GNNBUILDER: AN AUTOMATED FRAMEWORK FOR GENERIC GRAPH NEURAL NETWORK ACCELERATOR GENERATION, SIMULATION, AND OPTIMIZATION

A Dissertation
Presented to
The Academic Faculty

By

Stefan Abi-Karam

December 2022

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

© Stefan Abi-Karam

GNNBUILDER: AN AUTOMATED FRAMEWORK FOR GENERIC GRAPH NEURAL NETWORK ACCELERATOR GENERATION, SIMULATION, AND OPTIMIZATION

Thesis committee:

Dr. Cong Hao
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Date approved: December 5, 2022

ACKNOWLEDGMENTS

I would like to thank the members of my thesis committee for providing their time and attention to review and providing feedback on my ideas: Prof. Callie Hao, who has primarily guided me through transiting into graduate school and into academia to become a stronger researcher, Prof. Tushar Krishna who guided me into more cross-collaboration with other research areas, and Prof. Hyesoon Kim for providing an outside perspective on my research area.

I would also like gratefully acknowledge the support from Georgia Tech Research Institute for funding me through my graduate studies and providing me with the academic freedom to explore novel research areas.

Finally, I would like to acknowledge my partner, Lauren Barger, for her endless love, support, and encouragement during my master's and my transition into a Ph.D.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables vii

List of Figures viii

Summary ix

Chapter 1: Introduction 1

Chapter 2: Background 5

2.1 Related Work and Motivations 5

2.1.1 Related Work 5

2.1.2 Limitations 6

Chapter 3: Methodology 8

3.1 GNNBuilder Framework Overview 8

3.1.1 GNNBuilder Components 8

3.1.2 Programming Model and User APIs 10

3.2 GNNBuilder Model Architecture 13

3.3 Hardware Accelerator Architecture 14

3.3.1 Graph Data and Internal Buffers 14

3.3.2 Degree Table and Neighbor Table Computation 15
3.3.3 Message Passing and Graph Convolution Kernels 15
3.3.4 Partial Aggregations 17
3.3.5 Linear Layer 17
3.3.6 Global Pooling 18
3.3.7 Activations 18
3.4 Accelerator Generation and Implementation 18
3.4.1 Kernel Code Generation 19
3.4.2 Hardware Simulation and Verification Testbenches 19
3.4.3 Hardware Deployment on FPGA 20
3.5 Performance Model and Design Space Exploration 21
3.5.1 Hardware Model Implementation Details 21
3.5.2 Hardware Performance Model 21
3.5.3 Design Space Exploration 22

Chapter 4: Experimental Setup 24
4.0.1 Analytical Performance Model 24
4.0.2 Accelerator Performance Evaluation 25

Chapter 5: Results 28
5.0.1 Analytical Performance Model 28
5.0.2 DSE Exploration 28
5.0.3 Accelerator Performance Evaluation 29

Chapter 6: Conclusion 32

References 34

LIST OF TABLES

1.1 Comparison with Existing Work 4

3.1 Supported GNNs 8

3.2 User Programming APIs 10

5.1 Implemented Accelerator FPGA Resource Usage 30

5.2 FPGA-Parallel Accelerator Speedup 31

LIST OF FIGURES

3.1 GNNBuilder Framework Workflow Overview 8

3.2 GNNBuilder Model Architecture 13

3.3 Hardware Kernel Architecture for GNNConv Layer 16

5.1 Latency + BRAM DSE Models 29

5.2 Cumulative DSE Runtime Evaluation 30

5.3 GNN Latency Comparison Across Implementations 31

SUMMARY

There are plenty of graph neural network (GNN) accelerators being proposed. However, they highly rely on users' hardware expertise and are usually optimized for one specific GNN model, making them challenging for practical usage. Therefore, in this work, we propose GNNBuilder, the first automated, generic, end-to-end GNN accelerator generation framework. It features four advantages: (1) GNNBuilder can automatically generate GNN accelerators for a wide range of GNN models arbitrarily defined by users; (2) GNNBuilder takes standard PyTorch programming interface, introducing zero overhead for algorithm developers; (3) GNNBuilder supports end-to-end code generation, simulation, accelerator optimization, and hardware deployment, realizing a push-button fashion for GNN accelerator design; (4) GNNBuilder is equipped with accurate performance models of the generated accelerator, enabling fast and flexible design space exploration (DSE). In the experiments, we show that our accelerator performance model has errors within 34% for latency prediction and 22% for BRAM count prediction. We also show that our generated accelerators can outperform CPU by $2.96\times$ and GPU by $2.99\times$. This framework is open-source, and the code is available at https://anonymous.4open.science/r/gnn-builder.

CHAPTER 1
INTRODUCTION

Graph Neural Networks (GNNs) are a powerful and popular tool for solving learning tasks where the data can be represented as a graph. Among different applications, GNNs can be used for node-level, edge-level, and graph-level tasks, such as drug discovery [1], recommender systems [2], social network analysis [3], traffic forecasting [4], electronic health records analysis [5], scene graph understanding [6], electronic design automation [7], natural language processing [8], autonomous driving [9], and high-energy physics [10]. Among these applications, some have real-time constraints for GNN inference and require hardware acceleration. One example is autonomous driving systems that use GNNs to process LIDAR point cloud data [11]. Another prominent example is in high-energy physics, where GNNs are used for real-time particle detection [12] and jet lag detection [13], which must be processed within several nano-second.

Given the acceleration needs for GNN inference, there are many GNN accelerators being proposed. Examples include earliest ASIC accelerators proposed by Auten et al. [14], HyGCN [15], and EnGN [16], as well as most recent accelerators such as AWB-GCN [17], BoostGCN [18], and I-GCN [19], GCNAX [20], Rubik [21], and GraphACT [22]. Among them, Rubik and GraphACT aim to accelerate GCN training using ASIC and FPGA, respectively.

Despite the great success of GNN accelerators, there are still significant **limitations**. First, *existing GNN accelerators are model-specific but not generic*. Specifically, most GNN accelerators focus on only one or two most popular GNN models, such as Graph Convolution Network (GCN) [23] or GraphSage [24], and provide fixed accelerator structures, fixed GNN layer types, activations, and other design choices that are specific to the implemented model. However, these accelerators are not generic, and *cannot handle advanced*

GNNs such as anisotropic GNNs, GNNs with edge embeddings, or complicated aggregation functions [25, 26, 27]. The fundamental reason is that the existing GNN accelerators simplify GNN computations to be a sequence of general or sparse matrix multiplications, which *does not hold true* for those advanced GNNs. Second, *most of the accelerators are hard-coded and require extensive hardware expertise to adapt to new GNN models*. There are no existing tools that can generate GNN accelerators automatically, optimally, and without any hardware knowledge. There is only two existing work that can support automated accelerator generation: DeepBuring-GL [28], and HP-GNN [29]. DeepBuring-GL targets inference acceleration but is limited to a fixed GCN or GraphSAGE model. HP-GNN targets training acceleration but not real-time-inference. More importantly, HP-GNN proposes their own programming language and lacks the flexibility of supporting a wide range of GNN architectures and different features. Table Table 1.1 summarizes the limitations of DeepBuring-GL and HP-GNN. Consequently, there is no way that researchers and practitioners can explore the best GNN model for their target applications in software and easily deploy their application-specific models to hardware for acceleration.

Motivated by the existing limitations of GNN accelerator designs and tools, we propose GNNBuilder, a generic, feature-rich, and extensible framework for end-to-end GNN accelerator generation, simulation, optimization, and deployment on FPGAs with bitstreams. To be generic, we follow the *message passing mechanism* of GNN models, which can express almost all types of GNN models at the theoretical formulation level, as stated by a recent work [30]. To be extensible, we directly take *standard PyTorch* as the programming language, which allows the programmers to design their own GNN models freely. We summarize our contributions as follows:

- **Generic: wide range of GNN model support**. Using an explicit message passing framework, GNNBuilder can support not only state-of-the-art GNN models such as GCN, GIN, GraphSAGE, and PNA, but also allows programmers to design customized GNN models with arbitrary layer type, activation, quantization (data precision), aggregation, pooling,

etc. Table Table 1.1 summarizes the features that can be customized in GNNBuilder but not in HP-GNN.

