GraphNoC: Graph Neural Networks for Application-Specific FPGA NoC Performance Prediction

Gurshaant Malik
University of Waterloo, Canada
gsmalik@uwaterloo.ca

Nachiket Kapre
University of Waterloo, Canada
nachiket@uwaterloo.ca

***Abstract*—We can democratize design of FPGA Network-on-Chips by replacing slow and expensive conventional NoC benchmarking tools with highly accurate and fast Graph Neural Networks based models. FPGA reconfigurability allows for tuning and designing of NoCs specific to the application being implemented on the FPGA, a facility not afforded to ASIC NoCs. However, such application-specific NoC designs can require thousands of incremental updates and customization to the NoC design, with each resulting NoC configuration needing benchmarking for packet performance to guide the design process. Additionally, each of these benchmark runs can take up to minutes with conventional tools like RTL simulation for modest packet trace lengths. As a result, tuning and design of a NoC even for a single FPGA application can last up to days, presenting a critical bottleneck to developer efficiency and iteration speed. We address this by presenting a framework to encode any FPGA NoC and any FPGA application traffic into graphs, called GraphNoC. We create a dataset of these graphs, comprising of different FPGA NoCs and applications. We use this dataset to train GNNs, including foundation models, to predict NoC routing latencies that can accelerate benchmarking run-times by up to 148× (506× using GPU) with prediction top-20 accuracies up to 97.2%. We also show these GNNs can accelerate end-to-end FPGA application-specific NoC design by up to 4.3× (37× using GPU) while regressing final NoC latency by only 30 cycles.**

I. Introduction

Design of Network-on-Chips (NoCs) has become a critical component in the success of increasingly complex applications on FPGAs and ASICs. With the ending of Moore's Law [1], application implementations have seen a shift towards progressively modular sub-systems [2], [3] and exotic data flow patterns [4]–[6]. NoCs have risen in prominence as a scalable and resource-efficient interconnect to meet an application's bandwidth requirement by supporting the intense and non-deterministic movement of data packets between these sub-systems. NoCs now find a prominent place in ASIC and FPGA designs running State-of-the-Art applications around Machine Learning [7], Neuromorphics [8], [9], Autonomous Driving [10]. This means that design of NoCs to meet these varying workload patterns is now an important engineering challenge.

FPGA NoCs provide a shared interconnect for communication between endpoints of the application being implemented on the programmable logic of the FPGA. Unlike ASIC NoCs, FPGAs' reconfigurability can be leveraged to design and tune low-cost and highly performant NoCs specific to the application being implemented. However, design of these application-specific FPGA NoCs will be significantly more time consuming than traditional ASIC NoCs as we must choose various NoC parameters [11], [12] such as switches, buffer depths, regulator rates, to best match them for the FPGA application being implemented. Design and performance optimization of these NoC parameters can take days of simulation and exploration, with each exploration step requiring computationally expensive FPGA RTL simulations lasting tens of minutes [13] for even modest packet traces. In a quest to remove this bottleneck in the design of application-specific FPGA NoCs and super-charge the harnessing of FPGA reconfigurability, *quick* and *accurate* performance benchmarking of NoCs can play a key role by significantly reducing these expensive time and computational investments.

Current work around alternatives to RTL simulations for NoC performance prediction can be broadly grouped into 2 categories: 1) Analytical models and 2) Machine Learning models. Analytical models are tightly coupled to the specific FPGA NoC's architecture and cannot be easily applied to other NoCs. For example, the HopliteBuf analysis tool [11], [12], [14] only works for FIFO and backpressure based Hoplite routers. NoC designs like the Backpressure based Butterfly Fat Trees (BFTs) [15] and HopliteRT [16] provide a provable upper bound on worst case FPGA routing latencies. But these bounds are exclusively a function of the NoC architecture only and not the application being routed. On the other hand, machine learning based models, enabled by the capacity of deep neural networks to learn non-linear vector mappings, can generalise better to a wide variety of NoC architecture and applications. However most of these models are based on conventional neural network layers like MLPs and Convolutions. Thus, they can only ingest highly structured euclidean data [17], [18], in turn limiting their application to only highly regular NoC topologies like meshes and tori.

To conclude, RTL simulations are accurate and generalisable but slow and expensive. Analytical models are faster but tied to a specific NoC design and do not usually take FPGA applications into account. Conventional machine learning based techniques are fastest but not generalisable to NoC architectures with irregular topologies. In this paper, we aim to democratize and accelerate design of application-specific FPGA NoCs by introducing a Graph Neural Network (GNN) based NoC benchmarking framework that combines the best of RTL simulations, analytical models and conventional machine learning based models into a single design: 1) Highly accurate

NoC performance estimates, 2) Support wide variety of NoC architectures and topologies, and 3) Fast run-time. Our framework can encode any FPGA NoC (regardless of its topology or architecture) and any FPGA application's traffic into graphs and leverage GNN [19] based models to *accurately* and *quickly* predict the NoC's routing latencies for that particular FPGA application. The key contributions of our work include:

- Framework to encode any FPGA NoC (regardless of topology/architecture) and any FPGA application into graphs, compatible with both soft [14], [15] and hard NoCs [20].
- Creating a dataset to train GNN models, including multi-modal foundation models, to predict performance of FPGA NoCs for a given FPGA application. It consists of over 1.5 million samples of FPGA NoCs (HopliteBuf/BP, BFT) and their ground-truth performance on over 200 synthetic and real-world FPGA applications.
- Quantification of trained GNNs on FPGA NoCs: Up to **97.2% top-20 accuracy** for HopliteBuf/BP NoCs while being up to ≈**148**× (≈**506**× **on GPU**) faster than RTL simulations for BFT NoCs. As a result of this quick and accurate NoC benchmarking, application-specific FPGA NoC design times are reduced by up to ≈**4.3**× (≈**37**× **on GPU**).
- Open-sourcing the entire work, including dataset and trained GNN models, enabling the FPGA community to leverage and build on top of our research for their own FPGA NoCs. Source code: https://git.uwaterloo.ca/watcag-public/graphnoc

II. Background

Unlike ASICs, FPGA reconfigurability facilitates design and tuning of NoCs specific to the application. However, as discussed above, iterating on the NoC design comes at a cost; RTL simulations for performance analysis can be a bottleneck. In this section, we first review existing alternatives to RTL simulations. Based on this review (summarised in Table I), we motivate the need of Graph Neural Networks for fast and accurate performance benchmarking of FPGA NoCs.

