EVA: Energy-Efficient Virtual Node Accelerator for Graph Neural Networks

Dhruva Barfiwala
ECE Department
Georgia Institute of Technology
Atlanta, GA
dbarfiwala3@gatech.edu

Parima Mehta
ECE Department
Georgia Institute of Technology
Atlanta, GA
parimam@gatech.edu

***Abstract*—Graph Neural Networks (GNNs) have gained traction as they can efficiently learn from graphical data structures commonly found in social science, natural science, knowledge graphs etc., unlike existing Deep Learning models that are limited to Euclidean data such as images. Virtual node augmentation on GNNs is becoming popular as modern GNNs fail to distinguish simple graph structures, which has severe repercussions if the downstream task demands an accurate graph representation vector (for example in the field of chemistry or medicine). While GNNs are powerful graph representation learning models, they are extremely hard to deploy for real-time inference due to their tremendous computation complexity. Prior work focuses on accelerating Graph Networks such as Graph Convolution Network (GCN), GraphSAGE, Graph Attention Network (GAT), Graph Isomorphism Network (GIN), however, acceleration of networks with Virtual Node remains unexplored. We aim to build an energy-efficient accelerator for GIN augmented with Virtual Node using High Level Synthesis (HLS) while minimizing performance and area overhead introduced due to the addition of a Virtual Node. We obtain latency estimates for our design on Xilinx Alveo U280 board and observe a minimal latency overhead of 10 us compared to the GIN implementations obtained from the Sharc Lab at Georgia Tech for Graph Property Predictions using the ogbg-molhiv dataset.**

***Index Terms*—Graph Neural Networks, Hardware Acceleration, Graph Isomorphism Network, Virtual Node, High Level Synthesis**

I. Introduction

The importance and popularity of Graph Neural Networks (GNNs) has been on the rise because they provide state-of-the-art performance in a variety of tasks, like node classification, graph matching, clustering, and graph generation. These tasks translate to a diverse set of real-life applications, such as network analysis, autonomous driving, high energy particle physics and a plethora of biochemical applications like protein detection, drug-drug interaction, and molecular property prediction [1].

First, let us motivate the need for GNNs; there is a wide array of data that can conveniently be represented as graphs - molecules, social media or citation networks, etc. There is no existing neat method that can directly be adopted to process graph data, and thus it made intuitive sense to adopt graph processing methods and integrate them with deep neural networks to perform different types of inference on graphs - namely node-level, edge-level and graph-level predictions. Graph processing is typically done using node-level message-passing, where a node passes its features (represented using embeddings) to its neighbors and aggregates the features it receives into its own embedding. This aggregation can be done in several ways, one of which is the Graph Isomorphism Network (GIN). GIN utilizes the Graph Isomorphism Operator to enhance the representation power of GNNs. For each graph node, GIN recursively aggregates and transforms representation vectors of its adjacent nodes. The high expressive power of GINs is useful as it can fit a diverse set of training data including large and small graphs [2].

Adding a virtual node is a common augmentation technique which helps GNNs break the Weisfeiler-Lehman test barrier and further enhances their representation power [3]. The virtual node is a dummy node that is connected to all the other nodes in the graph, and accumulates information of the whole structure during the message passing phase. Intuitively, it provides a shortcut for message passing between nodes along the graph edges [4]. It has also been shown to encode structural information which other techniques are unable to detect, such as the presence of cycles in small graphs [3]. The virtual node augmentation has provided better results when evaluated against the Open Graph Benchmark datasets [5].

In summary, GNNs achieve high performance due to their ability to extend traditional graph processing with the end-to-end learning capability of deep learning. The Deep Neural Network operations in GNNs can be translated into general matrix multiplications, which are easily parallelizable and can be optimized for latency. Graph processing in a GNN is done via node-level message-passing, which translates into two different computing modes with highly irregular data access patterns on the hardware level - the Aggregation phase, which involves vector additions with irregular strides in vector fetching, and the Transformation phase, which involves either dense or sparse-dense matrix multiplication. This leads to considerable computational costs and memory overhead when performed on conventional CPU-GPU architectures. [2], [6]. There is also a class of GNN applications which requires real-time inference, making energy-efficient GNN acceleration a lucrative field to explore [7].

