetching and storing layer weights and neighbouring feature embeddings into local memories (the Weight Bank and Feature Bank, respectively).
- **Aggregation Engine (AGE)**: performing permutation-invariant aggregation functions over all neighbours of a node through a Network-on-Chip architecture.
- **Aggregation Buffer (ABF)**: storage element containing aggregated feature embeddings generated by the AGE.
- **Feature Transformation Engine (FTE)**: computing the updated feature embeddings for each node by performing a matrix multiplication between weights in the Weight Bank and aggregation results in the Aggregation Buffer.

3.1 Event-Driven Programming through the Node Instruction Decoder

Communication between AMPLE and the host is handled by the Node Instruction Decoder (NID), which is a memory-mapped register bank comprised of a configurable number of nodeslots. As shown in Table 2, each nodeslot contains the information required to perform a node's aggregation and transformation steps, and a state machine is maintained indicating each node's state. Algorithm 1 shows how work can be offloaded by the host, which runs concurrently with the accelerator. First, the NID is programmed with a number of global and layer-wise parameters, including node/feature counts and aggregation functions. Subsequently, the host programs the nodeslots and updates values in the mask $available\_nodeslots \in \{0,1\}^n$ where n is the number of nodeslots. While a node is programmed, the accelerator performs aggregation and transformation over previously-programmed nodes. The *available_nodeslots* mask is then deasserted independently by the accelerator when the computation is finished. Thus, the accelerator supports a node-wise, event-driven computation paradigm. Note that $'1$ and $'0$ indicate a mask full of ones and zeros, respectively.

Algorithm 1 Host programming pseudocode

```
Require: global parameters $\mathcal{P}$, layers $\mathcal{L}$, nodes $\mathcal{V}$
  nid_register_bank.global_parameters ← $\mathcal{P}$
  available_nodeslots ← '1
  for layer in $\mathcal{L}$ do
      nid_register_bank.layer_config ← layer
      layer.prefetch_layer_weights()
      while $\mathcal{V} \neq \emptyset$ do
          if available_nodeslots != '0 then
              chosen_nodeslot ← choose(available_nodeslots)
              chosen_nodeslot.programming ← $\mathcal{V}$.pop_head()
              available_nodeslots [chosen_nodeslot] ← 0
          end if
      end while
  end for
```

After a nodeslot is programmed, the NID then drives the Prefetcher, AGE and FTE as detailed in Table 3 to perform the computation, and updates the node's internal state machine after each functional step. No further intervention is required from the host, and an interrupt is sent after step 7 to indicate the nodeslot can be reused. It should be noted that the order in which nodes are programmed within the nodeslots does not imply any priority or time correlation. Typical graph datasets often display high variance in execution time per node, depending on neighbour count and numerical precision. Whenever a nodeslot finishes its computation, it can be immediately reprogrammed by the host with the next node. This event-driven control flow requires the host to run concurrently with the accelerator to monitor its state and drive further work when resources are available. Within the NID, nodes running concurrently are serviced with round-robin arbitration to grant access to shared resources within the Aggregation and Transformation Engines.

Table 3. After nodeslot programming, the NID drives the other units in the accelerator in the sequence shown to perform aggregation and transformation.

	From	To	Dataflow request
1	NID	Prefetcher	Fetch node's adjacency list and neighbouring feature embeddings
2	NID	AGE	Aggregate neighbouring features
3	AGE	ABF	Store aggregation results into the buffer
4	NID	FTE	Multiply aggregation results with layer weights to compute updated features
5	FTE	Prefetcher	Request layer weights stored in the Weight Bank.
6	FTE	DRAM	Store updated features in off-chip memory after transformation is complete.
7	FTE	NID	Response signalling transformation is is complete, triggering the nodeslot to be made available for the next node.

3.2 Mixed-precision Arithmetic

Computing units within the AGE and FTE are locally homogeneous, meaning each processing element supports a single numerical precision. Within the Aggregation Engine, these are arranged in a Network-on-Chip (NoC) architecture comprising a heterogeneous grid of processing elements, where the ratio of PEs allocated to each precision can be configured at compile time according to the application requirements. Each PE is coupled to a router responsible for transferring packets over the network. Each packet is comprised of a head flit carrying routing payloads, an arbitrary number of body flits carrying data, and a tail flit. Since there is no requirement for communication between PEs of different precisions, these are placed within isolated sub-networks as shown in Figure 2, which acts to reduce packet congestion.