A. Limitations of Conventional Alternatives to RTL Simulation

1. **Analytical Models**: They estimate NoC performance by building a mathematical representation of a specific NoC's architecture into its formulation. FPGA NoCs like HopliteRT [16] and Flow Control BFTs of [15] can guarantee an upper bound on the worst case routing latency as a function of the NoC architecture itself. However, these performance numbers are a function of the NoC architecture only and do not account for the application being routed. Analytical models of the HopliteBuf/BP suite of [12], [14] make use of network calculus to build application aware models to predict NoC performance, routability and size of statically allocated FIFO buffers. However, while the HopliteBuf/BP suite is application aware, it can only be applied to estimate performance of overlays built using the HopliteBuf/BP switches connected in a directed torus topology. Such analytical models are often hyper-sensitive to even minor perturbations to the NoC architecture, leading to poor generalization capabilities and preventing them from being used for any other FPGA NoC.

2. **Conventional Neural Networks**: Designed by stacking a large number of linear and non linear layers, they are trained using supervised learning. The training data is a tuple of: 1) Input FPGA NoC and FPGA application traffic, and 2) Ground truth performance numbers extracted from actual simulation. This bestows said models with powerful capacity to represent increasingly abstract concepts, foregoing the need for low-level NoC details to be encoded into the input; these details can be learnt as part of the training loop. We see commercial NoC IP vendors offering machine learning based models to predict performance of their NoC products [21]. [18] leverages Artificial Neural Networks (ANNs) to achieve a 1500–2000× speedup over cycle-accurate Booksim NoC simulator [22] with a predictor error of 5–8% for mesh and torus shaped NoCs. However, conventional neural networks, regardless of prediction capacity, exhibit high degree of rigidity in their structure. They can only ingest inputs with a specific shape and thus are incompatible with the rest. This limits only *highly regular* and *symmetric* NoC topologies (mesh, tori etc) to be compatible with an ANN model in a straightforward implementation. Other non-euclidean topologies, while technically possible, will require inefficient and obtuse tensor manipulation for seamless composition into inputs. This introduces a strong inductive bias, which leads to poor generalization to different NoC topologies. We present evidence for this claim shortly in Section II-C.

	NoC Archs.	Irregular Topologies	Time to Predict
RTL Simulation	**Adaptable**	**Compatible**	Tens of minutes
Analytical Models	Fixed	**Compatible**	Seconds to Minutes
Conventional NNs	**Adaptable**	Incompatible	**Sub-Seconds**
GNNs	**Adaptable**	**Compatible**	**Sub-Seconds**

TABLE I: Comparing alternatives to RTL based simulations.

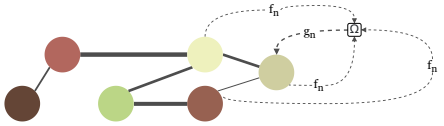


Fig. 1: Foundation of Message Passing GNNs.

B. Motivating Graph Neural Networks for NoC benchmarking

To summarise our discussions in Table I, RTL simulations are accurate but slow. Analytical models are faster but tightly coupled to a specific NoC. Conventional NNs are quick and generalisable, but only for NoCs with regular topologies.

Graph Neural Networks (GNNs) work directly on graphs. A graph consists of node attributes, edge attributes and an adjacency matrix specifying graph connectivity. A graph neural network is composed of a series of differentiable functions and a message passing scheme [23]. While different types of graph neural networks [24], [25] apply different combinations of differentiable functions and message aggregation schemes, the general mechanics of a layer of graph neural networks (shown in Figure 1) are similar and can be described as:

$$h_i = g_n(\Omega_{j \in N_i}(f_n(h_i), f_n(h_j))) \quad (1)$$

where h_i are the attributes of the node i. h_j are the attributes of the neighbor node $j \in N_i$. f_n and g_n are neural network

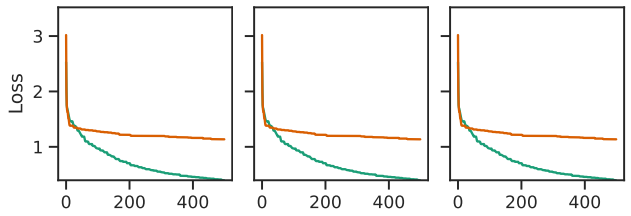
operations. Ω is the message aggregation operator (like sum, max, min etc).

NoCs can be represented with high fidelity as graphs; by encoding NoC topologies into the adjacency matrix while NoC architecture (switches, flow control etc) can be encoded into the graph's nodes and edges. Since graphs are heterogeneous (different types of nodes and edges within same graph), the FPGA application can also be richly represented using a different set of node and edge attributes within the same graph. Finally, a Graph Neural Network based model (GNN) can ingest these graphs as inputs, irrespective of graph connectivity and size. These qualities make GNNs an ideal candidate to build and train models for NoC performance benchmarking.

C. Motivating Example: GNNs vs Conventional NN Models

We now motivate the use of GNNs over conventional neural networks by comparing their composability with non-euclidean NoC topologies. We setup an experiment to compare their ability to predict worst case routing latency for 2 FPGA NoCs with extremely different topologies: Torus shaped 4×4 HoptliteBuf/BP [12] and Tree shaped BFT0 [15] with 8 PEs. We construct 2 models with the same number of parameters: 1) MLP based conventional ANN with 9 layers and 1024 embeddings, and 2) 10 layer GraphSage [26] GNN with 256 embeddings. We train both networks to predict NoC performance for the same set of FPGA applications.

While encoding these NoCs as inputs, we encounter ANNs' incompatibility (discussed above in section II-A) with irregular NoC topologies: an MLP based network can only ingest a vectorized tensor. As a result, we are forced to unroll and flatten both NoCs' switches and traffic traces into a 2D tensor of shape ($num_{switches}$ + num_{traces}, inp_{size}). This results in significant loss of information; the NoC's specific topology is homogenised and hence not captured in the inputs. In contrast, the GNN model can ingest the NoC topology, encoded as graphs, without any loss of connectivity related information. Hence, GNNs can use message passing [23] (Figure 1, Equation 1) to leverage the graph's connectivity in extracting higher order features more effectively.