In this project, we accelerate the inference of a Graph Isomorphism Network augmented with a Virtual Node using High

Level Synthesis (HLS). The goal of our project is to ensure that virtual node functionality can be augmented to an existing GNN dataflow with minimal latency and resource overhead. To that end, we ensure that an accurate GIN-virtual node functionality is achieved at a latency overhead of 10 us and an area overhead of 7.5% for two separate implementations described in IV-B. Further, we observe an overall speed up of 82.93x compared to our base implementation without any optimizations.

II. Related Work

Hardware acceleration of Graph Neural Networks is being actively pursued by the research community. However, the majority of the accelerators focus on Graph Convolution Network (GCN). BoostGCN [8] improves inference latency by exploiting massive data parallelism and reuse through hardware-aware Partition-Centric Feature Aggregation Scheme. This requires graph pre-processing making it undeployable for real-time inference. AWB-GCN [9] eliminates workload imbalance in GCN inference by implementing hardware-based autotuning framework. However, it can only be extended to GNNs with SpMMs as major arithmetic primitives, e.g. GraphSAGE.

There has been some work on acceleration of Graph Neural Networks such as GraphSAGE, GraphSAGEPool, GAT, GIN in GNNerator [10] and VersaGNN [2]. However, none of the above works explore Graph Neural Networks augmented with Virtual Node, which can lead to higher accuracy.

There is a relative dearth of accelerators for GIN due to the dense nature of its matrix multiplication operations in the Transformation phase, and the further complexity added by the presence of the virtual node leads us to believe that ours is the first accelerator specifically tailored for the GIN-Virtual Node combination. There have been works which propose generic platforms for GNN acceleration as a whole, but we believe that making network specific decisions can improve latency for a promising GNN with a potential to achieve high accuracy in real-time and memory-intensive applications.

Finally, we found that GenGNN [7] is the only current solution that proposes a generic framework to accelerate Graph Neural Networks including Virtual Nodes.

III. Background and Motivation

In this section, we describe the Virtual Node dataflow and mention the challenges associated with Virtual Node augmentation which forms the basis for the problem we are targeting in this project.

A. Virtual Node Dataflow

Virtual node augmentation to any GNN model involves three major computational steps namely Virtual Node Broadcast, Virtual Node Message Passing, and Virtual Node Update. We describe each operation in detail below:

- Virtual Node Broadcast: This operation involves adding virtual node embedding to all the node embeddings in a graph as shown in Fig. 1. This enables the virtual node to propagate structural information about the graph during message passing.
- Virtual Node Message Passing: This step involves aggregating messages from all the nodes in the graph to the virtual node as shown in Fig. 1. This message passing encodes critical structural information about the graph in the virtual node embeddings, making the augmentation particularly useful for graph property prediction.
- Virtual Node Update: Post aggregation, the virtual node embedding is transformed using a Multi-Layer Perceptron as shown in Fig. 1. This prepares the virtual node embedding for the subsequent convolution layer, where all three aforementioned steps are repeated.

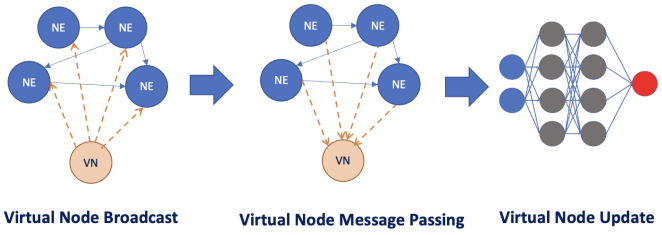

Fig. 1. Computational steps involved with Virtual Node Augmentation

B. Challenges with Virtual Node Augmentation

In any GNN augmented with virtual node, virtual node broadcast is required before message passing for graph nodes can be carried out. This step introduces huge latency overhead as it forms a major bottleneck in the GNN inference as shown in Fig. 2. Subsequently, virtual node message passing involves contribution from all nodes in the graph and can require high latency (or high resources to hide the latency) unless performed in a clever manner. This step is a prerequisite for the virtual node update phase, but does not affect any steps in the GNN dataflow. The virtual node update step is similar to the computationally complex update step in a GNN, thus requiring sophisticated optimization techniques to reduce the inference latency.

Our project focuses on the aforementioned challenges posed by virtual node augmentation, and provides a promising solution to minimize the latency and area overhead without compromising on the accuracy benefit provided by the virtual node.