- **Extensibility: interoperability with PyTorch**. GNNBuilder is the first work that allows users to define their model architectures freely in native PyTorch using a parameterizable GNNModel PyTorch module. This allows users to seamlessly integrate accelerator design as part of existing deep learning workflows. Therefore, GNNBuilder does not only support standard GNNs (as listed in Table Table 3.1) but can extend to almost all customized GNN models.

- **Complete model architecture for multi-level tasks**. Our GNNBuilder supports node-level, edge-level, and graph-level tasks with user-defined parameterizable multi-layer perceptron (MLP). Specifically, graph-level classification and regression tasks are particularly important for drug and molecule-related applications and high-energy physics applications.

- **Accelerator design space exploration (DSE) and optimization**. GNNBuilder provides tolling to automatically aid the designer in selecting the best configurations, such as hardware parallelism, resource allocation, and fixed point precision, rather than manual user tuning. This allows for gains in performance to achieve the best latency under fixed resource constraints with a trade-off in model output error.

- **Open-source Python API with end-to-end workflow**. Our GNNBuilder provides open-source Python library APIs, which allow users to define their own model. It also has an end-to-end workflow with hardware-compatible simulation, testbench build and execution, automated hardware code generation and synthesis, and deployment on FPGA with host code. It provides a push-button flow from development to deployment with zero hardware expertise required.

- **Superior performance against CPU and GPU.** GNNBuilder generates high-performance

Table 1.1: Comparison with existing work. The most fundamental differences are programming language, support for anisotropic GNN family, and extensibility, making it highly practical for future GNNs and broader usage.

	HP-GNN [29]	**DeepBurning-GL [28]**	**GNNBuilder**
Acceleration Goal	Training	Inference	Inference
Programming Language	Self-defined	PyTorch and DGL	PyTorch 👍
Anisotropic GNN Family	No	No	Yes 👍
Extensibility	Low	Low	Very High 👍
Arbitrary Quantization	No	No	Yes 👍
Arbitrary Aggregation	No	No	Yes 👍
Arbitrary Activation	Fixed	Fixed	Arbitrary 👍
Skip Connections	No	No	Yes 👍
Arbitrary Global Pooling	No	No	Yes 👍
Arbitrary MLP Head	No	No	Yes 👍
Fixed / Floating Point Testbench	No	No	Yes 👍
Open Source	No	No	Yes 👍

accelerators on FPGA that outperform CPU and GPU baselines on various datasets with a **2.18**× average speed up and **3.03**× max speedup.

CHAPTER 2

BACKGROUND

2.1 Related Work and Motivations

2.1.1 Related Work

GNN Accelerators and Graph Accelerators

The growing use of GNNs in real-time and large-data applications in the research community and industry has resulted in numerous GNN accelerator works. A recent survey paper [31] presents an overview of GNN accelerator works for CPU, GPU, ASIC, FPGA, and heterogeneous platforms. Some specific GNN accelerator works include Auten et al. [32], HyGCN [15], AWB-GCN [17], EnGN [16], GRIP [33], GCNAX [20], Rubik [21], GraphACT [22], Boost-GCN [18], and I-GCN [19]. All these works explore different implementations and model-specific design choices to achieve speedups in GNN inference and training. Newer works, like GenGNN [34] and FlowGNN [35], also take a GNN model agnostic approach to inference acceleration without sacrificing performance.

Conversely, general graph processing accelerator works are also popular for tasks such as PageRank, shortest path, connected components, and normalizing flow problems. These works include GraphH [36], Blogel [37], Giraph++[38], ForeGraph [39], FabGraph [40], HitGraph [41], AccuGraph [42], and ThunderGP [43]. These approaches typically adopt a scatter-apply-gather to support generalized graph problems.

GNN Accelerator Automation

Some existing works explore the automated generation of hardware accelerators for GNNs. One key work is DeepBuring-GL [28], which is targeted for generating GNN inference accelerators for FPGAs for a fixed subset of GCN-based architectures. Another work,

HP-GNN [29], also targets acceleration but for GNN training on CPU-FPGA platforms. HP-GNN also supports a subset GCN-based and SAGE-based architectures.

2.1.2 Limitations

GNN Accelerators

Most acceleration works for GNN inference using fixed model architectures. The majority also only support models that are not anisotropic in order to take advantage of existing sparse matrix multiplication acceleration techniques. Many works also implement GCN architectures and simplify the computations with sparse matrix multiplications (SpMM) and general matrix multiplications (GEMM). However, advanced GNNs cannot be simplified as matrix multiplications. Furthermore, many other works use specialized graph preprocessing and model computation patterns that are not generalizable to anisotropic models. This optimization focus is another limitation of existing acceleration works, which prevents them from generalizing to advanced GNN architectures. Recent works like GenGNN and FlowGNN implement hardware architectures that can generalize to advanced model architectures with support for anisotropic message passing by adapting an explicate message passing hardware dataflow.

GNN Accelerator Automation

Existing accelerator automation works lack key features that allow them to generalize to advanced GNN architectures. We highlight these limitations in Table Table 1.1 in the context of our work. Even though DeepBuring-GL supports end-to-end code generation, it is limited to GCN and GraphSAGE models because it is based on a systolic array design. Moreover, HP-GNN is limited to GCN and GraphSAGE models, and not explicitly designed for inference. Neither works support anisotropic GNNs such as PNA as more expressive GNNs such as GIN. Furthermore, they do not support features found in GNN models like mean and variance neighbor pooling, arbitrary activation functions, skip connections, sum /

mean / max global pooling, and MLP prediction heads. Additionally, these frameworks do not provide simple fixed-point quantization or code generation for fixed-point and floating-point testbenches for rapid debugging. Most importantly, these works don't provide source code for researchers and practitioners to use in their own applications.

CHAPTER 3

METHODOLOGY

3.1 GNNBuilder Framework Overview

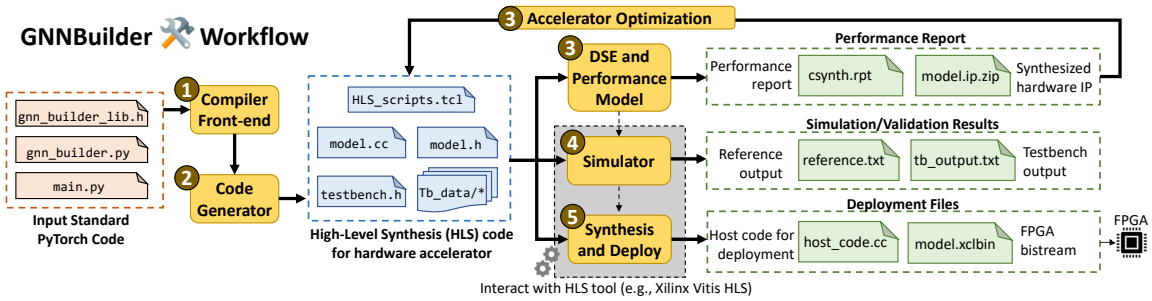

Figure 3.1: Workflow of the GNNBuilder framework.

3.1.1 GNNBuilder Components

The goal of GNNBuilder is to provide users with simple tooling to design, implement, validate, and optimize GNN models from standard PyTorch models to FPGA bitstream. As shown in Fig. Figure 3.1, GNNBuilder is composed of five components.

❶ **Compiler front-end**, which parses the native PyTorch GNN model definition, such

Table 3.1: Supported representative GNNs by our framework GNNBuilder. It also supports user-defined GNN models.

Model	**Representativeness**
GCN [23]	GNN family that can be represented as sparse matrix-matrix multiplications (SpMM)
GraphSage [24]	GNN family with flexible / non-sum aggregation methods
GIN [26]	GNN family with *edge embeddings*, SpMM *does not* apply
PNA [27]	A popular Anisotropic GNN family arbitrarily using multiple aggregation methods and sophisticated message function, SpMM *does not* apply
GCN: graph convolutional network; **GIN:** graph isomorphism network; **GraphSAGE:** graph sample and aggregate; **PNA:** principal neighbourhood aggregation.	

as the number of GNN layers, layer type, activation type, data precision, pooling type, aggregation type, and MLP definition.

❷ **Code generator**, which builds on top of a library of pre-defined hardware accelerator templates. Note that the template (to be introduced in Section section 3.3) adopts the message passing mechanism and thus is generic to almost all GNN types. Targeting FPGAs, we generate High-Level Synthesis (HLS) code supported by Xilinx's VitisHLS tool.