(a) Train (b) Validation (c) Test

Fig. 2: Loss (**Y axis**) over epochs (**X axis**) for **ANN** vs **GNN**

We observe these effects reflected in the quality of training in Figure 2. Despite training both the models with the same training recipe (500,000 steps, 128 batch size, 0.1 dropout), we see that **ANN's** loss plateaus at 1.13 whereas **GNN's** loss continues to improve down to 0.39, a ≈2.9× improvement. Based on these observations, we conclude that GNN based models are an attractive proposition to accurately predict routing performance in sub-second run-times for a wide variety of FPGA NoC architectures and topologies.

GNNs have not been leveraged to benchmark FPGA NoCs, with previous related works [13] very limited in scope: 1) Focusing on a single ASIC NoC only which, unlike FPGA NoCs, has fixed switch architecture, routing and flow-control, 2) Single-channel only application support, and 3) Not open-sourced. We now summarise a list of key features (and our objectives) of a GNN based NoC performance predictor:

1) **NoC Diversity**: The graph encoding framework is compatible with any FPGA NoC architecture/topology, by encoding diverse NoC parameters (switches, flow-control, topology etc) into richly detailed graphs.
2) **Application Traffic Expressibility**: In addition to encoding NoCs into input graphs, any FPGA application can be encoded into a graph with rich detail.
3) **Fast and Accurate GNN Models**: The trained GNN models accurately predict NoC performance while significantly accelerating benchmarking run-time compared to existing techniques (RTL simulation, analytical models etc).
4) **Open to FPGA Community**: The framework, dataset and models are open-sourced for the community to leverage for their own use-case and make improving contributions.

III. The GNN based GraphNoC framework

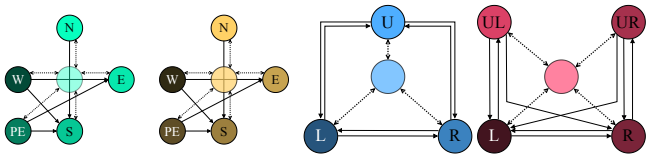
In this section, for a given FPGA NoC and the traffic generated by the FPGA application's endpoints, we present the 2 key building blocks to using GNNs to predict NoC performance for that application: 1) Encoding the given NoC and application into graphs, and 2) Designing a GNN that ingests this graph and predicts NoC performance.

A. Encoding NoC and Application into Graphs

To use GNNs, we must first encode the given FPGA NoC and FPGA application into graphs. Our encoding framework is designed from the ground-up to support any NoC and application, independent of architecture and topology, and offers precise control over graph contents. We do this by leveraging the principles of heterogeneous graphs [27] in a *layered*, *hierarchical* fashion:

- *Layer 1*: Encoding NoC switches into graphs (Figure 3).
- *Layer 2*: Encoding NoC topology into a graph by using layer 1 as building blocks (Figure 4).
- *Layer 3*: Encoding application's traffic into graphs, and overlaying them onto layer 2's topology graph (Figure 4).

1) Layer 1: Encoding FPGA NoC's Switches into Graphs: Note that an FPGA NoC can be composed of a mixture of switch types. For example, the BFT-1, BFT-2 topologies of [15] are composed of a mix of `pi` and `t` switches. Further, the mixing of different switches is sometimes central to designing highly performant NoCs specifically for the FPGA application [12]. Hence, for every switch (even of different types) of the NoC, capturing its fundamental properties (physical dimensions, arbitration, flow control etc) is essential.



(a) HopliteBP (b) HopliteBuf (c) BFT-T (d) BFT-Pi

Fig. 3: Hoplite [12] and BFT [15] switches encoded as graphs.

A graph can be composed of nodes and edges of different types. To encode a NoC switch as a graph, we use 2 types of nodes. The `port` nodes capture the IO ports of the switch. Figure 3 shows a switch's ports (with text overlays) encoded as `port` nodes, with different shades of the same color capturing each port's different properties (numerically represented as tensors). The `central` node of a switch sub-graph is used to represent the architectural properties of the switch (flow-control etc), as shown in the translucent nodes at the center of each graph in Figure 3. The `port` nodes are connected to each other via the directed `route` edge (——→) to capture the available routing data-paths between ports. Finally, the bidirectional `switch` edge (←·····→) connects the `central` and `port` nodes.

2) Layer 2: Encoding NoC Topology into Graphs: We now present the rules around connecting the NoC switches (layer 1 graphs) to encode the NoC topology into a graph. We introduce a new directed edge type called `topology` that connects together the ports of different switches to reflect the NoC's topology. For example, in Figure 4, we connect the different `port` nodes of HopliteBP (Figure 3a) and HopliteBuf (Figure 3b) graphs with the `topology` edge (⟶) to encode the hybrid HopliteBuf/BP NoC topology. Additionally, the directed nature of this edge type implicitly captures the direction of data-flow between switches of the NoC. Finally, note that the graph's recursive (*i.e* topology graph made of switch sub-graphs) and heterogeneous (*i.e* graph made of different node and edge types) properties allows us to encode any NoC topology into a graph. This is key to GraphNoC supporting any FPGA NoCs, even hard FPGA NoCs like Versal [20], irrespective of topology and architecture.

3) Layer 3: Encoding Application Traffic into Graphs: We now present the methodology to encode an FPGA application's traffic into a graph, which is then overlayed onto the encoded NoC topology graph (layer 2, in turn built using layer 1). This final graph is then ingested as input by our GNN model.

An FPGA application's traffic is a collection of individual traffic traces between 2 NoC endpoints. We aim to encode any number of traffic traces between 2 endpoints. This allows for an accurate expression of application behaviour and better scalability; any new traffic traces can simply be added to the application graph without modification. Each traffic trace is represented as a `traffic` type node, with its numerical value capturing attributes like injection rate etc. A new directed edge type `application_traffic` connects the source NoC port (entry point of traffic) to the `traffic` node. Then, another `application_traffic` edge connects this `traffic` node to the destination NoC port (exit point of traffic). For example, in Figure 4, the PE ports of switch (1,0) send traffic to the PE ports of switch (1,1) using the `application` node and `application_traffic` edge (●,⟶). Having encoded the application traffic traces into graphs, they can be overlayed on the layer 2 NoC topology graph to complete a (NoC, application) pair's encoding into a heterogeneous graph, as shown in Figure 4.