As discussed in Section 1, static pipelining through the double buffering mechanism leads to pipeline gaps when computing over graphs with high variance in node degree, since low-degree nodes must wait for high-degree nodes to release resources. This is alleviated in the AGE by dynamically allocating processing elements within each aggregation sub-network according to a node's feature count and precision. As such, nodeslots are allocated resources independently of any other ongoing workload, and these resources can be immediately reused upon completion, thus forming an event-driven programming model.

3.3 Large Graph Processing

Inference over large graphs is enabled by the Feature Bank in the Prefetcher, which contains a storage element named "Fetch Tag" for each nodeslot in the NID. A group containing a parametrizable number of Fetch Tags is coupled to each HBM bank on the Alveo U280 card, meaning up to 32 Fetch Tags can access memory resources concurrently, alleviating the inherent memory boundedness associated with sparse graph data. Within each Fetch Tag group, access to the HBM bank is granted using round robin arbitration.

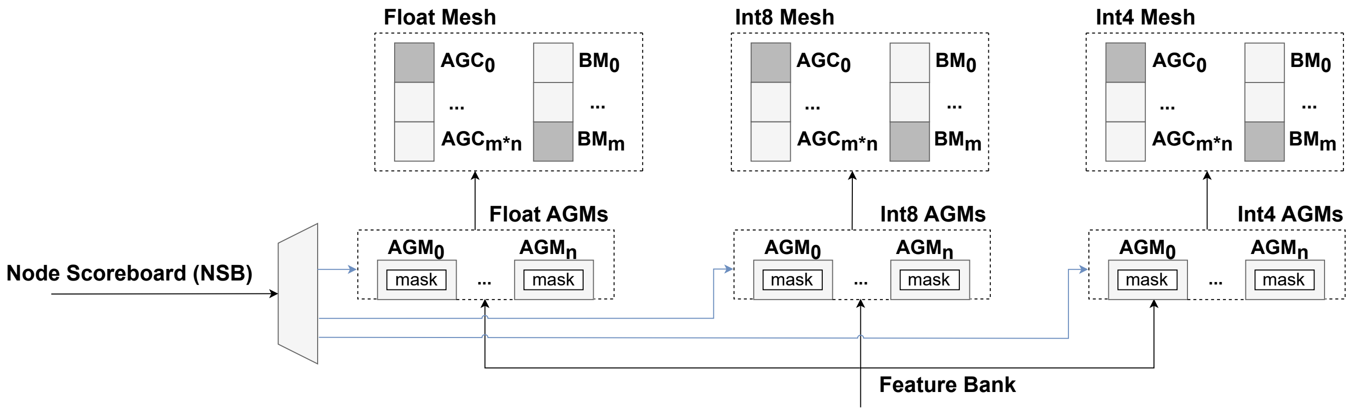

Fig. 2. Microarchitecture of AGE configured with three supported precisions. NID requests drive the Aggregation Managers (AGMs), which receive fetched embeddings from the Feature Bank (See Figure 1). These are then transferred to the Aggregation Cores (AGCs) through the network. Aggregation results are then buffered by the Buffering Managers (BMs).

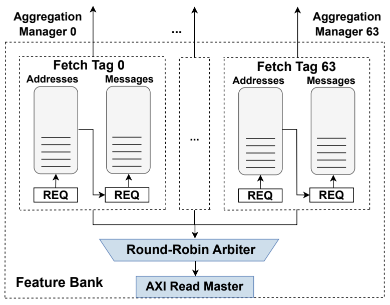

Fig. 3. Fetch Tags in the Feature Bank make concurrent memory access requests in a two-stage process; First, the list of neighbouring node IDs is stored in the Address Queue, and these are then used as pointers for the neighbouring feature embeddings, which are stored in the Message Queue.

The Feature Bank supports the large graph use case via its **partial response mechanism**. For nodes with degree higher than the Fetch Tag capacity, the Fetch Tag fills the Message Queue and directly unblocks the AGE to begin the aggregation process. Once aggregation begins, the Fetch Tag is re-enabled and continues to fetch the remaining neighbours, hiding the memory access latency. This mechanism leads to lower storage requirement per nodeslot, allowing a higher number of Fetch Tags in the Feature Bank, i.e. deeper node parallelism.

4 Experimental Results

Three foundational GNN models were deployed for evaluating the accelerator, with varied architectures, as shown in Table 4. See Section 2 for each model's update laws.