❸ **Design space exploration and performance model**, which applies automated DSE for the accelerator generation, including hardware parallelism, resource allocation, and quantization (data precision).

❹ **Simulation and testbench**, which allows transparent hardware-compatible simulation using automatically generated testbenches to guarantee the correctness of the accelerator functionality. More importantly, it generates plain C++ code for "true" quantization simulation, which can honestly reflect on-FPGA quantization accuracy[1].

❺ **Hardware synthesis and deployment**, which automatically generates hardware synthesis scripts, synthesizes the design into an FPGA bitstream, and generates host code for executing the bitstream.

Table Table 3.1 lists the representative GNNs supported by our framework. Note that these are just examples, but GNNBuilder can flexibly support a wide range of customized GNN models. Examples include residual and skip connections, arbitrary quantization, arbitrary aggregation function, graph attention, activation, global pooling, and MLP head. Such user-defined features can be naturally expressed using PyTorch, and thus GNNBuilder enjoys great extensibility.

[1]The quantization simulated in Python is usually regarded as "fake" quantization since the arithmetic operations in Python can only be done in floating point precision, even though the operands are represented using fixed-point precision.

API Functions	Description
code_gen.Project()	GNNBuilder Project Class
code_gen.Model_AGNN()	Wrapper Class for GNNModel
model.GNNModel(nn.Module)	PyTorch Model for GNNBuilder Arch.
model.GCNConv_AGNN(nn.Module)	GCN Conv. Layer
model.GINConv_AGNN(nn.Module)	GIN Conv. Layer
model.PNAConv_AGNN(nn.Module)	PNA Conv. Layer
model.SAGEConv_AGNN(nn.Module)	GraphSAGE Conv. Layer
model.GlobalPooling(nn.Module)	Global Graph Pooling Layer
model.MLP(nn.Module)	MLP Prediction Head
perf_model.compute_model_runtime	Runtime Model
Project.gen_hw_model()	Code Gen. For HW Kernel
Project.gen_testbench()	Code Gen. For Testbench
Project.gen_makefile()	Code Gen. For Testbench Makefile
Project.gen_vitis_hls_tcl_script()	Code Gen. For Vitis HLS Synth. Script
Project.build_and_run_testbench()	Build and Run Testbench
Project.run_vitis_hls_synthesis()	Launch Vitis HLS Synthesis Run

Table 3.2: User Programming APIs.

3.1.2 Programming Model and User APIs

Table Table 3.2 displays all user APIs provided by GNNBuilder, and Listing 1 displays an example of the user interface for a customized GNN model.

A user begins by defining a GNNModel instance which incorporates an MLP and an xxxConv_AGNN module (ex. PNAConv_AGNN). GNNBuilder provide these wrapper classes for each graph convolution layer to allow the user to specify parallelism factors p_in and p_out. The higher level GNNModel supports arguments for defining architecture parameters as well as separate parallelism factors for the GNN head (gnn_p_in, gnn_p_hidden, gnn_p_out) and the MLP head (p_in, p_hidden, p_out). The user can then train and manipulate the GNNModel instance as a standard PyTorch module. Once ready for accelerator implementation, the user can then wrap the GNNModel instance with the Model_AGNN by passing it to that class (ex. Model_AGNN(GNNModel_inst)).

A user can then define a GNNBuilder Project instance. The Project class has several

arguments to define build paths, the GNNModel model instance, the PyTorch Geometric dataset for he model task, max_nodes and max_edges, numerical precision, and average number of nodes, edges, and node in-degree for synthesis runtime estimation.

Once a Project instance is created, the user can then call the code gen. instance functions to generate the model kernel HLS code, the kernel testbench code + data, the testbench makefile, and the Vitis HLS build script. After code gen., the user can then call build_and_run_testbench() to build and execute testbench and run_vitis_hls_synthesis() to execute the Vitis HLS synthesis run. These execution scripts also return data for the testbench runtime + MAE and synthesis latency + resource usage.

```
import torch
import torch.nn as nn
from torch_geometric.datasets import TUDataset

import gnnbuilder as agnn
from gnnbuilder.code_gen import FPX

dataset = TUDataset(root="TUDataset")

model = agnn.GNNModel(
    graph_input_feature_dim=dim_in,
    graph_input_edge_dim=0,
    gnn_hidden_dim=128,
    gnn_num_layers=6,
    gnn_output_dim=64,
    gnn_conv=conv,
    gnn_activation=nn.ReLU,
    gnn_skip_connection=True,
    global_pooling=agnn.GlobalPooling(["add", "mean", "max"]),
    mlp_head=MLP(in_dim=64 * 3, out_dim=dim_out, hidden_dim=64, hidden_layers=4, activation
      =nn.ReLU, p_in=8, p_hidden=8, p_out=1,),
    output_activation=None,
```

```
    gnn_p_in=1,
    gnn_p_hidden=8,
    gnn_p_out=8
)
agnn_model = agnn.Model_AGNN(model)

MAX_NODES = 600
MAX_EDGES = 600
num_nodes_avg, num_edges_avg = agnn.compute_average_nodes_and_edges(dataset)
degree_avg = compute_average_degree(dataset)

proj = agnn.Project(
    "gnn_model",
    agnn_model,
    "classification_integer",
    VITIS_HLS_PATH,
    BUILD_DIR,
    dataset=dataset,
    max_nodes=MAX_NODES,
    max_edges=MAX_EDGES,
    num_nodes_guess=num_nodes_avg,
    num_edges_guess=num_edges_avg,
    degree_guess=degree_avg,
    float_or_fixed="fixed",
    fpx=FPX(32, 16)
)

proj.gen_hw_model()
proj.gen_testbench()
proj.gen_makefile()
proj.gen_vitis_hls_tcl_script()

tb_data = proj.build_and_run_testbench()
```

```
print(tb_data)
synth_data = proj.run_vitis_hls_synthesis()
print(synth_data)
```

Listing 3.1: Example usage of GNNBuilder Framework

3.2 GNNBuilder Model Architecture

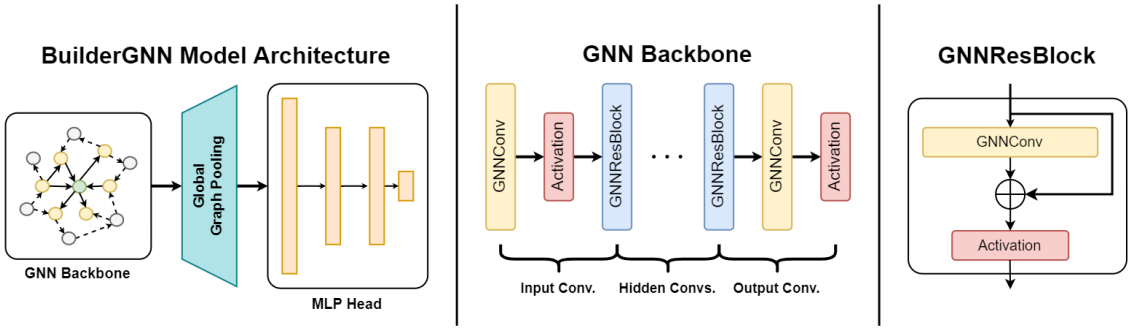

Figure 3.2: The GNNBuilder model architecture for graph-level tasks.

GNNBuilder supports node-level, edge-level, and graph-level tasks. Since most GNN models share similar architectures, GNNBuilder constructs GNN models follow a general architecture as shown in Fig. Figure 3.2 left. It demonstrates a general GNN model architecture for graph-level tasks, including a GNN backbone, a global graph polling, and an MLP prediction head. Note that for edge and node level tasks, users can simply remove the global pooling and the MLP head.

The **GNN Backbone** consists of a sequence graph convolution layers, activation, and skip connections, much like a traditional MLP head. Graph convolution layers are referred to as GNNConv layers, and blocks of skip connections, activations, and GNNConvs are referred to as GNNResBlocks. The user has the option of specifying the number of layers, the size of the hidden layer embedding, the size of the output layer embedding, the activation function to be used, and whether skip connections should be used. When there are more than two layers specified and skip connections are used, GNNResBlocks will be

used as middle layers in the GNN backbone. The GNNConv layers supported are GCN, GraphSAGE, GIN, and PNA.