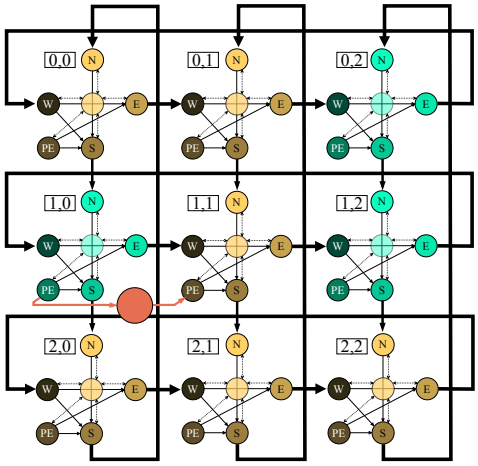


Fig. 4: Encoding a 3×3 Hoplite NoC and application being routed. Notice the different edge types: port routes (——→), switch (←·····→), topology (⟶) and application (⟶).

B. Message Passing Graph Neural Networks

Having encoded the FPGA NoC and the FPGA application's traffic into a single graph, we now present the GNN based model used to predict the performance of that NoC for that application. As shown in Figure 5, our GNN based model is composed of 3 sequential stages of transformations:

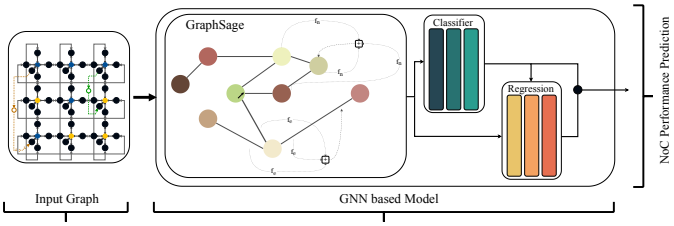


Fig. 5: Design of GNN models: GraphSage, followed by residual application of classification and regression MLPs.

1) *GraphSage GNN Module* [26]: As discussed, our input graph is heterogeneous and is composed of 3 different node types (`port`, `central`, `traffic`) and edge types (`route`, `switch`, `application_traffic`). For each node x_i and its neighbor nodes x_j over an edge type, the GraphSage layer implements the in-place operation of Equation 2, where W_1, W_2 are the learnt weights. The GraphSage module is made up of a sequence of such layers, with the exact recipe detailed later in section V.

$$x_i^{'} = W_1.x_i + W_2.mean.x_j, x_j \in \xi_{x_i}, x_i \in X \quad (2)$$

2) *Classification Module*: Our GNN model predicts performance as a function of classifier and regression (see Figure

5) modules as shown in Equation 3. The classifier predicts class c, whose cardinality C in Equation 4 is determined by the resolution of the upper limit on performance predictions. The classifier is composed of 3 `Linear` layers, with the first 2 gated with the `ReLU` non-linear activation and layer normalization. The classifier is trained with the `CrossEntropy` loss function.

3) *Regression Module*: The regression module predicts a value between [0, res) to complete the performance prediction of Equation 3. To increase it's predictive capacity, we add a residual connection [28] to ingest outputs from both GraphSage and classification modules. The regression module is composed of 4 `Linear` layers, with the first 3 gated with the `ReLU` non-linear activation and layer normalization. The regression module is trained with the `Huber` loss function.

$$perf_{app}^{NoC} = res * c + reg, c \in C \tag{3}$$

$$|C| = \frac{limit}{res} \tag{4}$$

IV. Constructing the GraphNoC Dataset

FPGA NoCs and applications are encoded as graphs in section III-A, which are then ingested as inputs by GNN models of section III-B to predict any NoC performance metric (latency, throughput, feasibility etc) for that application. We now lay the foundations for a community-driven dataset, consisting of a diverse set of encoded graphs of FPGA NoCs and applications, that can be used to train GNN models to quickly and accurately predict NoC routing performance.

Algorithm 1: Framework to add data to GraphNoC

```
Result: train set, val set, test set
NoCTool = RTL.Sim/Analytical.Model;
for var_NoC in NoC.variations do
    for app in Application.collections do
        data = [];
        for rate in Rate.range do
            perf = NoCTool.calculate(var_NoC, app, rate);
            graph = encode(var_NoC, app, rate, perf);
            dataset.add(graph);
        shuffle(data);
        train.add(data[:train_split]);
        val.add(data[train_split:train_split+val_split]);
        test.add(data[train_split+val_split:]);
```

We now present the steps for adding encoded graphs to the dataset in a NoC and application agnostic manner in Algorithm 1. We first initialise the NoC tool to calculate performance. The tool can be RTL simulation or an analytical model (NoC specific availability; section II-A), depending on use-case. We loop over all the different variations of the baseline NoC (eg. mix of Buf/BP switches in HopliteBuf/BP [12]). For a particular NoC variation var_{NoC}, we consider a range of synthetic and real-world [29], [30] FPGA applications. For all combinations of NoC (var_{NoC}) and application (app) pairs, we evaluate ground truth performance using the NoC tool over a range of injection rates and encode each pair as a graph. Finally, this set of graphs is split between train, validation and test sets.

We position the GraphNoC dataset as an open-sourced initiative that the FPGA community can leverage and contribute to. This will allow the FPGA community to train increasingly sophisticated GNN models over time, which will in-turn enable them to explore design space of their own FPGA NoCs even faster. To prove GraphNoC's flexibility with any FPGA NoC and application, we seed the dataset with 3 different FPGA NoCs, over 200 different FPGA applications and both RTL simulation and analytical model as ground truth sources for latency prediction. See Table II for details.

	`HopliteBuf/BP`	`BFT-TREE`	`BFT-XBAR`
`Applications`	14 real-world, 2 synthetic	100 synthetic	100 synthetic
`NoC Variations`	500	BFT0	BFT3
`Rate (incr=1)`	[0.1, 25]	[1, 100]	[1, 100]
`NoC Sizes`	3x3, 4x4, 5x5, 6x6, 7x7	4, 8, 16, 32	4, 8, 16, 32
`Ground Truth`	Analytical Model [12]	RTL Sim.	RTL Sim.

