Implementation of GNN Convolutional layers for GNNBuilder Framework

Harigovind Anil
ECE, Georgia institute of technology
Atlanta, USA
hanil3@gatech.edu

Nagasayee Guduru Gopalakrishna
ECE, Georgia institute of technology
Atlanta, USA
ngg7@gatech.edu

Mentor - Stefan Abi-Karam
ECE, Georgia institute of technology
Atlanta, USA
stefanabikaram@gatech.edu

***Abstract*—With the increase in the use of FPGAs to accelerate Machine Learning models, our projct aims at implementing different convolution layers in HLS and integrate them into GNNBuilder framweork which automates the generation of accelerated Machine Learning model on FPGA based on the input pytorch Machine learning model as input.**

***Index Terms*—ML, FPGA, HLS, GNN**

I. Introduction

Graph Neural Networks (GNNs) are a class of deep learning methods designed to perform inference on data described by graphs. GNNs can be directly applied to graphs and provide an easy way to do node-level, edge-level, and graph-level prediction tasks. GNNs can be used for various applications such as drug discovery, recommender systems, social network analysis, traffic forecasting, electronic health records analysis, scene graph understanding, electronic design automation, natural language processing, autonomous driving, and high-energy physics. 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. Another prominent example is in high-energy physics, where GNNs are used for real-time particle detection. Given the acceleration needs for GNN inference, there are many GNN accelerators which are model specific but not generic.

GNNBuilder is an automated, end-to-end GNN accelerator generation framework that can automatically generate GNN accelerators for a wide range of GNN models arbitrarily defined by users. It supports end-to-end code generation, simulation, accelerator optimization, and hardware deployment.

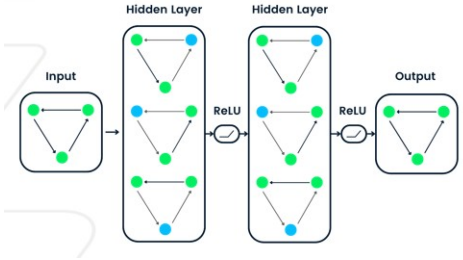

Fig. 1. Typical Representation of a GNN model

II. Problem Description

GNNBuilder Framewrk uses the the convolution layer code written in C++ HLS as a template to generate the accelerated ML model. Currently, few layers of convolution have been implemented in the framework. Our project is to convert the PyTorch GNN convoluion layers to C++ HLS libraries that can be used by GNNBuilder Framework. We will also optimize dataflow and computation patterns in graph convolutions kernels to achieve greater speedups over CPU and GPU.

III. Related Work

GNN can capture the dependance of graphs and are used in applications where CNN fails. Applying CNN on a graph poses challenges due to the graph's arbitrary size and complex topologies. GNN has been widely used in applications such as social influence prediction, healthcare, Natural Language Processing (NLP), and traffic movement forecast. It is also increasingly being used for image classification and life sciences as well.

Some of the existing work in the GNN accelerator domain are DeepBurning-GL [1] that includes FPGA accelerators for GCN-based architectures, and HP-GNN [2] that also supports GCN-based architectures. However, most of the implementations are application-specific and not generic, and requires extensive hardware expertise to adapt to new GNN models. GNNBuilder aims to tackle these issues by providing a generic, feature-rich extensible framework that is built using PyTorch as the programming language. It aims to design and implement optimized GNN models from standard PyTorch models. An optimized FPGA bitstream is generated as the output.

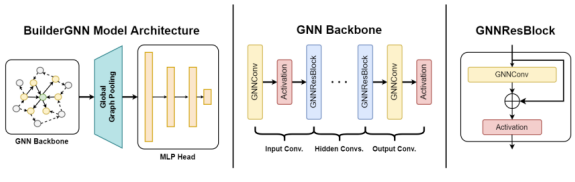

Fig. 2. GNNBuilder Model Architectue

The GNNBuilder framework consists of five layers, namely: Compiler front-end, Code generator, Design space exploration

and performance model, Simulation and testbench, Hardware synthesis and deployment.

A. Compiler front-end

This layer parses the parameters of native PyTorch GNN model like number of GNN layers, layer type, activation type, data precision, pooling type, aggregation type, and MLP definition.

B. Code generator

This layer uses the pre-defined hardware accelerator templates to generate the High-Level Synthesis (HLS) code based on GNN model parameters passed targeting to the specific FPGAs.

C. Design space exploration and performance model

This layer applies automated DSE for the accelerator generation, including hardware parallelism, resource allocation, and quantization (data precision).

D. Simulation and testbench

This layer allows transparent hardware-compatible simulation using automatically generated testbenches to guarantee the correctness of the accelerator functionality. It also generates plain C++ code for "true" quantization simulation, that reflects on-FPGA quantization accuracy.