The **Global Graph Pooling** module aggregates the node embedding output from the GNN backbone. To produce a single embedding for an aggregation function, all node embeddings are aggregated across all nodes. In the case of multiple aggregations, the aggregated embeddings are concatenated together. GNNBuilder supports sum pooling, mean pooling, and max pooling global aggregation functions.

The **MLP Prediction Head** transforms the output from the global graph pooling to the output for the specified model task. The user can specify the input embedding size, the output embedding size, the number of hidden layers, and the activation used for intermediate layers.

Since the final layer of the MLP head is a linear layer with no output activation, the user also has the option to specify a final output activation if desired.

The user also can specify parallelism factors for both the GNN head and the MLP head. However, these factors are purely used in the framework for hardware implementation and don't affect the software implementation (ex. PyTorch training).

3.3 Hardware Accelerator Architecture

3.3.1 Graph Data and Internal Buffers

Any buffers in the model kernel that depend on the number of nodes (num_nodes) or the number of edges (num_edges) in an input graph require the buffer sizes to be set to an upper bound. These are the MAX_NODES and MAX_EDGES parameters which are set as part of a GNNBuilder Project instance.

GNNBuilder also generates an input buffer, an output buffer, and two ping-pong buffers to store intermediate outputs from layers in the GNN head and the MLP head. These buffers are of size MAX_NODES $\times$ emb_dim and emb_dim respectively.

The model kernels generated GNNBuilder expect an input graph to be represented as

COOrdinate format matrix along with an input node feature table. The COO matrix is represented as a MAX_EDGES $\times 2$ integer array. The input node feature table is represented as a MAX_NODES $\times$ input_dim array using a fixed-point datatype with each row representing the input features vector for each node. An in-degree buffer and out-degree buffer of size MAX_NODES exist alongside the COO buffer.

There are also two additional buffers: a neighbor table and a neighbor offset table. The neighbor table is size MAX_EDGES stores a block of the neighbors of each node. The neighbor offset table is size MAX_NODES and stores the offset index into the neighbor table to index every node's block of neighbors.

3.3.2 Degree Table and Neighbor Table Computation

Before performing the model computations, the degree table of the input graph needs to be computed. Node degrees are used by various graph convolutions in order to perform normalization based on node degree. Since these values are only known at runtime, the in-degree and out-degree tables need to be computed in the kernel for each input graph. The COO format of input graphs allows for the computation to iterate with the bounds of num_edges. After the degree tables are computed, the neighbor table and neighbor offset table are computed simultaneously with two loops, one over num_edges and one over num_nodes.

3.3.3 Message Passing and Graph Convolution Kernels

Inspired by a recent generic GNN accelerator, GenGNN, we adopt an explicit message passing architecture for the GNNConv / graph convolution kernels. This allows us to support GNN layers, including PNA, that cannot be supported by traditional SpMM accelerator approaches.

For each node in the graph, the operations in Figure Figure 3.3 are performed. First, the current node's neighbors' indexes are gathered using the neighbor table and neighbor

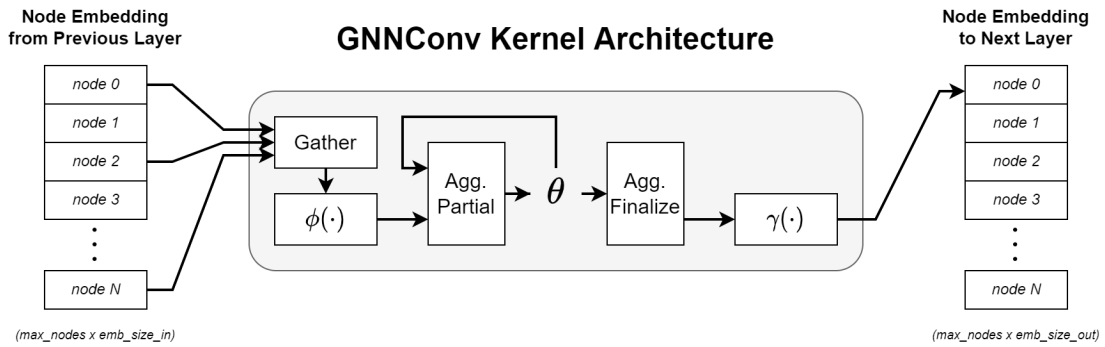

Figure 3.3: The high-level hardware kernel architecture for GNNConv layers.

offset table. Then the kernel iterates through each neighbor index to load its associated embedding from the input node embedding table, transform the embedding defined by $\phi(\cdot)$, and aggregate it with a partial aggregation. Once all the neighbors have been processed, the partial aggregation is finalized and combined with the current node embedding to be transformed again in the apply function $\gamma(\cdot)$. The newly computed embedding is then written to the output node embedding table.

The functions $\phi(\cdot)$ and $\gamma(\cdot)$, as well as the aggregation(s) used, are specific to the layer being implemented. Kernels for GCN, GraphSAGE, GIN, and PNA layers are provided as part of the initial GNNBuilder kernel library.

It is important to note that for each node's gather operation, a kernel needs to iterate at a number of cycles equal to the node's in degree. Consequently, the number of transforms and partial aggregation depends on the number of neighboring nodes, which is also only known at runtime. Additionally, our approach does not have any node-level parallelism for computing new node embeddings. All these details result in sub-optimal performance, which can be explored in future works as a key area for optimization with ideas borrowed from new accelerator works.

3.3.4 Partial Aggregations

Aggregations must be designed in a partial manner to efficiently aggregate neighbor embedding using constant memory ($O(1)$ space complexity). If it is assumed that a value is only seen once during an aggregation, then it does not need to be buffered. This is important in this context because it is optimal not to store all the neighbor embedding in an intermediate buffer. Since the number of neighbors of each node is not known at compile time, this buffer would have to be of size MAX_NODES, which can consume a large amount of BRAMs.

To avoid this buffer, a single-pass algorithm can be used where the aggregation is computed by looking at each value only once. However, for an aggregation like variance, a trivial approach may require two passes over all the values requiring them to be buffered. However, a one-pass algorithm for computing variance is possible using Welford's algorithm [44].

GNNBuilder supports sum, min, max, mean, variance, and standard deviation aggregations. Each aggregation has an associated $\mathrm{struct\ agg\_incremental\_data}$ for storing the partial and final aggregation data, a $\mathrm{agg\_incremental\_update(agg\_incremental\_data¡T¿\ \&data,\ T\ x)}$ function for updating the partial values of the aggregation, and a $\mathrm{agg\_incremental\_finalize(agg\_incremental\_data¡T¿\ \&data)}$ function for computing the final aggregation value.

3.3.5 Linear Layer

We implement linear / dense layers using tiled matrix multiplication. This allows us to exploit hardware parallelization to decrease latency at the cost of increased resource usage, mainly DSPs and BRAMs. The parallelization factor of each linear layer can be controlled by BLOCK_SIZE_IN and BLOCK_SIZE_OUT template arguments for the linear kernel function. These control the partition factors of the input, weight, and bias arrays and, thus, the parallelism of the multiply-accumulate (MAC) operations. This design allows

for BLOCK_SIZE_IN $\times$ BLOCK_SIZE_OUT multiply-accumulate operations to occur in parallel each clock cycle.

3.3.6 Global Pooling

GNNBuilder support sum, mean, and max global graph pooling. These pooling operations aggregate the node embeddings across all nodes resulting in a single embedding of the same size. Using more than one pooling method is also supported; the pooled embeddings from each pooling method are connected together. The implementation uses the same partial aggregations as described in Section subsection 3.3.4.

3.3.7 Activations

GNNBuilder support ReLU, Sigmoid, Tanh, and GELU [45] activations. These functions are implemented using fixed-point math functions provided by the Vitis HLS fixed-point math library. Extending support to other PyTorch activations to the GNNBuilder template library is trivial using the Vitis HLS fixed-point math functions.

3.4 Accelerator Generation and Implementation

Automated kernel generation is one of the key contributions and advantages over existing works. No existing works provide tooling to covert a native software model defined using an existing deep learning library to a hardware accelerator. GNNBuilder uses the PyTorch framework to define the initial GNN software model. The initial software model allows deep learning practitioners to construct, train, evaluate and inspect their models like any other software model. This also reduces development friction since practitioners don't have to use a customized and limited API for defining their initial hardware. GNNBuilder can then build sufficient code generation tools through dynamic introspection of the software model objects in combination with a templating system and pre-defined kernel library.