TABLE II: Seeding GraphNoC dataset with 3 different NoCs.

V. Experimental Setup and Training Recipes

We author 3 GNN models of varying complexity using the `PyTorch Geometric` [31], [32] library: small (GNN_s; 1 million), medium (GNN_m; 2.9 million) and large (GNN_l; 7.7 million). We detail authoring and training recipes in Table III. Each model is trained and run on consumer grade hardware with Nvidia 4060 TI GPU, Intel i5-13500 CPU, 32GB RAM. We use `EarlyStopping` [33] on GraphNoC's dataset to predict worst case routing latencies under 1000 cycles. GNN's tensor based computation is broadcast across multiple dimensions, called batch size, allowing multiple NoC samples to be evaluated at once. To ensure fair speedup calculations, we also parallelize conventional RTL simulations and analytical models by wrapping each call in a `Process` module of python's `Multiprocessing` library.

`Model`	≈`Num. Params.`	`Num. Layers`	`Emb. Size`	`Batch Size`	`Rate`	`Schedule`	`Dropout`
GNN_s	1 Million	20	64	128	$5 * 10^{-7}$	Hold	0.2
GNN_m	2.9 Million	15	128	64	$5 * 10^{-7}$	Hold	0.2
GNN_l	7.7 Million	10	256	64	$5 * 10^{-7}$	Hold	0.2

TABLE III: Authoring and training recipes of GNN models.

We design experiments to evaluate GraphNoC's encoding framework, dataset and trained GNN models for speed, accuracy and impact on accelerating the designing and tuning of application-specific FPGA NoCs. We also train a foundation GNN model to test GraphNoC's multi-modal abilities to predict performance for completely different NoCs. See Figure 5 to refresh on the main learning task for a GNN based model, along with its inputs and outputs.

VI. Evaluation

A. Measuring Accuracy of GraphNoC's GNN Models

Worst case latency is critical in defining service level agreements and guarantying reliable operations of NoCs under all

	Train Set								Validation Set								Test Set							
	Top				Delta				Top				Delta				Top				Delta			
Type	20	15	10	5	20	15	10	5	20	15	10	5	20	15	10	5	20	15	10	5	20	15	10	5
GNN_s	90.2	86.6	79.6	61.9	80.7	77.7	72.5	60.2	90.8	87.6	81.2	63.6	85.2	82.7	77.9	66.0	89.6	85.6	78.3	60.5	76.8	73.6	68.2	55.7
GNN_m	95.5	93.1	88.2	73.6	86.4	83.9	79.5	68.4	96.2	94.6	91.2	80.6	90.6	89.0	86.0	77.0	95.6	93.4	89.0	77.2	84.0	81.4	77.3	67.0
GNN_l	97.0	95.4	91.9	81.3	88.8	86.6	83.0	73.9	97.8	96.5	94.0	86.5	92.5	91.1	88.6	82.2	97.2	95.4	92.0	83.0	86.5	84.2	80.4	72.2

TABLE IV: Measuring accuracy of differently sized GNNs against GraphNoC dataset for HopliteBuf/BP hybrid NoCs.

operating conditions. In this experiment, we measure GraphNoC GNNs' accuracy in predicting worst case routing latency for a given FPGA NoC and FPGA application. We setup this experiment by training GNNs to predict worst case latencies for different sizes (3×3 to 7×7) of hybrid HopliteBuf/BP [12] NoCs across a range of real and synthetic FPGA applications (Table II). Hoplite's analytical models generate stricter bounds on latency than RTL simulations [14]. Hence, we measure the accuracy of GNN predictions against ground truth of the analytical model for the test subset of GraphNoC dataset. We study the accuracy through 2 different lenses:

- *Precision* (`Delta-K`): Precision is a measure of *absolute* difference between the ground truth and GNN predictions, important for measuring accuracy for routing latencies with small magnitudes. We measure precision with the `Delta-k` metric, defined as the number of predictions within K absolute cycles of the ground truth. For example, for a ground truth latency of 80 cycles, the GNN prediction would have to lie in [75, 85] to be counted for `Delta-5` precision. We observe that GraphNoC's GNNs can achieve `Delta-5,10,15,20` as high as 72.2%, 80.4%, 84.2%, 86.5% for the test set in Table IV (right most column). Additionally, as we increase GNN size from small to large, the increase in `Delta-20` is only 12.6% whereas the increase is much higher for `Delta-5` at 29.6%, suggesting that larger GNNs can better address high precision use-cases.
- *Scale* (`Top-K`): Scale is a measure of *relative* difference between ground truth and GNN predictions. Unlike precision, scale takes into account the order of magnitude of ground truth latencies. We measure scale with the `Top-K` metric, defined as the number of predictions within K% points of the ground truth. For example, for a ground truth latency of 300 cycles, the GNN predictions would have to lie in [285,315] to be counted for `Top-5`. As shown in Table IV, we note that trained GNN models can achieve `Top-5,10,15,20` as high as 83%, 92%, 95.4%, 97.2%. Like precision, as we move from small to large GNNs, scaling benefits from big increases for `Top-5` (37.1%) but not for `Top-20` (8%).

To summarise this experiment, GraphNoC's GNN models can achieve highly accurate predictions for NoC performance. We observe high accuracies across different NoC architectures, NoC sizes and FPGA applications. These accurate predictions are maintained across different latency scales with high precision, achieving up to 97.2% `Top-20` and 86.5% `Delta-20` metrics. Thus, despite analytical models generating the strictest of upper bounds on latency, GNN can be used in their place for designing/testing FPGA NoCs. We further analyze distribution of GNN predictions and measure outlier deviations vs ground truth in experiment VI-D.