E. Hardware synthesis and deployment

This layer automatically generates hardware synthesis scripts, synthesizes the design into an FPGA bitstream, and generates host code for executing the bitstream.

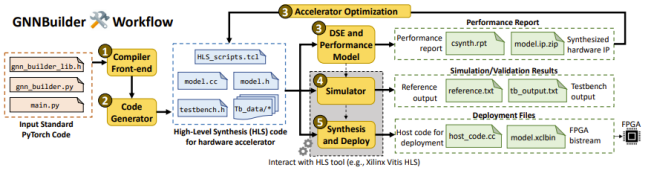

Fig. 3. GNNBuilder framework

IV. Proposed Solution

We aim to create an optimized synthesizable HLS C++ library for two chosen GNN convolution PyTorch layer. The two PyTorch Convolution layers are SimpleConv and LGConv. SimpleConv is a simple message passing operator that performs (non-trainable) propagation. LGConv is a Light Graph Convolution operator from [3].

Simple Convolution (SimpleConv) is used as a simple message passing operator that performs propagation from one layer to another in a complex Convolution model. A custom aggregation scheme is used to perform the propagation which could be "add", "sum" "mean", "min", "max" or "mul".

$$\text{SimpleConv} : \mathbf{x}'_i = \bigoplus_{j \in \mathcal{N}(i)} e_{ji} \cdot \mathbf{x}_j$$

where $\bigoplus$ defines a custom aggregation scheme.

Graph Convolution Network (GCN) has become new state-of-the-art for collaborative filtering. But it was found that feature transformation and nonlinear activation which are the two most common designs in GCNs contribute very less to the collaborative filtering performance. These two designs also increases the training difficulty. Light Graph Convolution (LGConv) aims at simplifying this design by including neighborhood aggregation which is the most essential component for collaborative filtering.

$$\text{LGConv} : \mathbf{x}'_i = \sum_{j \in \mathcal{N}(i)} \frac{e_{j,i}}{\sqrt{\deg(i)\,\deg(j)}} \mathbf{x}_j$$

Once the convolution layers are implemented in C++, the same is generated in PyTorch to be used as a golden reference. The generated output is then compared against the golden reference to verify functional correctness.

V. Execution

The project is split into several steps from literature review and understanding of GNNBuilder framework to porting the layers mentioned in the previous section and optimizations. We will start with understanding the existing architecture of GNNBuilder and the layers that are ported and optimized already. We will then proceed with porting the SimpleConv and LGConv convolution layers to HLS C++. This step also involves functional correctness verification with the PyTorch model that is present already. The next stage is checking the synthesizability of the code that is written and to perform dataflow optimizations. Finally, the code will be optimized and the final deliverables will be generated.

We will split the task equally between the team members where each one will be assigned a convolution model to port and optimize.

A. GNNBuilder Architecture

GNNBuilder architecture has the framework for writing the C++ template library for the PyTorch libraries, the python testbench to verify the functionality of the library, and a template generator to synthesize and verify the operation on an FPGA.

- The C++ library is present under the gnn_builder/gnn_builder_lib folder. Our implementation of the convolution layers in C++ are integrated to this C++ library file.
- The testbench, both Python code to generate golden sample and C++ code to verify the functionality, are present under gnn_builder/gnn_builder_lib_test folder. Pytorch models of the convolution layers we have implemented are integrated into the python testbench file that generates

the golden reference outputs for the respective convolution layers.
- To synthesize the library code that is built, a Jinja framework is used to build the synthesizable code based on the pragmas used for an application by specifying the graph parameters. demos.py that is specified under demos folder accesses the template files present in the templates folder. The header file template (model.h.jinja), C++ template (model.cpp.jinja), makefile template for HLS (makefile_testbench.jinja), script to run HLS TCL (run_hls.tcl.jinja), and the testbench template (makefile_testbench.jinja) are all specified in jinja format. The FPGA parameters are specified under code_gen.py file.

B. Creating the library file

The first step in creating the GNNBuilder framework is writing the C++ implementations for the PyTorch libraries. Two functions are implemented in both LG_Conv and SimpleConv – the convolution function and the aggregate function. The <convolution>_conv function sets up the graph parameters such as the number of edges, number of nodes, input degree table, output degree table, input and output embedding sizes, neighboring nodes data. For each node that is present in the graph, <convolution>_conv function calls the aggregate function. The <convolution>_agg is where the actual implementation of the convolution for each node is written.

The aggregate function is implemented with the PyTorch implementation as reference. Both LGConv and SimpleConv do not support a linear function. LGConv supports only summation operator whereas the SimpleConv implementation supports an aggregation function that could be add, sum, mean, min, max, or mul operations. The aggregation function is implemented with the graph data obtained from the convolution function and the corresponding operations are added.