3.4.1 Kernel Code Generation

Once a user has defined a GNNModel in software with their chosen design parameters as well as a GNNBuilder project, GNNBuilder is able to perform code generation to generate C++ HLS code for the top-level model kernel and associated header file. The code generation itself is done using the Jinja2 Python library to template the C++ code. This support conditional and loop control flows for template blocks. This is useful for features such as code gen. for skip-connections, double-buffer array selection, and mapping layer kernel calls in the right order with the right input/output size.

GNNBuilder takes advantage of the parameterized structure of the GNNModel in order to match the appropriate function calls to the corresponding kernels from GNNBuilder C++ header-only template library. For example, a GCNConv layer class in Python can be matched to the associated gcn_conv function from the GNNBuilder C++ template library. We can also perform the same matching for activation and global pooling functions. The main advantage of this approach is its extensibility. If a user is interested in adding support for another layer, aggregation, or activation, they can create an associated kernel in the GNNBuilder template library, add that template matching into the Jinja template, and create a pull request to merge the contribution.

Additionally, GNNBuilder also takes advantage of PyTorch's existing functionality to gather all parameters of the model and their associated shape. This allows for the templating of all model-dependent parameters that can vary from model to model configuration.

3.4.2 Hardware Simulation and Verification Testbenches

GNNBuilder also provides the ability for a designer to generate and build C++ testbenches for their designed models. In addition to the testbench code itself, the model parameters, dataset graphs, true output, and the PyTorch model outputs are also generated and exported as binary files. At runtime, the generated testbench reads in these files, loads the weights into the model kernel, evaluates the model kernel on all the inputs in the dataset, and com-

pares the output to the PyTorch model outputs.

Each testbench also computes metrics for verification. The mean absolute error between the PyTorch generated model output and generated model kernel output is computed. The runtime of each kernel execution on an input graph is also measured to compute the average kernel runtime. At runtime, these values are written to text files.

When specifying a fixed-point model, a user can be sure that the fixed-point representations of the input graphs and model parameters are accurately reflected in the testbench. The testbench uses the same fixed-point library provided by Vitis HLS [46] to ensure functional equivalence to how they would be modeled in hardware. The loaded data is exported from PyTorch as a floating-point type but is cast to the user-specified fixed-point format in the testbench.

3.4.3 Hardware Deployment on FPGA

Users can use already introduced $\mathrm{run\_vitis\_hls\_synthesis()}$ function to build synthesised accelerator which is outputted as RTL source code. Within the Vitis HLS flow, GNNBuilder support executing the implementation flow to generate Vivado IP blocks (.zip) or Vitis Kernels (.xo). This simplifies the model designer's workflow allowing them to go from the SW model to fully implemented design all within GNNBuilder's framework.

GNNBuilder also has experimental support for fully implementing Vitis kernels for deployment on Vitis-supported devices such as CPU-FPGA platforms. This includes the full implementation and bitstream generation for Vitis kernels (.xclbin) as well as a templated host code testbench that can load the kernel onto the FPGA fabric and evaluate a graph dataset directly on-chip. This testbench is very similar to the C++ testbench previously discussed but uses Xilinx's runtime library, XRT, to interface with the FPGA from the host.

3.5 Performance Model and Design Space Exploration

3.5.1 Hardware Model Implementation Details

All of the models in our experiments were implemented for the Xilinx Alveo U280 FPGA accelerator. In our implementation, we target a 300 MHz clock frequency. We use the high-level synthesis (HLS) and FPGA implementation tools provided by Vitis HLS and Vivado in our framework. The HLS code generated by the framework is what is directly provided to the tooling for synthesis. The logs and reports from the tooling are also captured in order to gather implementation timing and resource usage for comparison to our analytical model.

3.5.2 Hardware Performance Model

To evaluate how effective runtime modeling can be used for DSE, we explore an analytical latency model, a compensation fit latency model, and a direct fit latency+BRAM model. Each model can be evaluated against a baseline of the post-synthesis latency and BRAM usage reported by Vitis HLS. When looking at resource usage modeling, we mainly focus on BRAM usage since that is the dominating resource that is most likely to violate resource constraints first on large models.

The analytical latency model is a manually designed performance estimate of inference on a single graph based on the implemented HLS kernels in the GNNBuilder kernel template library. Each kernel is assessed line by line, and the latencies of each of the associated operations are used to write a simple analytical model in Python to compute the total latency of that kernel. The compensation fit latency model is fit on a dataset of model configurations and aims to predict a compensation value to be added to the analytical model to match the reported post-synthesis latency. The direct-fit latency model is a random forest regressor that is fit on a dataset of model configurations and aims to directly predict reported post-synthesis latency numbers. Similarly, the direct fit BRAM model is directly fit

on a dataset of model configurations and reported post-synthesis BRAM usage values. We chose the random forest as the regression model through empirical testing; we found the random forest model avoided overfitting better than linear models, support vector machine models, and gradient boosting tree models.

The fitted models require a database of designs to be synthesized ahead of time to fit and distribute trained models. The possible number of model configurations is intractable to explore by brute force. However, by sparsely sampling the design space to build the database, the fitted models should be able to interpolate between the sampled design space to provide accurate estimates for unseen configurations.

3.5.3 Design Space Exploration

Model designers typically have to tune and evaluate models by hand to choose the optimal model configuration parameters for the best latency vs. accuracy vs. resource usage. To enable design space exploration (DSE), the user can use the runtime and BRAM models to quickly evaluate specific model configurations. This allows designers to integrate these estimates into their own workflow to build more complex co-design tools to jointly optimize SW and HW metrics in their training workflow.

For more straightforward DSE, we provide the user with random sampling and brute force approaches to explore design using the previously discussed models. All configurations can be evaluated quickly using the models for small configuration spaces. For a larger configuration space, a user is able to randomly sample a fixed number of configurations in order to evaluate a sparse random subset of the larger configuration space. When evaluating models, the designer can then rank models based on predicted latency and identity feasible configurations based on predicted BRAM usage.

Rather than having the user be required to run the HLS build for each design configuration they want to evaluate or create a dataset of design configurations, we provide serialized trained versions of direct fit models described in the previous section. Evaluat-

ing the trained models is on the order of milliseconds compared to running HLS synthesis, which is on the order of minutes. This reduction in performance prediction runtime allows users to build intelligent co-design tools for real-time optimization rather than just pure DSE tools. This opens up the possibility for train-time model sparsity + quantization and neural architecture search, among other ideas.

CHAPTER 4

EXPERIMENTAL SETUP

4.0.1 Analytical Performance Model

We evaluate the accuracy of our runtime and BRAM models by looking at their performance against a large database of 400 synthesized designs. We randomly sampled these designs from the following configuration space of model parameters:

```
QM9_DATASET = QM9(root="./tmp/QM9").index_select(list(range(1000)))
DATASET_IN_DIM = QM9_DATASET.num_features
DATASET_OUT_DIM = QM9_DATASET[0].y.ravel().shape[0]

AVG_NODES, AVG_EDGES = compute_average_nodes_and_edges(QM9_DATASET, round_val=
    True)
AVG_DEGREE = compute_average_degree(QM9_DATASET, round_val=True)

MAX_NODES = 600
MAX_EDGES = 600

CONVS = ["gcn", "gin", "pna", "sage"]
GNN_HIDDEN_DIM = [64,128,256]
GNN_OUT_DIM = [64,128,256]
GNN_NUM_LAYERS = [1,2,3,4]
GNN_SKIP_CONNECTIONS = [True, False]
MLP_HIDDEN_DIM = [64,128,256]
MLP_NUM_LAYERS = [1,2,3,4]

GNN_P_HIDDEN = [2,4,8]
GNN_P_OUT = [2,4,8]
MLP_P_IN = [2,4,8]
```

MLP˙P˙HIDDEN = [2,4,8]

For the fitted models, we evaluated a random forest regressor with 10 estimators for each model. Each model is assessed using the mean absolute percent error (MAPE) between the true post-synthesis metrics and predicted metrics. To evaluate overfitting, we performed a 10 K-Fold cross-validation (CV). The test MAPE for each fold is averaged to produce a final cross-validation MAPE.