Furthermore, 6 graph datasets were chosen, the first three being small citation networks, and the last three being larger social media graphs. Table 5 shows the node count and mean node degree for each evaluated dataset - the latter acts as an indicator of graph sparsity, with an inverse relationship between sparsity and mean degree.

Table 4. GNN models used for benchmarking the accelerator. A residual connection denotes the addition of the node's original embedding after the aggregation or transformation steps.

Model	Aggregation	Residual	Normalization
GCN	sum	✗	aggregation
GIN	sum	aggregation	✗
GraphSAGE	mean	transformation	transformation

Table 5. Datasets used for benchmarking. DQ ratio shows the ratio of nodes mapped to float precision by the DegreeQuant algorithm, with the rest running in int8.

	Name	Nodes	Mean Degree	Features	DQ Ratio
CR	Cora	2,708	3.9	1,433	2.1 %
CS	CiteSeer	3,327	2.7	3,703	2.7 %
PB	PubMed	19,717	4.5	500	2.9 %
FL	Flickr	89,250	10.0	500	0.2 %
RD	Reddit	232,965	99.6	602	2.7 %
YL	Yelp	716,847	19.5	300	0.4 %

4.1 Mixed-Precision Arithmetic

The DegreeQuant algorithm was used to assign the precision for each node in the graph datasets, by stochastically protecting nodes according to their degree (see Section 2). As shown in Table 5, the ratio of protected nodes is below 3% for all datasets, suggesting a similar ratio of resources on the accelerator should be allocated to float. Configuration parameters were then chosen as follows; given two node groups, for float and int8 nodes, a resource budget $R_p^{max,r}$ (where $p \in$ [float, int8] is the numerical format and $r \in$ [LUT, FF, BRAM, DSP] is the resource type) is allocated to each group using the ratio obtained from DegreeQuant. A single-arithmetic variant was synthesized for float and int8, and the resource utilization per nodeslot C_p^r was estimated for each precision

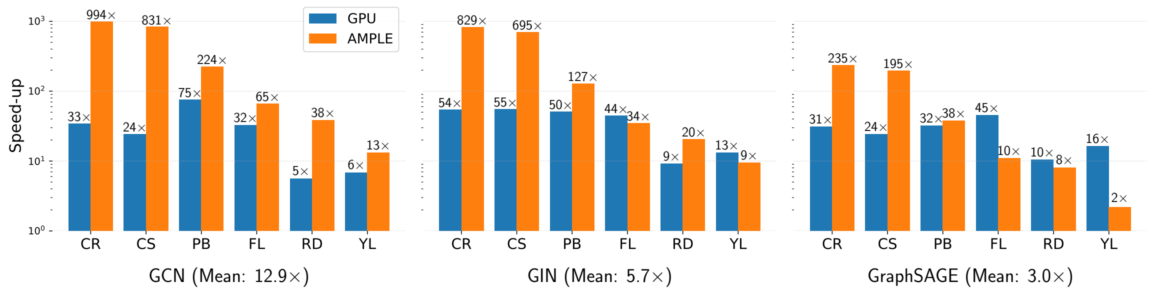

Fig. 4. Inference speedup compared to Intel Xeon CPU baseline obtained on the RTX A6000 GPU and AMPLE simulation. The GPU shows an average speedup of 29.8×, 37.8× and 26.7× across all datasets for GCN, GIN and GraphSAGE, respectively. Equivalent speedups on AMPLE were 361.3×, 285.8× and 81.7×.

and resource type. Finally, the number of nodeslots N_p for each precision is determined as shown in equation 6, where the brackets denote rounding up to the nearest integer.

$$N_p = \left\lceil \min_r \frac{R_p^{max,r}}{C_p^{rt}} \right\rceil \tag{6}$$

It was expected that at lower ratios of protected nodes, resources can be distributed across a higher number of nodeslots, due to the lower resource usage of fixed-point cores. In fact, it was found that allocating a single nodeslot to floating-point is normally enough to meet the precision requirement for task accuracy while maximising hardware node parallelism.

4.2 Performance Analysis

Each model was first benchmarked on the Intel Xeon CPU and RTX A6000 GPU across all datasets, with randomly initialized node features and layer weights. In each case, the mean latency was obtained over 100 trials to account for runtime jitter due to non-deterministic processes. The GPU cache was emptied prior to each prediction step such that latency readings include off-chip memory access for features and weights. GPU warm-up time was not included, meaning inference times are taken after driver initialization is complete. Finally, inference latency on AGILE was obtained from Modelsim 19.2 simulation results at a frequency of 200MHz, obtained for the Alveo U280 card using the Vivado 23.1 toolflow. As shown in Figure 4, AMPLE led to an improvement in mean inference time compared to the CPU/GPU baselines across all models. Table 6 shows the obtained values for latency and node throughput for GCN.

5 Conclusion

This work presented AMPLE, an FPGA accelerator for GNN inference over large graphs. An event-driven programming flow was introduced, coupled with a dynamic resource allocation mechanism through on-chip network communication, overcoming the performance bottleneck associated with node batching in graphs with non-uniform distribution of node degrees. Using a node-centric data prefetcher, we alleviate the requirement for on-chip storage of weights and activations, enabling GNN acceleration over social media graph datasets. These factors led to an average speedup of 243× and 7.2× compared to CPU and GPU baselines. Finally, we provide the first platform to accelerate graphs quantized at node granularity, demonstrating an optimal resource mapping to maximise node parallelism at a low cost to model accuracy.

Table 6. Inference time for evaluated datasets using a single-layer GCN model. Mean latency is reported over 100 iterations.

	CPU (Intel Xeon)		GPU (RTX A6000)		AMPLE @200MHz			
	Mean Latency [ms]	Throughput [nodes/ms]	Mean Latency [ms]	Throughput [nodes/ms]	Mean Latency [ms]	Throughput [nodes/ms]	Latency Gain (CPU)	Latency Gain (GPU)
Cora	244.4	11.1	7.2	376.3	0.246	11,022.0	994.8×	29.3×
CiteSeer	244.3	13.6	10.1	330.0	0.294	11,318.6	831.2×	34.3×
PubMed	362.4	54.4	4.8	4,099.5	1.617	12,193.2	224.1×	3.0×
Flickr	475.4	187.8	14.5	6,146.2	7.227	12,350.0	65.8×	2.0×
Reddit	953.3	244.4	171.0	1,362.0	24.6	9,463.6	38.7×	6.9×
Yelp	760.8	942.2	110.9	6461.6	57.5	12,471.7	13.2×	1.9×
Average	506.8	242.2	53.1	3,129.3	15.2	11,469.9	**361.1×**	**12.9×**

References

Stefan Abi-Karam, Yuqi He, Rishov Sarkar, Lakshmi Sathidevi, Zihang Qiao, and Cong Hao. 2022. GenGNN: A Generic FPGA Framework for Graph Neural Network Acceleration. (1 2022). doi:10.48550/arxiv.2201.08475

Andry Alamsyah, Budi Rahardjo, and Kuspriyanto. 2021. Social Network Analysis Taxonomy Based on Graph Representation. (2 2021). https://arxiv.org/abs/2102.08888v1

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. arXiv:1704.01212 [cs.LG]

William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Neural Information Processing Systems*. https://api.semanticscholar.org/CorpusID:4755450

Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings* (9 2016). doi:10.48550/arxiv.1609.02907

Adrien Leman. 2018. THE REDUCTION OF A GRAPH TO CANONICAL FORM AND THE ALGEBRA WHICH APPEARS THEREIN. https://api.semanticscholar.org/CorpusID:49579538

Shyam A. Tailor, Javier Fernandez-Marques, and Nicholas D. Lane. 2020. Degree-Quant: Quantization-Aware Training for Graph Neural Networks. (8 2020). doi:10.48550/arxiv.2008.05000

Petar Veličković, Arantxa Casanova, Pietro Liò, Guillem Cucurull, Adriana Romero, and Yoshua Bengio. 2017. Graph Attention Networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings* (10 2017). doi:10.48550/arxiv.1710.10903

Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z. Sheng, Mehmet A. Orgun, Longbing Cao, Francesco Ricci, and Philip S. Yu. 2021. Graph Learning based Recommender Systems: A Review. *IJCAI International Joint Conference on Artificial Intelligence* (5 2021), 4644–4652. doi:10.24963/ijcai.2021/630

Hsiang-Yun Wu, Martin Nöllenburg, and Ivan Viola. 2021. Graph Models for Biological Pathway Visualization: State of the Art and Future Challenges. (10 2021). https://arxiv.org/abs/2110.04808v1

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? *ArXiv* abs/1810.00826 (2018). https://api.semanticscholar.org/CorpusID:52895589

Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. *Proceedings - 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA 2020* (1 2020), 15–29. doi:10.48550/arxiv.2001.02514

accepted 11 March 2025