B. Benchmarking Throughput of GraphNoC's GNN Models

In the previous experiment, we established that GNNs can accurately predict NoC routing latencies. In this experiment, we investigate GNNs' prediction speed; to establish if the minimal accuracy loss can be tolerated in exchange for significant speedups over using conventional RTL simulations or analytical models. Since Hoplite's analytical models are significantly faster than RTL simulations (Table I), we stress test the potential of GraphNoC's GNNs to outperform the already fast analytical models instead. In this experiment, we measure maximum throughput (input samples per second) achieved by GNN models (on both CPU and GPU) and compare it to the analytical model. We vary batch size from 1 to 256 (refer to section V implementation details), for different HopliteBuf/BP variations and sizes in GraphNoC's dataset. We measure throughput for 2 different computation scenarios:

- *Constrained Compute*: One of the benefits of using GNNs is to accelerate NoC benchmarking and application-specific design for compute constrained organizations. Given a maximum batch size (a function of end-user's compute capabilities), we investigate the speedups GNNs offer over analytical model in Figure 6. We observe that, given a batch size, GNNs can help increase throughput by up to ≈66× using GPUs and ≈9× using CPUs. We also note that GNNs benefit from increased utilization of compute resources with larger batch sizes (x axis). For example, speedups over analytical model increase from ≈1× (batch size=1) to ≈14.5× (batch size=64) in Figure 6e. However, unilaterally increasing batch size hurts GNNs' throughput, with the affect more pronounced for larger GNNs or NoC sizes; a clear sign of saturation of available compute. For example, as we increase NoC size (Figures 6a → 6e), we observe that GNNs see a drop in their throughput beyond a certain batch size, with this affect being observed at increasingly smaller batch sizes for larger GNNs.
- *Unconstrained Compute*: We now analyze if GNNs also maintain their competitiveness when compute is not a bottleneck. To answer, we consolidate the results of Figure 6 by selecting highest achieved throughput per NoC size, and present them in Table V. We observe that GNNs of all sizes: small, medium and large outperform Hoplite's analytical model by ≈9×, ≈5.8× and ≈3.5× respectively when running on CPUs. When the GNNs make use of the GPU, this gap further increases to ≈66×, ≈55× and

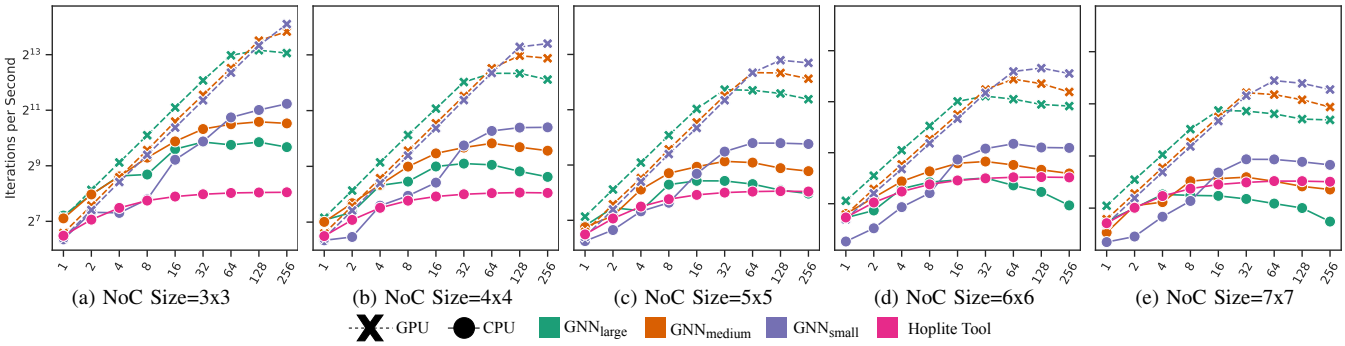


Fig. 6: Throughput (**Y axis**, log_2 scale) as a function of batch size (**X axis**) for GNNs and conventional tools.

≈34× respectively. We also note that the competitive edge of GNNs increases as we move from larger NoC sizes (7×7; ≈1.8× CPU, ≈14.5× GPU) to smaller NoC sizes (3×3; ≈9× CPU, ≈66× GPU), primarily explained by the analytical model's static setup time and flow-dependency bottlenecks overshadowing any size related throughput gains (hovering around ≈260).

To summarise this experiment, GraphNoC's GNNs can significantly accelerate FPGA NoCs' performance benchmarking for a given FPGA application. This speedup over the already fast analytical model is maintained for all GNNs over all NoC sizes. These speedups further increase by an order of magnitude when GNNs leverage the GPU. This is evidenced by GNNs outperforming conventional tools by up to ≈9× on CPUs and ≈66× on GPUs. The high accuracy (section VI-A), combined with these significant speedups, presents a potential opportunity for GraphNoC to significantly accelerate application-specific NoC design/tuning. We investigate this possibility in the next experiment.

	Iterations per Second				
Type	N=3x3	N=4x4	N=5x5	N=6x6	N=7x7
Hoplite Tool	264	266	266	263	262
$GNN_{s,CPU}$	2404	1351	887	652	468
$GNN_{m,CPU}$	1533	907	561	402	291
$GNN_{l,CPU}$	926	548	344	257	184
$GNN_{s,GPU}$	17585	10830	7093	5100	3801
$GNN_{m,GPU}$	14575	8026	5192	3773	2751
$GNN_{l,GPU}$	9157	5181	3388	2382	1717

TABLE V: Best achieved throughput, across all batch sizes.

C. Application-Specific FPGA NoC's Design with GraphNoC

[11], [12]'s application-specific design of the FPGA HopliteBuf/BP NoC exposes the NoC's switches as a categorical choice between HopliteBuf or HopliteBP. For a given FPGA application, this application-specific design results in an optimized NoC with the minimum possible routing latency for this application. The design runs for 100s of epochs. At each epoch, 100s of HopliteBuf/BP's variations are generated and all of them are benchmarked for routing performance on the given FPGA application. At the end of each epoch, the best performing variants are used to seed the base configuration for the next epoch's variants. In the original work, Hoplite's conventional analytical model is used to benchmark all NoC variants. In our experiment, we swap the analytical model with GraphNoC's trained GNN models. We investigate:

1) *Speed*: If GraphNoC's GNN can significantly accelerate the run-time of application-specific NoC design for FPGAs.
2) *NoC Quality*: If the slight inaccuracy of GNNs' latency predictions (compared to the analytical model; section VI-A), accumulating over multiple epochs, significantly deteriorates the final NoC's performance.