4.0.2 Accelerator Performance Evaluation

We evaluated different model architecture configurations across a range of datasets to compare the performance of our hardware implementation against CPU and GPU implementations. We use the following implementations:

- **PyG-CPU**: A PyTorch Geometric CPU model
- **PyG-GPU**: A PyTorch Geometric GPU model
- **CPP-CPU**: A C++ floating point CPU model
- **FPGA-Base**: Proposed HW model with no parallelism
- **FPGA-Parallel**: Proposed HW model with parallelism

The CPU models were evaluated on an Intel Xeon Gold 6226R, and the GPU models were evaluated on an NVIDIA RTX A6000. The hardware models are implemented as discussed in section subsection 3.5.1.

For each implementation type, we explore the same fixed GNN model while changing which GNNConv is being used. We look at the following layers: GCN, GraphSAGE, GIN, and PNA.

We also analyze the performance of each implementation and model configuration across a range of datasets for graph-level tasks. We use the QM9, ESOL, FreeSolv, and Lipophilicity regressions datasets and the HIV classification dataset from the larger MoleculeNet dataset. We evaluate each baseline on a batch size of 1, noting that batch size 1 is

the only fair comparison for real-time inference graphs. Other works such as [13] and [47] also commonly compare exclusively with batch size 1.

The runtime for CPU and GPU implementations was computed by averaging the runtime of the first 1000 graphs of each dataset (if the dataset had fewer than 1000 graphs, the complete dataset was used). The latency of the FPGA implementations was gathered from the worst caste estimate provided by Vitis HLS after model synthesis. Since GNNBuilder allows the user to pass in estimates for the average number of nodes and edges in a graph as well as the average in-degree of nodes in a graph, this knowledge can be passed on to the synthesis tool to provide accurate latency estimates. This is done using the #pragma HLS loop_tripcount pragma, which is applied to loops that are bounded by the number of nodes and edges in a graph or the in-degree of a node.

For all models, we used the following architecture configuration:

```
agnn.GNNModel(
    graph_input_feature_dim=dim_in,
    graph_input_edge_dim=0,
    gnn_hidden_dim=128,
    gnn_num_layers=6,
    gnn_output_dim=64,
    gnn_conv=conv,
    gnn_activation=nn.ReLU,
    gnn_skip_connection=True,
    global_pooling=agnn.GlobalPooling(["add", "mean", "max"]),
    mlp_head=MLP(in_dim=64 * 3, out_dim=dim_out, hidden_dim=64, hidden_layers=4, activation
      =nn.ReLU),
    output_activation=None,
)
```

For the FPGA-Parallel implementations, the GCN, SAGE, and GIN models use the following parallelism factors: gnn_p_in=1, gnn_p_hidden=16, gnn_p_out=8, p_in=8, p_hidden=8, p_out=1. The PNA models use gnn_p_hidden=8 and gnn_p_out=8 in order to

fit the model on the device fabric. The FPGA-Parallel implementations are also synthesized with ¡16, 10¿ bit fixed-point data representations; this is also done to fit within the device fabric.

All FPGA-Base implementations have all parallel factors set to 1 and implement node features using ¡32, 16¿ bit fixed-point types.

CHAPTER 5

RESULTS

5.0.1 Analytical Performance Model

The results of fitting the latency and BRAM models on our database of generated designs are shown in Figure Figure 5.1. The key takeaway is that direct fit models are the most effective for performance modeling. The direct-fit latency model achieved a CV MAPE of 33.94%, which is significantly better than the analytical model with a CV MAPE of 425.10%. The direct-fit BRAM model achieved a CV MAPE of 22.41%. Figure Figure 5.1 indicates that the analytical model consistently underestimates the post-synthesis reported latency while the direct fit and compensated fit models are able to more accurately produce a better fit to the true latency values. These results show that a direct fit of models on a design database sparsely sampling a design configuration space is an effective and simple approach for performance modeling in GNNBuilder to enable rapid DSE.

5.0.2 DSE Exploration

To exemplify the speed up of direct fit models over standard evaluation of the HLS tool, we also analyze the performance estimate compute time for all 400 model configurations used to train the direct fit models. We present the results in Figure Figure 5.2, which can be viewed as a timeline of runs. All model calls for the direct fit models to finish in under a second, while all Vitis HLS synthesis runs finish in under two days. An average direct fit model call takes 2.5 ms, while an average Vitis HLS synthesis run takes 7.94 minutes. This difference is around 6 orders of magnitude, emphasizing real-time performance estimation of direct fit models.

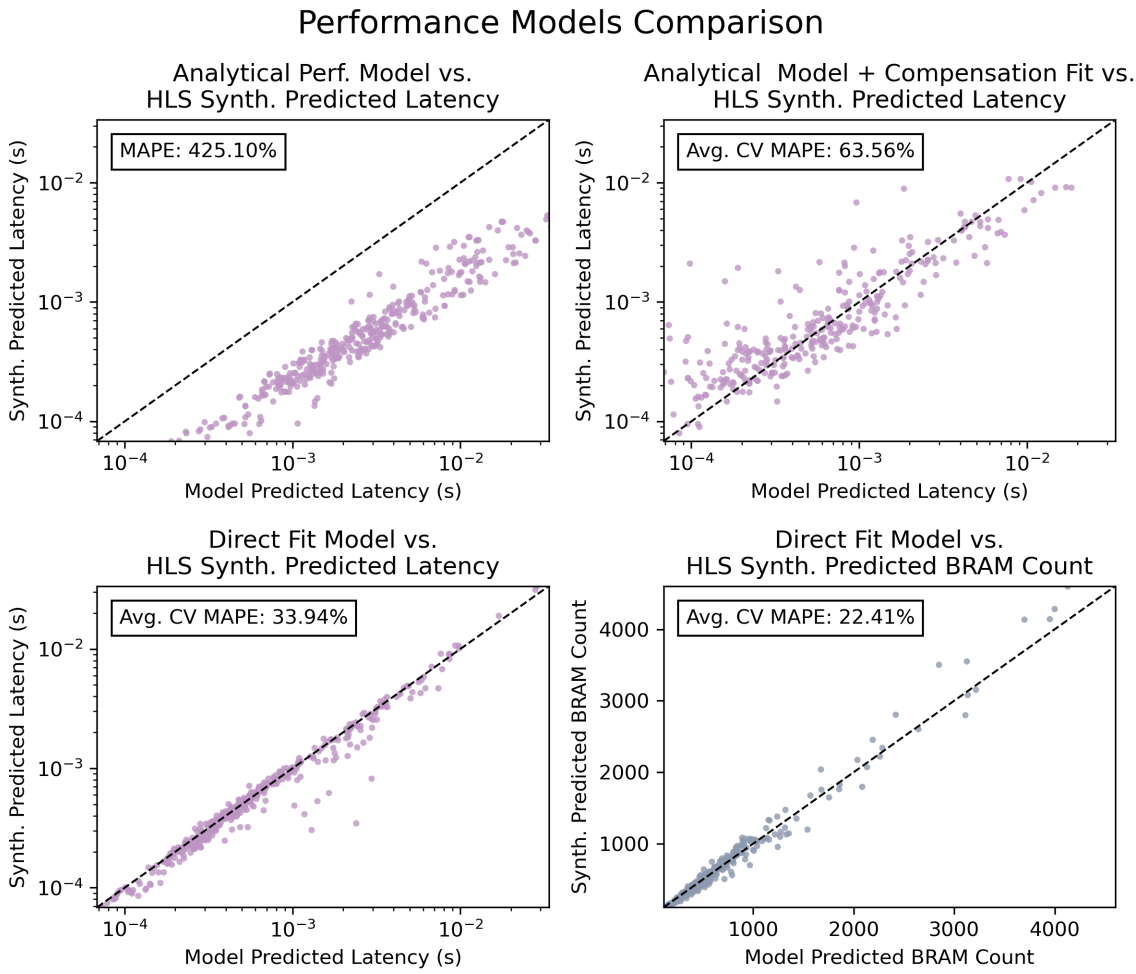

Figure 5.1: Comparison of latency prediction models with true post-synthesis latency and BRAM usage reported from Vitis HLS