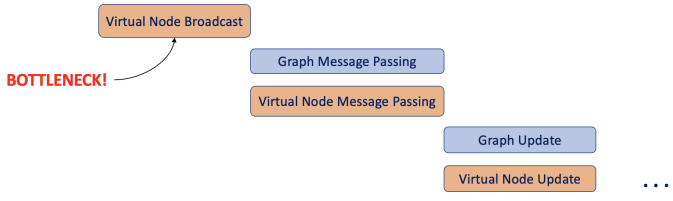

Fig. 2. Figure illustrating the challenges introduced by Virtual Node Augmentation

IV. Methodology

In this section, we describe the design flow followed to obtain a synthesizable design using HLS and explain the optimization techniques used to hide the latency overhead introduced by virtual node augmentation.

A. Golden Model Implementation

To ensure functional correctness of our GIN-Virtual node HLS implementation, we created a C implementation of the inference (hereafter referred to as "Golden C") against which any HLS code could be verified. This golden model implementation is not synthesizable using Vitis-HLS and is used strictly for verification purposes. The process of generating the golden model has been described below.

1) Extracting Model Parameters:

- Training GIN-Virtual Node Model: We trained the GIN-Virtual Node model implemented in PyTorch with 5 layers and embedding size 100 using ogbg-molhiv dataset. The weights corresponding to the trained model were stored for inference in Golden C and Accelerator implementation.
- Weight Fusion: The trained weights were obtained for GIN-Virtual Node model equipped with BatchNorm to reduce the training time. To reduce hardware complexity, we employed *weight fusion* wherein weights of linear layer in MLP are merged with the BatchNorm layer. To achieve this, we added support for GIN-Virtual Node model without BatchNorm in PyTorch and loaded the pre-trained merged weights. We verified its functional correctness by calculating the Mean Squared Error (MSE) between outputs of the model with and without BatchNorm, and observed a negligible MSE of the order of 10^{-7}.
- Weight Organization and Golden Output Collection: The fused weights of the trained GIN-Virtual Node model were organized and packed into binaries as per Golden C and accelerator implementation requirement. Further, the output of the GIN-Virtual Node model with fused weights was collected for functional verification of Golden C implementation.

2) Golden C implementation:

- Header Generation: We automate the process of generating a header containing all model weights required by the Golden C implementation.
- Loading model parameters: We implemented a function to populate global weights from binaries obtained in the parameter extraction process.
- Golden C implementation: We carried out a thorough analysis of the GIN-Virtual Node Model implemented in PyTorch and translated the same in C. In particular, support for addition of Virtual Node to Node embeddings, pooling of Node embeddings to update the Virtual Node, and transformation of Virtual Node was added. The Golden C implementation is used to verify HLS implementation.
- Golden C Verification: The Golden C implementation was verified against the PyTorch implementation and a Mean Squared error of the order of 10^{-4} was observed, which can be attributed to precision difference.

B. Virtual Node Augmentation

In order to create the HLS implementation of the GIN-Virtual Node model, we obtained the HLS implementation for GIN model created by the Sharc Lab at Georgia Tech and augmented it with our implementation for Virtual Node. Two versions of the GIN implementation were obtained, and careful analysis was carried out leading to the following observations:

- One of the implementations was a synthesizable "base" implementation with a pipelined MLP in the update phase of inference. This code will hereafter be referred to as the *Base Implementation*.
- The other implementation had further optimizations to reduce latency, with inter-layer pipelining of node embedding preparation and message passing. This code will hereafter be referred to as the *Pipelined Implementation.*

Further description of the optimizations in each code and our augmentation of virtual node with minimal latency overhead has been explained in detail below.

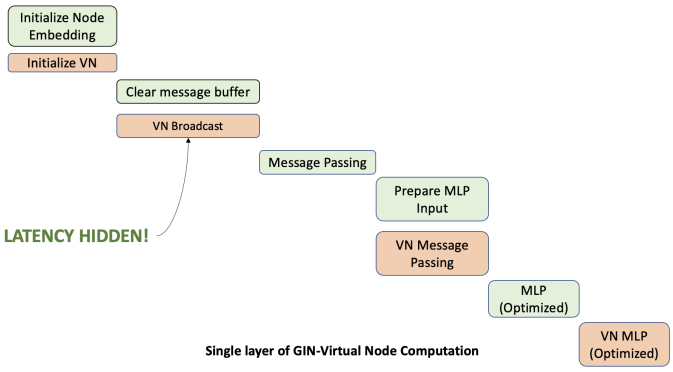

Fig. 3. Dataflow of Base Implementation

1) Base Implementation: The dataflow of one layer (including node embedding computation from node feature tables, which is only done before the first layer) of the base implementation has been described in Fig. 3. The strategy to hide virtual node latency for each part of the dataflow explained in III-A is detailed below:

- Virtual Node Broadcast: The base implementation employs the usage of a message buffer which accumulates the message to be aggregated on a per-node basis. This buffer needs to be reset to zero before the start of message passing. This is referred to as *Clear Message Buffer* in Fig. 3. The virtual node broadcast step can be parallelized with this, thus completely hiding the latency required for this.
- Virtual Node Message Passing: Upon the completion of message passing, the value in the message buffer is added to the corresponding node embedding, thus completing

aggregation and preparing the node embeddings for the *update* phase. This step is referred to as *Prepare MLP input* in Figure 3. Virtual node message passing can be performed in parallel with this, as the two tasks are independent.

- Virtual Node Update: The virtual node update step is compute intensive for GIN, as it comprises two MLP layers. In order to minimize the latency required for this, a combination of *array partitioning* and *loop pipelining* is implemented, which has been diagrammatically represented in Fig. 4. Here, virtual node update has been pipelined across the hidden layer dimensions, with each hidden layer dimension computed in one cycle using *loop unrolling* once the input layer has been partitioned. After being subjected to ReLU, this hidden layer value is then multiplied by corresponding weights and broadcast to the output layer, again in a single cycle. This aggresive pipelined implementation reduces the cycles required for virtual node update to the number of dimensions in the hidden layer.

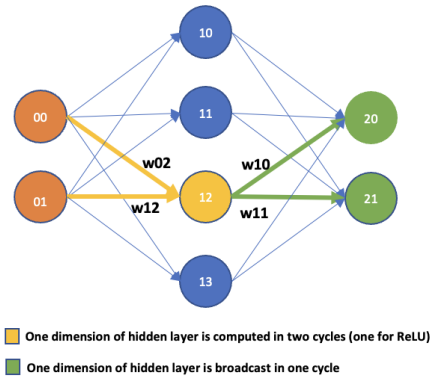

Fig. 4. MLP Dataflow after array partitioning and loop pipelining

2) Pipelined Implementation: The major shortcoming of the base implementation is the lack of pipelining across different stages of computation, thus making it unscalable for graphs with large number of nodes and edges as shown in Fig. 5. The pipelined implementation addresses this issue by cleverly pipelining the update phase for node embeddings in one layer with message passing of these node embeddings in the next layer. The dataflow of one layer of the pipelined implementation (including node embedding computation from node feature tables, which is only done before the first layer) along with the virtual node augmentation is shown in Fig. 6. To implement this successfully, the following features are required:

- Degree and Neighbor table implementation: In the pipelined implementation, message passing is performed node-by-node, which requires the computation of a neighbour table. The neighbour table stores the possible destinations to which each node sends its message, as well as the index of the corresponding edge in the edge list, while the degree table tracks the outgoing degree of each node, as well as the starting index for each node in the neighbour table. This combination enables expedited implementation of message passing for one node.
- Double buffering of message buffer: In order to pipeline message passing and MLP as seen in Fig. 6, we require double buffering of the message buffer. One buffer is used in the update phase in the current layer, while the other buffer is used in the message passing phase for the next layer.

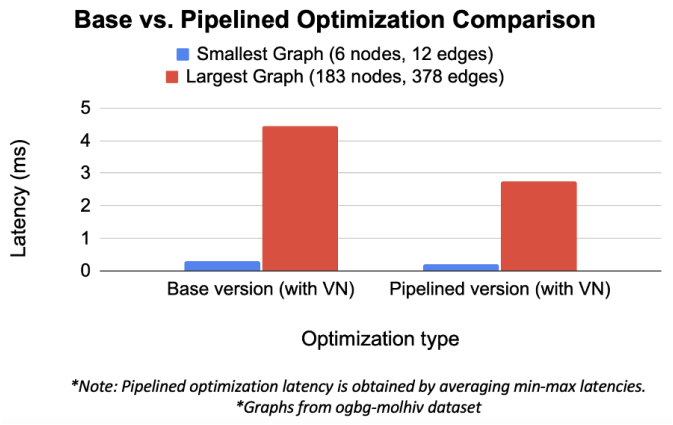

Fig. 5. Comparison of Base and Pipelined implementation for the smallest and largest graphs in the ogbg-molhiv dataset