To setup this experiment, we target learning switch configuration for a 4×4 HopliteBuf/BP NoC for `random` and `local` FPGA applications' traffic. For each FPGA application, we vary the injection rates' from 0.01 to 0.09. We generate 128 NoC variants in each epoch and run the design process for 100 epochs. The batch size for both GNNs and analytical model is set to 128. We run GNNs on both CPU and GPU. Summarised in Figure 7, we investigate 2 different use cases and compare the final results to the original work:

- *GNNs Only*: Here, we calculate all the NoC variants' (128 variants per epoch, 100 epochs total) latency using GNNs only. At the end of the design process, we compare the resulting NoC design's performance (Figures 7a–7c) against the NoC design generated using original work's analytical model (Figure 7d). We observe, that GNN driven NoC design regresses latencies by up to 19, 24 and 30 cycles, while accelerating end-to-end runtime by up to ≈4.3×, ≈2.6× and ≈1.4× for small, medium and large GNNs on CPUs respectively. The speedup increases to ≈37×, ≈30× and ≈20× when GNNs use the GPU.
- *GNNs+Analytical Model*: Here, we invest the time saved by GNN driven runs into 25 additional epochs driven by analytical models (100 by GNNs, then 25 by analytical model). We observe, in Figures 7e–7h, that this hybrid setting finds the most optimized NoC designs at all times. No latency regressions are observed when compared to NoC designs generated using original work's analytical model. While this hybrid approach is 2.1×, 1.64×, 1.34× slower than just using GNNs (10×, 8.5×, 6.1× on GPU), it is still 2.05×, 1.57×, 1.02× faster than analytical model for small, medium and large GNNs respectively (3.57×, 3.5×, 3.2× on GPU). As a result, even without GPUs, hybrid approach with small GNNs is a good middle ground that moderately accelerates application-specific NoC design without any

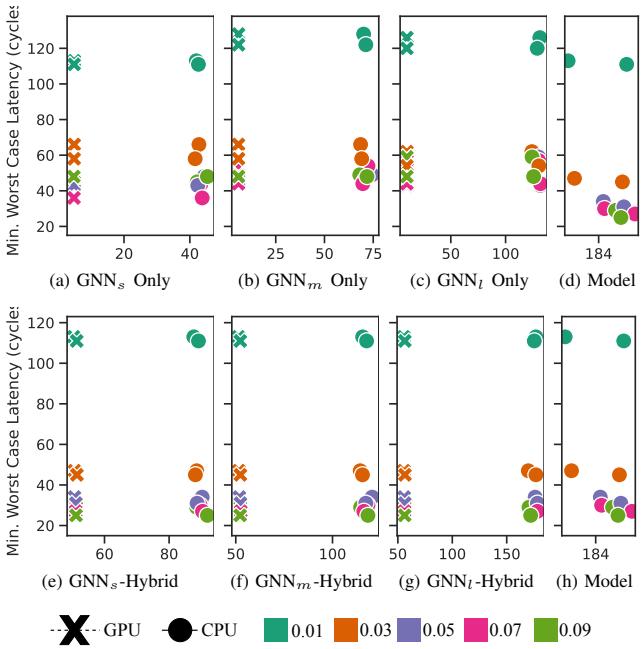


Fig. 7: Learnt NoC's latency (**Y axis**) vs seconds spent (**X axis**)

performance regressions.

To summarise this experiment, GraphNoC's GNNs can significantly accelerate application-specific NoC designs' runtime without sacrificing performance of the resulting NoCs. We observe that using GNNs accelerates NoC design's runtime by up to ≈4.3× on CPUs (increasing to ≈37× with GPUs) while only regressing up to 30 cycles in routing latency. Latency regressions can be eliminated completely with the hybrid approach while maintaining competitive speedups. Thus, GraphNoC GNNs' can supercharge the leveraging of FPGA reconfigurability by streamlining the design of highly performant NoCs specific to the FPGA application.

D. Building Foundation GNN Models with GraphNoC

In this final experiment, we investigate a GNN based model's multi-modal capabilities. We do this by training a GNN to predict worst case latency for both HopliteBuf/BP [12] and TREE BFT [15] NoCs. Note that these 2 NoCs have completely different topology, data routing and arbitration rules. We train a 10-layer, 256 dimension GraphSage+Classifier+Regression GNN model. The model is trained on encoded graphs of FPGA NoCs and FPGA applications, distributed equally between TREE BFT (8 PEs) and HopliteBuf/BP (16 PEs; 4×4). We use a batch size of 128 and cyclic learning rate schedule varying between $[10^{-7}, 5 * 10^{-6}]$.

We now share results of this model's performance on the test set. The foundation model is able to achieve a `Delta-5,10,15,20` score of 75.7%, 81.0%, 83.9%, 85.8% and a `Top-5,10,15,20` score of 69.0%, 77.5%, 82.0%, 85.3%. We now plot the distribution of ground truth and predicted latencies, faceted by NoC type, in Figure 8. Different topologies, data routing and arbitration rules are evidenced by starkly different distribution of ground truth latencies between Figures 8a and 8b. Despite this, we note that the model is able to maintain high accuracy across the distribution for both NoCs in Figure 8, even robustly covering the abnormal spike of distribution around the 2^8 mark for the BFT in Figure 8b. In addition to GNN based models' capabilities to generalise across different NoCs, this is also evidence that GraphNoC's graph encoding framework is able to encode important features of completely different NoCs and applications into richly detailed graphs for maintaining model accuracy.

We now also discuss the speed improvements of this GNN foundation model against the BFT TREE's RTL simulation (Hoplite discussed previously in section VI-B). Unlike HopliteBuf/BP, BFT has no analytical model to calculate latencies, and is thus relegated to using RTL simulations. Despite 20 `Process` wrapped parallel RTL simulations (maximum that can be launched without OOM issues), it is only able to achieve a throughput of ≈4.3 iterations per second. Contrasting this with GNN foundation model's maximum throughput of 640 iterations per second on CPU (increasing to 2176 on GPU) at batch size 128, we conclude that BFT TREE's latency estimation can be accelerated by up to ≈148× on CPU (increasing to ≈506× on GPU).