5.0.3 Accelerator Performance Evaluation

The performance results for the proposed accelerator hardware framework in comparison to other implementations are shown in Figure Figure 5.3, Table Table 5.1, and Table Table 5.2. The values in Table Table 5.2 indicate the speedup factors of the FPGA-Parallel implementation for the latency values averaged across datasets. For all cases, there is at least a 2x speedup in the parallelized FPGA implementation over the PyG CPU, PyG GPU, and C++ CPU implementations. Across all models, there is a geometric mean speedup of **2.96**× over PyG-CPU and **2.99**× over PyG-GPU. The resource usage also shows more room for

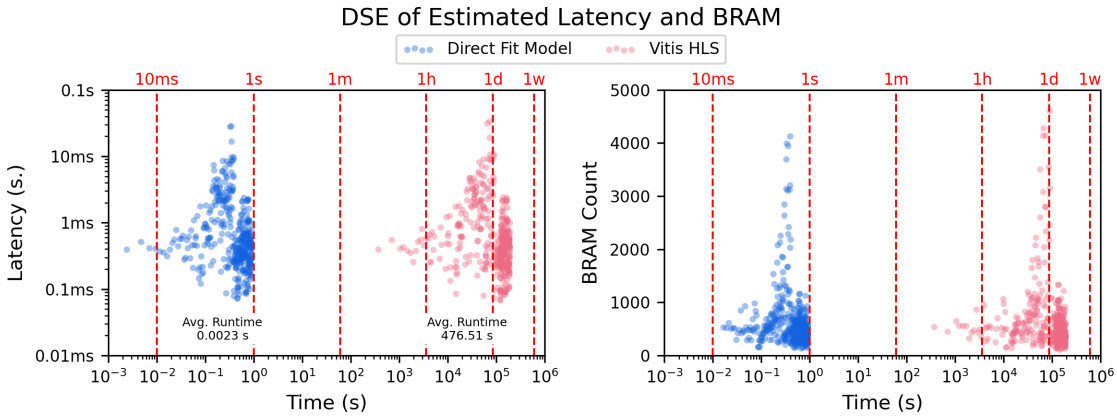

Figure 5.2: Cumulative runtime for evaluating 400 design configurations to predict mode latency and BRAM usage. The x-axis represents time going forward from left to right, and each point represents a performance estimate which has just finished computing. Note the log scale for the x-axis.

BRAM utilization across models indicating there is more room for increased parallelism and higher speedups.

Table 5.1: Resource usage of FPGA-Base and FPGA-Parallel model implementations as a percent of total resources available on the Xilinx Alevo U280 Accelerator Card.

	FPGA-Base				**FPGA-Parallel**			
	BRAM	DSP	FF	LUT	BRAM	DSP	FF	LUT
GCN	5.8%	0.5%	1.9%	5.5%	39.9%	6.2%	1.3%	8.9%
GIN	8.1%	0.2%	1.0%	2.8%	73.7%	11.6%	1.3%	13.1%
PNA	10.9%	0.7%	2.8%	6.7%	59.2%	6.9%	2.1%	10.4%
SAGE	7.9%	0.2%	1.4%	3.3%	69.9%	10.2%	1.3%	15.4%

Table 5.2: FPGA-Parallel speedup over PyG CPU, PyG GPU, and C++ CPU runtimes.

	PyG-CPU	PyG-GPU	CPP-CPU
GCN	3.80x	4.95x	1.99x
GIN	2.16x	2.24x	2.27x
PNA	3.02x	2.32x	9.68x
SAGE	3.09x	3.10x	6.37x
Geo. Mean	**2.96x**	**2.99x**	**4.09x**

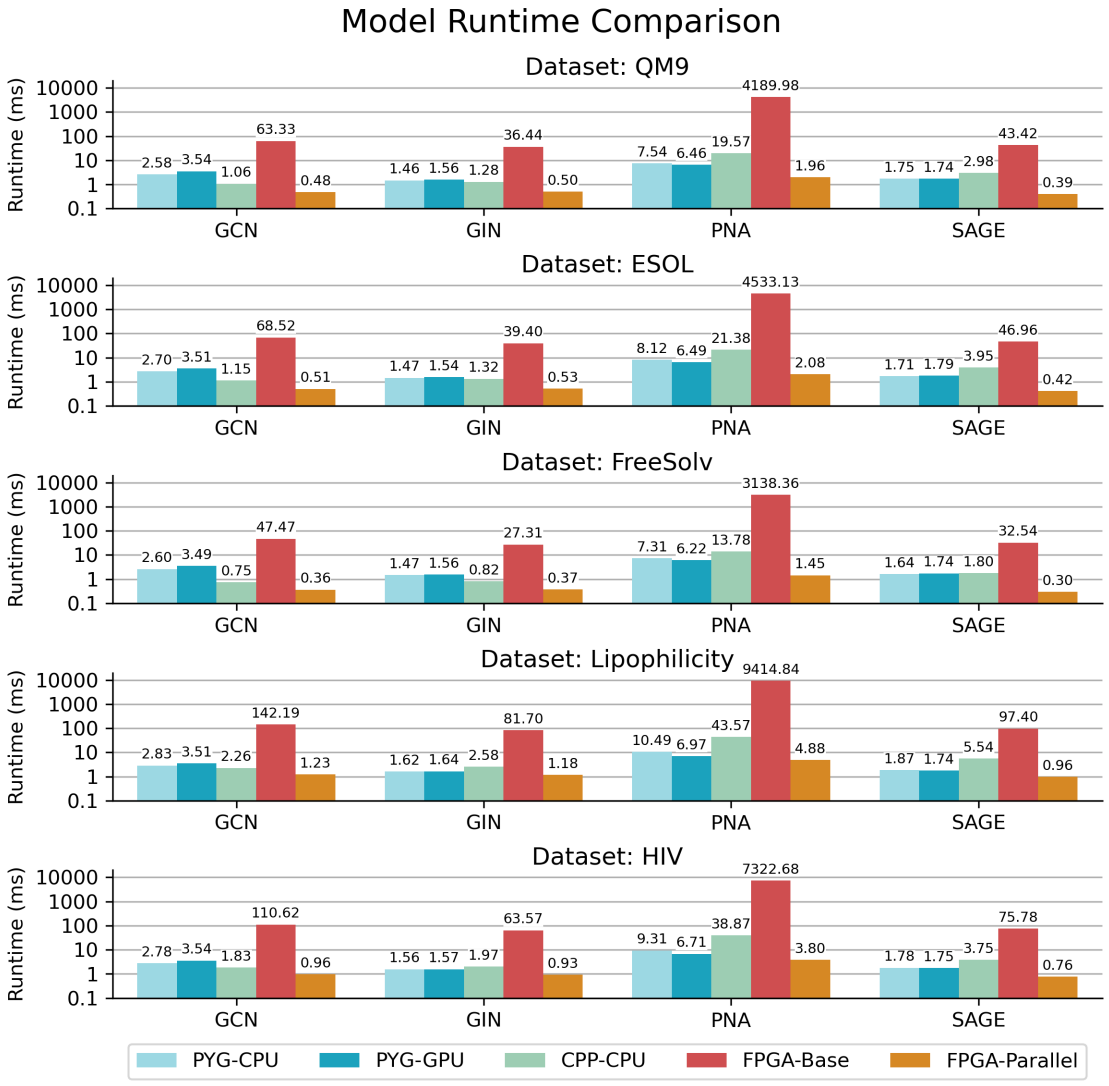

Figure 5.3: GNN model latency across a range of models, datasets, and implementations (Note that the y-axis is a log scale).

CHAPTER 6
CONCLUSION

We proposed **GNNBuilder**, the first automated, generic, end-to-end GNN accelerator generation framework. Due to its explicit message passing architecture, it supports a wide range of expressive GNNs, including anisotropic GNNs. We also provide an easy-to-use Python API that can interface directly with PyTorch modules to provide a complete design loop for end users and deep learning workflows. Additionally, GNNBuilder supports features not supported by most inference accelerator works, such as skip connections, global pooling, and MLP heads. Furthermore, we demonstrate the capabilities of GNNBuilder to generate hardware kernels and testbenches as well as to run testbenches on PyTorch Geometric datasets and launch Vitis HLS synthesis kernels. Moreover, we show that model configuration spaces can be sparsely sampled to fit latency and BRAM prediction models against post-synthesis metrics. These models can then be used by end users to perform simple DSE and develop more complex co-design workflows. Lastly, we demonstrate how our accelerator can achieve faster performance over CPU and GPU by parallelizing the hardware implementation.

Our initial software framework can be accessed at this repo:

https://anonymous.4open.science/r/gnn-builder.

As we clean up our codebase, we will update the repository and provide more end-use documentation to get started using the framework. Our goal is for this framework to be open-source and accessible to both software and hardware practitioners.

There are still many optimizations and avenues for future work branching off the proposed framework. One key area is to optimize dataflow and computation patterns in graph convolutions kernels in the GNNBuilder kernel template library to achieve greater speedups over CPU and GPU. This work does not explore HLS dataflow and HLS streaming opti-

mizations which can provide greater speedups and lower resource utilization across kernels. Exploring intelligent DSE search, such as genetic algorithms, mixed-integer non-linear optimization, and train-time co-design is another area for future research that can provide end users with even more powerful design tools to integrate into their workflow. Lastly, we hope to continue maintaining our framework and expand the kernel template library to add support for more graph convolution kernels, such as graph attention networks (GAT)[48] convolutions and other emerging GNN architectures.

REFERENCES

[1] Z. Wu *et al.*, "MoleculeNet: A benchmark for molecular machine learning," *Chemical Science*, vol. 9, no. 2, pp. 513–530, 2018.

[2] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, *Graph Neural Networks in Recommender Systems: A Survey*, arXiv:2011.02260 [cs], Apr. 2022.

[3] A. Benamira, B. Devillers, E. Lesot, A. K. Ray, M. Saadi, and F. D. Malliaros, "Semi-Supervised Learning and Graph Neural Networks for Fake News Detection," in *2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, ISSN: 2473-991X, Aug. 2019, pp. 568–569.

[4] A. Derrow-Pinion *et al.*, "ETA Prediction with Graph Neural Networks in Google Maps," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, arXiv:2108.11482 [cs], Oct. 2021, pp. 3767–3776.

[5] S. N. Golmaei and X. Luo, "DeepNote-GNN: Predicting hospital readmission using clinical notes and patient network," in *Proceedings of the 12th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, ser. BCB '21, New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1–9, ISBN: 978-1-4503-8450-6.

[6] X. Chang, P. Ren, P. Xu, Z. Li, X. Chen, and A. Hauptmann, *A Comprehensive Survey of Scene Graphs: Generation and Application*, arXiv:2104.01111 [cs], Jan. 2022.

[7] N. Wu, Y. Xie, and C. Hao, "IronMan-Pro: Multi-objective Design Space Exploration in HLS via Reinforcement Learning and Graph Neural Network based Modeling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2022, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[8] L. Wu *et al.*, *Graph Neural Networks for Natural Language Processing: A Survey*, arXiv:2106.06090 [cs], Jun. 2021.

[9] Y. Lin, Z. Zhang, H. Tang, H. Wang, and S. Han, "PointAcc: Efficient Point Cloud Accelerator," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, arXiv:2110.07600 [cs], Oct. 2021, pp. 449–461.

[10] A. Elabd *et al.*, "Graph Neural Networks for Charged Particle Tracking on FPGAs," *Frontiers in Big Data*, vol. 5, p. 828 666, Mar. 2022, arXiv:2112.02048 [hep-ex, physics:physics, stat].

[11] Y. Li *et al.*, *Deep Learning for LiDAR Point Clouds in Autonomous Driving: A Review*, arXiv:2005.09830 [cs], May 2020.

[12] A. Elabd *et al.*, "Graph neural networks for charged particle tracking on FPGAs," *Frontiers in Big Data*, vol. 5, p. 828 666, Mar. 2022.

[13] H. Qu and L. Gouskos, "Jet tagging via particle clouds," *Physical Review D*, vol. 101, no. 5, p. 056 019, Mar. 2020.

[14] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[15] M. Yan *et al.*, "HyGCN: A GCN accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, San Diego, CA, USA: IEEE, Feb. 2020, pp. 15–29, ISBN: 978-1-72816-149-5.

[16] S. Liang *et al.*, "EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1511–1525, Sep. 2021.

[17] T. Geng *et al.*, "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Athens, Greece: IEEE, Oct. 2020, pp. 922–936, ISBN: 978-1-72817-383-2.

[18] 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)*, IEEE, 2021, pp. 29–39.

[19] T. Geng *et al.*, "I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, Virtual Event Greece: ACM, Oct. 2021, pp. 1051–1063, ISBN: 978-1-4503-8557-2.

[20] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Seoul, Korea (South): IEEE, Feb. 2021, pp. 775–788, ISBN: 978-1-66542-235-2.

[21] X. Chen *et al.*, "Rubik: A hierarchical architecture for efficient graph neural network training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 4, pp. 936–949, Apr. 2022.

[22] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside CA USA: ACM, Feb. 2020, pp. 255–265, ISBN: 978-1-4503-7099-8.

[23] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2016.

[24] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[25] S. A. Tailor, F. Opolka, P. Lio, and N. D. Lane, "Do we need anisotropic graph neural networks?" In *International Conference on Learning Representations*, 2021.

[26] K. Xu *et al.*, "How powerful are graph neural networks?" In *ICLR*, 2019.

[27] G. Corso *et al.*, "Principal neighbourhood aggregation for graph nets," in *NeurIPS*, 2020.

[28] S. Liang, C. Liu, Y. Wang, H. Li, and X. Li, "DeepBurning-GL: An Automated Framework for Generating Graph Neural Network Accelerators," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, ISSN: 1558-2434, Nov. 2020, pp. 1–9.

[29] Y.-C. Lin, B. Zhang, and V. Prasanna, "Hp-gnn: Generating high throughput gnn training implementation on cpu-fpga heterogeneous platform," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 123–133.

[30] P. Veličković, *Message passing all the way up*, 2022.

[31] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Comput. Surv.*, vol. 54, no. 9, Oct. 2021.

[32] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA: IEEE, Jul. 2020, pp. 1–6, ISBN: 978-1-72811-085-1.

[33] K. Kiningham, C. Re, and P. Levis, "GRIP: A graph neural network accelerator architecture," *arXiv:2007.13828 [cs]*, Jul. 2020. arXiv: 2007.13828 [cs].

[34] S. Abi-Karam, Y. He, R. Sarkar, L. Sathidevi, Z. Qiao, and C. Hao, "Gengnn: A generic fpga framework for graph neural network acceleration," *arXiv preprint arXiv:2201.08475*, 2022.

[35] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, *FlowGNN: A Dataflow Architecture for Universal Graph Neural Network Inference via Multi-Queue Streaming*, arXiv:2204.13103 [cs], Apr. 2022.

[36] G. Dai *et al.*, "GraphH: A processing-in-memory architecture for large-scale graph processing," *IEEE TCAD*, vol. 38, no. 4, pp. 640–653, 2018.

[37] D. Yan *et al.*, "Blogel: A block-centric framework for distributed computation on real-world graphs," *VLDB*, vol. 7, no. 14, pp. 1981–1992, 2014.

[38] Y. Tian *et al.*, "From" think like a vertex" to" think like a graph"," *VLDB*, vol. 7, no. 3, pp. 193–204, 2013.

[39] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 217–226.

[40] Z. Shao, R. Li, D. Hu, X. Liao, and H. Jin, "Improving performance of graph processing on fpga-dram platform by two-level vertex caching," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 320–329.

[41] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "Hitgraph: High-throughput graph processing framework on fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[42] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–12.

[43] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "Thundergp: Hls-based graph processing framework on fpgas," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 69–80.

[44] B. P. Welford, "Note on a Method for Calculating Corrected Sums of Squares and Products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962, Publisher: [Taylor & Francis, Ltd., American Statistical Association, American Society for Quality].

[45] D. Hendrycks and K. Gimpel, *Gaussian Error Linear Units (GELUs)*, arXiv:1606.08415 [cs], Jul. 2020.

[46] Vitis, *Vitis high-level synthesis user guide (ug1399)*, https://docs.xilinx.com/r/en-US/ug1399-vitis-hls, Accessed: 2021.

[47] Groq, Inc., "The challenge of batch size 1: Groq adds responsiveness to inference performance," p. 7, 2020.

[48] P. Veličković *et al.*, "Graph attention networks," in *arXiv preprint arXiv:1710.10903*, 2017.