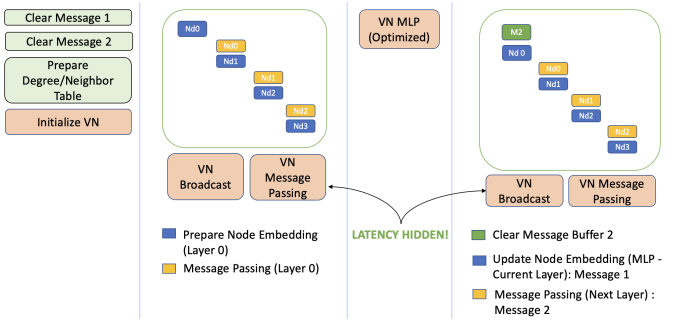

Fig. 6. Dataflow of Pipelined Implementation

The strategy to hide the virtual node latency in this pipelined implementation has been detailed below. The virtual node update step has not been mentioned because it implements the same strategy as the base implementation.

- Virtual Node Broadcast: The pipelined implementation computes the node embedding one-by-one. This enables us to reduce the virtual node broadcast into a unicast, by adding it to the node embedding as soon as it has been computed. In this manner, the latency of virtual node broadcast is completely hidden, as it adds a single cycle to the depth of a pipelined loop.
- Virtual Node Message Passing: Post the node embedding preparation (from the update phase) and the virtual node broadcast, virtual node message passing is performed to

prepare the input for the virtual node update phase. This adds a single cycle to the depth of the pipelined loop prepared to process nodes one-by-one in a graph, thus completely hiding the latency associated with virtual node message passing as demonstrated in Fig. 6.

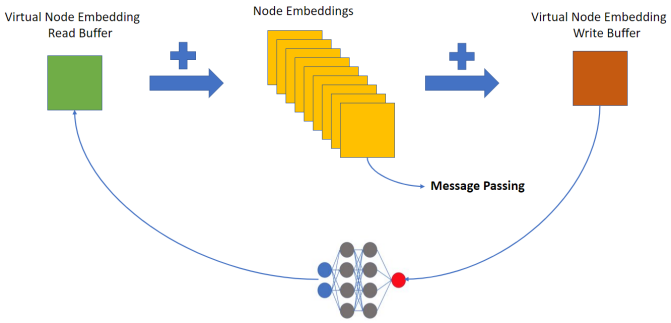

Fig. 7. Read-Write Buffer for virtual node embeddings

In order to enable simultaneous implementation of virtual node broadcast and virtual node message passing, we employ the usage of read-write buffers for the virtual node embedding as demonstrated in Fig. 7. The **read buffer** is used during Virtual Node Broadcast wherein the virtual node embedding for the current layer is read and added to all the node embeddings of a graph. This buffer can not be modified till all the nodes of the graphs are processed. Thus, to handle simultaneous message passing, a **write buffer** is introduced that performs Virtual Node message passing to prepare input for the Virtual Node Update phase which produces the virtual node embedding to be utilized by the subsequent layer.

V. Evaluations

We conduct two sets of evaluations on our design. First, we analyse the performance and area overhead introduced by virtual node augmentation by comparing against the GIN model implementations obtained from the Sharc Lab as described in section IV-B. Second, we analyze the incremental speedup obtained from each optimization technique mentioned in section IV-B.

We obtain overhead and speedup estimates using the setup described in Table I. To simplify our analysis, our estimates are based on one graph containing 19 nodes and 40 edges. This can be extended to graphs of different sizes to gain insight into the average latency and resource utilization of the design.

TABLE I
Evaluation Setup

Vitis HLS	2021.1
Board	Xilinx Alveo U280
Dataset	ogbg-molhiv
Graph Info	19 nodes, 40 edges

A. Virtual Node Overhead Analysis

In this section, we analyze the trade-off between accuracy and overhead introduced by the augmentation of virtual node to the GIN model implementation obtained from the Sharc Lab at Georgia Tech.

Table II enumerates the test accuracy of GIN and GIN-Virtual Node models that are trained for 1 epoch.

TABLE II
Test Accuracy (ROC AUC score)

Model Type	Test Accuracy
GIN	66.4
GIN-Virtual Node	68.05

1) Performance Overhead: It can be observed from Fig. 8 that the virtual node augmentation to GIN implementation incurs a minimal latency overhead of 10 us for both base and pipelined implementations. This is because latencies associated with Virtual Node Broadcast and Virtual Node Message Passing are hidden as described in section IV-B.

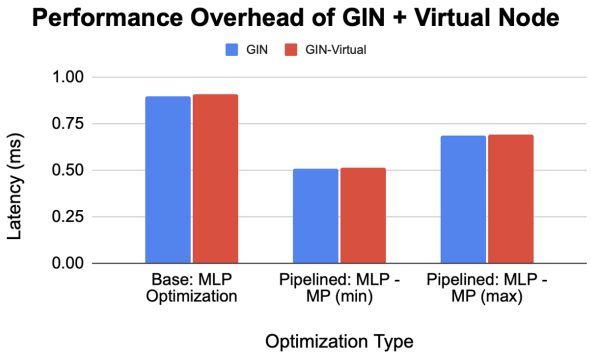

Fig. 8. Net compute time overhead of Virtual Node

2) Area Overhead: It can be observed from Fig. 9 and 10 that the resource utilization is higher in both base and pipelined implementations which is expected due to the addition of virtual node embeddings. However, the overhead is minimal with all resource types being within 30% of the total available resource mentioned in Table III.

TABLE III
Total Available Resource on Xilinx Alveo U280)

Resource Type	Availability
BRAM_18K	4032
DSP	9024
FF	2607360
LUT	1303680
URAM	960

B. Speedup Analysis

In this section, we analyze the speedup associated with each optimization type employed in section IV-B. We start with a vanilla implementation of GIN model augmented with Virtual Node and add optimizations in the update phase followed by inter-layer pipelining of message passing and the update phase. It can be observed from Table IV that the inter-layer pipelining

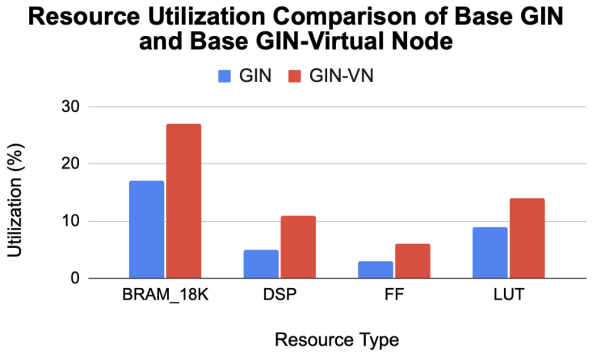

Fig. 9. Resource Utilization overhead of Virtual Node in the Base Implementation

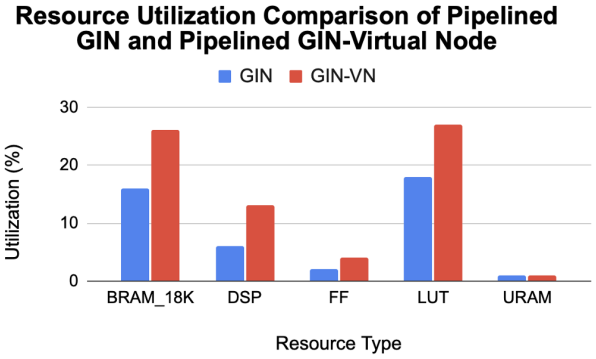

Fig. 10. Resource Utilization overhead of Virtual Node in the Pipelined Implementation

provides the maximum speedup of 82.93x compared to the unoptimized implementation of GIN-Virtual Node model.

TABLE IV
SPEED UP ANALYSIS OF GIN-VIRTUAL NODE MODEL

Optimization Type	None	Base	Pipelined
Latency ms	50.116	0.907	0.6043
Speedup	1	55.25x	82.93x

Further, we compare the resource utilization of base and pipelined implementation with the unoptimized implementation. Trade-off between area and latency can be observed as the resource utilization increases with the increased optimizations as seen in Fig. 11. The pipelined implementation has the highest resource utilization which can be attributed to the use of degree and neighbor table, double buffers, that are essential to achieve inter-layer pipelining. Finally, it is evident that the resource utilization is well within limits with all resource types being within 30% of the total available resource mentioned in Table III.