C. Python file for golden sample

Once the library file is implemented, it is necessary to test the functionality and correctness of the library function. This is tested against the standard PyTorch implementation of the function. The graph data required, i.e. edge data and nodes data are generated using np.random. The maximum edges and nodes are specified within this code. SimpleConv and LGConv do not require any linear weights or bias since they do not perform any linear operations on the graph data. The generated graph parameters are then stored in binary files so that the C++ implementation can reuse the same parameters. The stored parameters are -

- The maximum number of edges
- The maximum number of nodes
- The number of edges
- The number of nodes
- The COO matrix, that is the matrix is represented as a set of triples, where x is an entry in the matrix and i and j denote its row and column indices, respectively.
- The in degree table
- The out degree table
- The neighbor table
- The input node feature size
- The input node features
- The output feature size
- The hidden feature size (For both the layers implemented, the feature sizes are the same for input and output)
- The actual output from the implementation of the library functions

The actual implementation of LGConv and SimpleConv (or any other convolution function) is specified under models.py. A class is created for LGConv_GNNB and SimpleConv_GNNB. The PyTorch implementation and the forward function is defined here to be used in the testbench. Based on the graph parameters specified, the graph network is created and the convolution outputs are calculated. This is then stored into a binary file to be used as the golden output. Two additional files are created in this implementation - tb_lgconv_output.bin for LGConv output and tb_simpleconv_output.bin for SimpleConv output.

D. C++ Testbench

A parallel to the python testbench is written in C++ that computes the convolution from the library created. The graph parameters are read from the binary files generated by Python. The graph parameters are passed on to the library file and the convolution output is obtained. The output from the Python implementation is also read and used as the golden sample. The maximum permitted error is limited to 0.001 between the values obtained in Python and C++. This is tested for both fixed point as well as floating point values where the fixed point provides an estimate of the values computed in a FPGA and the floating point computes the values in a general system and provides the functional correctness of the implemented library.

E. FPGA Optimizations

There were two HLS pragmas that were predominantly used in this design - dataflow and stable. #pragma HLS stable [6] is applied to arguments of a dataflow or pipeline region and is used to indicate that an input or output of this region can be ignored when generating the synchronizations at entry and exit of the dataflow region.

```
void dataflow_region(int A[...], int B[…] ...
#pragma HLS stable variable=A
#pragma HLS dataflow
    proc1(...);
    proc2(A, ...);
```

Fig. 4. Stable pragma operation

#pragma HLS dataflow enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation and increasing the overall throughput of the design [6].

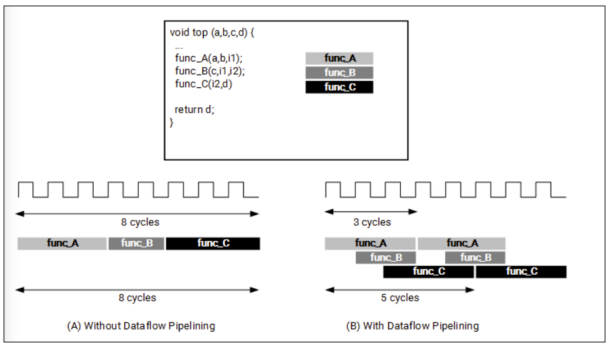

Fig. 5. Dataflow pragma operation

In the example above, without the STABLE pragma, proc1 and proc2 would be synchronized to acknowledge the reading of their inputs (including A). With the pragma, A is no longer considered as an input that needs synchronization.

```
for (int node = 0; node < num_nodes; node++) {
    #pragma HLS loop_tripcount min = 0 max = NUM_NODES_GUESS

    #pragma HLS DATAFLOW
    #pragma HLS stable variable=edge_list
    #pragma HLS stable variable=neightbor_table_offsets
    #pragma HLS stable variable=neighbor_table
    #pragma HLS stable variable=in_degree_table
    #pragma HLS stable variable=out_degree_table

    T current_node_embedding_in[EMB_SIZE_IN];
    for (int i = 0; i < EMB_SIZE_IN; i++) {

    int num_in_neighbors = in_degree_table[node];
    int neighbors[MAX_NODES];

    gather_node_neighbors<
        MAX_NODES,
        MAX_EDGES,
        NUM_NODES_GUESS,
        NUM_EDGES_GUESS,
        DEGREE_GUESS
    >(

    T agg_embedding[EMB_SIZE_IN];
    simple_conv_agg<
        MAX_NODES,
        MAX_EDGES,
        EMB_SIZE_IN,
        EMB_SIZE_OUT,
        T,
        NUM_NODES_GUESS,
        NUM_EDGES_GUESS,
        DEGREE_GUESS,
        P_IN,
        P_OUT
    >
    (

    for (int i = 0; i < EMB_SIZE_OUT; i++) {
}
```

Fig. 6. Implementation example

In our convolution layers implementation, we used two for loops to copy input and output data from two functions namely gather_node_neighbors() and <convolution>_agg(). The output of the gather_node_neighbors() is provided as input to the <convolution>_agg(). This producer consumer relationship is pipelined by DATAFLOW pragma. The stable pragma is used to mention the variables that are not altered in this operation to ensure concurrent execution of loop iterations. These variables include the edge list, the neighbor table offsets, neighbor table itself, the in degree table and the out degree table. All these are parameters obtained from the binary file generated by the python script and are not impacted by the loop iterations.