Based on our evaluation, we can conclude that GraphNoC's foundation GNNs can continue to become more advantageous over time. The foundation GNN already achieves high accuracy (`Delta-20` 85.8%; `Top-20` 85.3%) and significant speedup (up to ≈506× on GPU, ≈148× on CPU) for 2 completely different FPGA NoCs. As the foundation GNN models are trained on more diverse FPGA NoCs and applications data, the models will build a richer representation of underlying NoC routing fundamentals, similar to large language models. This can allow new NoCs to be on-boarded quicker onto GNNs, paving the way for foundation GNN models potentially replacing RTL simulations and analytical models completely in the future, including for hard FPGA NoCs like Versal [20].

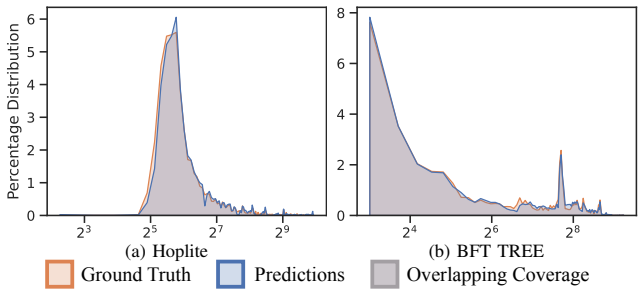


Fig. 8: Distribution of ground truth vs latency predictions.

VII. Conclusion

In this paper, we present GraphNoC, a GNN based framework and dataset, to accelerate NoC benchmarking and application-specific NoC design. GraphNoC can encode any FPGA NoC, including hard NoCs, and any FPGA application into graphs with high fidelity. We encode 3 different FPGA NoCs, over 200 different FPGA applications to create a dataset of 1.5M+ graphs that is used to train GNNs, accelerating NoC benchmarking and application-specific design by up to 148× (506× using GPU) with prediction accuracies up to 97.2%.

REFERENCES

[1] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017.

[2] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *Ieee Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[3] S. V. M. K. R. Schreiber and H. Kamepalli, "Generating simd instructions for cerebras cs-1 using polyhedral compilation techniques," 2020.

[4] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, *et al.*, "Think fast: a tensor streaming processor (tsp) for accelerating deep learning workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 145–158, IEEE, 2020.

[5] D. Ielmini and H.-S. P. Wong, "In-memory computing with resistive switching devices," *Nature electronics*, vol. 1, no. 6, pp. 333–343, 2018.

[6] A. Samajdar, T. Garg, T. Krishna, and N. Kapre, "Scaling the cascades: Interconnect-aware fpga implementation of machine learning problems," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 342–349, IEEE, 2019.

[7] W. Choi, K. Duraisamy, R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu, "On-chip communication network for efficient training of deep convolutional networks on heterogeneous manycore systems," *IEEE Transactions on Computers*, vol. 67, no. 5, pp. 672–686, 2017.

[8] H. Fang, A. Shrestha, D. Ma, and Q. Qiu, "Scalable noc-based neuromorphic hardware learning and inference," in *2018 International joint conference on neural networks (IJCNN)*, pp. 1–8, IEEE, 2018.

[9] N. Akbari and M. Modarressi, "A high-performance network-on-chip topology for neuromorphic architectures," in *2017 IEEE international conference on computational science and engineering (CSE) and IEEE international conference on embedded and ubiquitous computing (EUC)*, vol. 2, pp. 9–16, IEEE, 2017.

[10] "Solutions for self-driving cars." https://www.nvidia.com/en-us/self-driving-cars/. Accessed: 2023-01-16.

[11] G. Malik, I. E. Lang, R. Pellizzoni, and N. Kapre, "Hopliteml: Evolving application customized fpga nocs with adaptable routers and regulators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–33, 2022.

[12] G. Malik, I. E. Lang, R. Pellizoni, and N. Kapre, "Learn the switches: Evolving fpga nocs with stall-free and backpressure based routers," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 18–25, IEEE, 2020.

[13] F. Li, Y. Wang, C. Liu, H. Li, and X. Li, "Noception: a fast ppa prediction framework for network-on-chips using graph neural network," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1035–1040, IEEE, 2022.

[14] T. Garg, S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitebuf: Network calculus-based design of fpga nocs with provably stall-free fifos," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, Feb. 2020.

[15] G. S. Malik and N. Kapre, "Enhancing butterfly fat tree nocs for fpgas with lightweight flow control," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 154–162, 2019.

[16] S. Wasly, R. Pellizzoni, and N. Kapre, "HopliteRT: An efficient FPGA NoC for real-time applications," in *F. Program. Technol. (ICFPT), 2017 Int. Conf.*, pp. 64–71, IEEE, 2017.

[17] J. Silva, M. Kreutz, M. Pereira, and M. D. Costa-Abreu, "An investigation of latency prediction for noc-based communication architectures using machine learning techniques," *The Journal of Supercomputing*, vol. 75, no. 11, pp. 7573–7591, 2019.

[18] A. Kumar and B. Talawar, "Machine learning based framework to predict performance evaluation of on-chip networks," in *2018 Eleventh International Conference on Contemporary Computing (IC3)*, pp. 1–6, IEEE, 2018.

[19] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[20] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Gaide, and Y. Arbel, "Network-on-chip programmable platform in versaltm acap architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, (New York, NY, USA), pp. 212–221, ACM, 2019.

[21] B. Winefeld, "Using machine learning for characterizations of noc components," March 2019.

[22] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 86–96, IEEE, 2013.

[23] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*, pp. 1263–1272, PMLR, 2017.

[24] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds.(2018)," *arXiv preprint arXiv:1801.07829*, vol. 222, 2018.

[25] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[26] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[27] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 793–803, 2019.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition. corr abs/1512.03385 (2015)," 2015.

[29] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, "Matrix market: a web resource for test matrix collections," in *Quality of Numerical Software*, pp. 125–137, Springer, 1997.

[30] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," June 2014.

[31] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[32] "Tutorial on heterogeneous graph learning." https://pytorch-geometric.readthedocs.io/en/2.6.0/notes/heterogeneous.html. Accessed: 2024-11-09.

[33] Y. Yao, L. Rosasco, and A. Caponnetto, "On early stopping in gradient descent learning," *Constructive Approximation*, vol. 26, pp. 289–315, 2007.