VI. Summary and Future Work

In this project, we successfully augmented Virtual Node to the base and pipelined GIN implementations provided by Sharc Lab. Further, we proposed a comprehensive solution

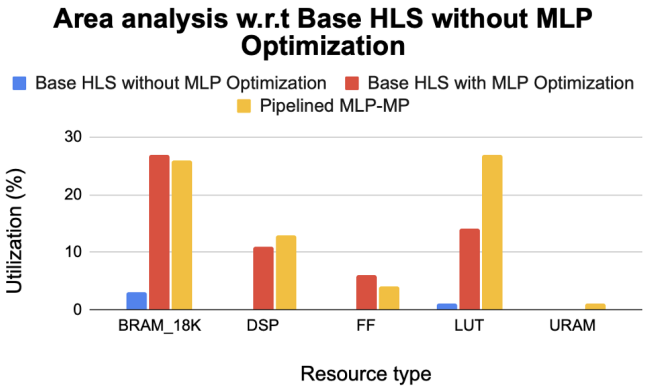

Fig. 11. Resource Utilization of Virtual Node for all implementations

to augment virtual node to enhance representation power of GIN while ensuring minimum latency and area overhead of virtual node. Finally, we conducted speedup analysis to understand the effect of each optimization technique on the overall inference latency. To summarize, we demonstrated that real-time inference of highly accurate GNN models can be achieved without incurring huge latency and area overhead due to the addition of Virtual Node.

In the future, this design can be extended by exploring the following areas:

- Multiple Virtual Nodes: To achieve high accuracy, more than one virtual node can be augmented to GNN models. The impact of adding multiple virtual nodes on latency and resource utilization can be analyzed, thus establishing a fine balance between model test accuracy, performance, and area.
- Intra-layer pipelining: Instead of inter-layer pipelining, intra-layer pipelining can be explored wherein update phase for a given node begins as soon as the message passing for that node finishes, as waiting for message passing of all nodes is not required to start the update phase for a given node. The speedup and resource utilization achieved from such an implementation can be compared against the design with inter-layer pipelining.
- Handling large graphs: Current design is targeted towards relatively smaller graphs that can be loaded entirely to the on-chip memory. Handling of large graphs can be achieved by enabling pipelining between loading of graphs with the message passing phase. Further, data streaming can also be explored.

VII. Acknowledgement

The authors would like to thank Professor Callie Hao and the Sharc Lab at Georgia Institute of Technology for providing the base GIN implementation to which the Virtual Node implementation was added, and for providing guidance throughout the project.

REFERENCES

[1] K. Atz, F. Grisoni, and G. Schneider, "Geometric deep learning on molecular representations," *Nature Machine Intelligence*, vol. 3, 12 2021.

[2] F. Shi, A. Y. Jin, and S. Zhu, "Versagnn: a versatile accelerator for graph neural networks," *CoRR*, vol. abs/2105.01280, 2021.

[3] R. Brossard, O. Frigo, and D. Dehaene, "Graph convolutions that can finally model local structure," *CoRR*, vol. abs/2011.15069, 2020.

[4] E. Hwang, V. Thost, S. S. Dasgupta, and T. Ma, "Revisiting virtual nodes in graph neural networks for link prediction," 2022.

[5] 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," *CoRR*, vol. abs/2005.00687, 2020.

[6] Z. Zhang, J. Leng, S. Lu, Y. Miao, Y. Diao, M. Guo, C. Li, and Y. Zhu, "ZIPPER: exploiting tile- and operator-level parallelism for general and scalable graph neural network acceleration," *CoRR*, vol. abs/2107.08709, 2021.

[7] S. Abi-Karam, Y. He, R. Sarkar, L. Sathidevi, Z. Qiao, and C. Hao, "Gengnn: A generic FPGA framework for graph neural network acceleration," *CoRR*, vol. abs/2201.08475, 2022.

[8] B. Zhang, R. Kannan, and V. Prasanna, "Boostgcn: A framework for optimizing gcn inference on fpga," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 29–39, 2021.

[9] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 922–936, 2020.

[10] J. R. Stevens, D. Das, S. Avancha, B. Kaul, and A. Raghunathan, "Gnnerator: A hardware/software framework for accelerating graph neural networks," *CoRR*, vol. abs/2103.10836, 2021.