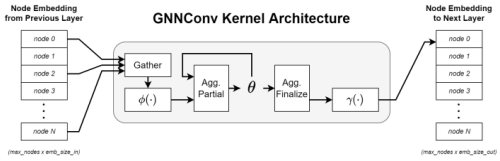

Fig. 7. GNN architecture

F. Jinja templates

Jinja is a templating engine that allows adding placeholders for data elements similar to Python code. The template is then passed the data and the final document is rendered. This templating engine is used to generate the synthesizable code for FPGAs based on the libraries created. The graph parameters are passed on as template data in this implementation.
The code is tested on an FPGA by passing the graph parameters extracted from MoleculeNet dataset. The number of classes, features and the edge and node data are extracted from this and passed on to the Jinja template. The file demo.py extracts this information and passes it on to the template. The template interface is done in code_gen.py file. From the templates defined, demo.py generates the following -

- Hardware model from the model.h.jinja template
- The testbench based on model_tb.cpp.jinja template
- The makefile for building the project based on makefile_testbench.jinja
- The VITIS HLS TCL script for running the project and specifying the runtime parameters
- The VITIS makefile for building the project and report generation. This is specified in makefile_vitis.jinja

VI. Proposed Timeline

Timeline	
March 13	Literature Review and understanding GNNBuilder
March 27	Porting Convolution layers 1 and 2 to C++
April 10	Check synthesizability and perform dataflow optimizations
April 24	Optimizations and final deliverables

VII. Milestones Achieved

- The structure of GNNBuilder contains the C++ implementation of the PyTorch libraries in the gnn_builder_lib\gnn_builder_lib.h file.
- A total of four functions were implemented in this file - lg_conv_agg, lg_conv, simple_conv_agg, and simple_conv which contains the C++ implementation of the pytorch models of the LG Convolution and Simple convolution layers.
- These functions are implemented as template functions to provide compatibility with the functions that are present already in the GNN Builder architecture.
- The aggregate functions are responsible for performing the convolution operation at every node. All the neighboring nodes are determined for the calculation and the custom aggregation function is also applied in this step. The aggregate function is invoked by the lg_conv, and simple_conv for each node present in the graph.
- GNNBuilder also provides a framework for testing the library ports written in the library. The test suite is present in gnn_builder_lib_test directory.
- Here, a python implementation is present to be used as the golden sample. This python file is gen_test_data.py where all the convolution functions are present and are implemented from PyTorch. We have integrated the PyTorch models of LGConvolution and SimpleConvolution layers here which are used for generating the golden sample data.
- The golden sample outputs are written to a binary file. The layers implemented are then tested with a Cpp file that contains a function to test the layers by invoking an instance of the library which contains the C++ implementation of the convolution layers and is run with the same graph network that was used to generate the golden sample outputs.
- The values obtained with this is compared against that of the golden samples with a maximum error of 0.001 to ensure the functionality of the implementation of both the convolution layers.
- The C++ libraries have been ported and the functionality is tested using the testbench and was found to be passing for all cases.

VIII. Future Work

- The library containing the C++ implementation of the LGConv and SimpleConv layers needs to be synthesized and then implemented on an FPGA.
- The jinja template for the code needs to be updated for the newly implemented layers so that it can run all the scripts and generates the bitstream using Vivado and Vitis HLS tools. This bitstream will then be used to test the functionality of the implemented layers on an FPGA board.
- The unoptimized implementation of these layers are to be synthesized and then run on an FPGA to understand the resource usage and time required to run these layers.
- The optimized implementation with the dataflow pragmas be synthesized and then run on same FPGA to understand the resource usage and time required to run these layers and compare them with the unoptimized implementaiton results
- Additional convolution libraries need to be added to ensure complete functionality of the C++ implementation.

References

[1] 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.

[2] 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.

[3] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang, "LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation"

[4] Stefan Abi-Karam, Yuqi He, Rishov Sarkar, Lakshmi Sathidevi, Zihang Qiao, and Cong Hao , "GenGNN: A Generic FPGA Framework for Graph Neural Network Acceleration"

[5] Stefan Abi-Karam, "GNNBuilder: An Automated Framework For Generic Graph Neural Network Accelerator Generation, Simulation, And Optimization"

[6] https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas