POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

An FPGA Toolchain for Graph Neural Network Acceleration using High-Level Synthesis

Master of Science Thesis in
Computer Science and Engineering

Author: **Giovanni Demasi**

Student ID: 987062
Advisor: Prof. Fabrizio Ferrandi
Co-advisors: Serena Curzel, Michele Fiorito
Academic Year: 2022-23

Abstract

Graph Neural Networks are a class of Machine Learning models that have emerged as an efficient approach to dealing with graph-structured data, encompassing domains ranging from social networks to molecular chemistry and more. Particularly in the contemporary era of Big Data, dedicated hardware accelerators are often required to achieve optimal computational performance when processing large amounts of information. FPGAs offer a promising solution due to their inherent parallelism, but the process of translating Graph Neural Network models into FPGA accelerators is complex and requires extensive knowledge and expertise. This thesis addresses the challenge of accelerating Graph Neural Networks on FPGA, introducing a comprehensive toolchain that simplifies the process of transitioning from the PyTorch high-level framework to synthesized hardware accelerators leveraging High-Level Synthesis. Torch-MLIR is employed to produce the MLIR representation of the GNN model, which serves as input for the synthesizer. Here, fine-tuned optimizations can be applied before generating the ultimate GNN accelerator, ready to enhance inference performance on FPGA architectures. Experimental results demonstrate the efficacy of the toolchain, confirming substantial improvements in both performance and resource utilization. This accomplishment became possible through the identification of model bottlenecks and a study on optimizing matrix multiplication operations, which resulted to be a critical component of GNN computations. The thesis also explores synthesis optimizations and their impact on matrix multiplication performance and overall GNN model efficiency. In conclusion, this thesis represents a significant advancement in the domain of FPGA-accelerated GNN models. By developing an accessible and versatile toolchain and exploring synthesis optimizations, the research sets the stage for more efficient and widely accessible FPGA-accelerated GNN implementations.

Keywords: Graph Neural Network, Hardware Acceleration, High-Level Synthesis, FPGA, Synthesis Methodology

Abstract in Lingua Italiana

Le Graph Neural Networks sono una classe di modelli di Machine Learning e rappresentano l'approccio predefinito nel trattamento dei dati strutturati a grafo, comprendendo domini che vanno dai social networks alla chimica molecolare e oltre. Particolarmente nell'era contemporanea dei Big Data, sono spesso necessari acceleratori hardware dedicati al fine di ottenere prestazioni di calcolo ottimali durante l'elaborazione di grandi quantità di informazioni. Le FPGA offrono una soluzione promettente grazie al loro parallelismo intrinseco, ma il processo di traduzione dei modelli di Graph Neural Networks in acceleratori su FPGA è complesso e richiede una vasta conoscenza ed esperienza. Questa tesi affronta la sfida dell'accelerazione delle Graph Neural Networks su FPGA, introducendo una toolchain completa che semplifica il processo di transizione dal framework di alto livello PyTorch agli acceleratori hardware sintetizzati mediante High-Level Synthesis. Torch-MLIR consente la generazione della rappresentazione MLIR del modello, utlizzata come input per il sintetizzatore. In quest'ultima fase è possibile applicare diverse ottimizzazioni prima della generazione dell'acceleratore, migliorando in modo mirato le prestazioni di inferenza sulle architetture FPGA. I risultati sperimentali dimostrano l'efficacia della toolchain, confermando miglioramenti sostanziali sia nelle prestazioni che nell'utilizzo delle risorse. Questo risultato è stato possibile grazie all'individuazione dei colli di bottiglia del modello e allo studio dell'ottimizzazione delle operazioni di moltiplicazione matriciale, che si sono rivelate una componente fondamentale delle computazioni delle GNNs. La tesi esplora anche diverse ottimizzazioni di sintesi, analizzandone il loro impatto sulle prestazioni della moltiplicazione matriciale e sull'efficienza complessiva del modello di GNN. In conclusione, questa tesi rappresenta un progresso significativo nel campo dei modelli GNN accelerati da FPGA. Sviluppando una toolchain versatile ed esplorando le ottimizzazioni di sintesi, la ricerca pone le basi per implementazioni di GNN accelerate su FPGA più efficienti.

Keywords: Graph Neural Network, Accelerazione Hardware, High-Level Synthesis, FPGA, Metodologia di sintesi

Contents

Abstract **i**

Abstract in Lingua Italiana **iii**

Contents **v**

1 Introduction **1**
1.1 Contributions 2
1.2 Thesis structure 3

2 Background **5**
2.1 Graphs 5
2.1.1 Graph Representation 6
2.2 Graph Neural Networks 8
2.2.1 Multi-Layer Perceptron 11
2.2.2 Graph Convolutional Network 11
2.2.3 Graph Isomorphism Network 12
2.3 MLIR 13
2.4 High-Level Synthesis 14
2.5 LLVM 15
2.6 SODA Toolchain 15
2.6.1 SODA-OPT Frontend 16
2.6.2 SODA Synthesizer Backend 16
2.7 Conclusion 16

3 Related Work **19**
3.1 Chapter structure 19
3.2 Software frameworks 19
3.3 Hardware accelerators 21
3.3.1 Unified architecture accelerators 21

3.3.2 GNN acceleration using Tiled architecture . . . 24
3.3.3 Hybrid architectures for GNN acceleration . . . 25
3.3.4 Challenges . . . 28
3.4 High-Level Synthesis based accelerators . . . 29
3.5 Software-Hardware co-design accelerators . . . 32
3.6 Graph processing acceleration . . . 36
3.7 Matrix multiplication optimization . . . 37
3.7.1 Matrix multiplication optimization in MLIR . . . 37
3.8 Conclusion . . . 38

4 Problem Formulation **41**
4.1 Graph Neural Network acceleration . . . 41
4.2 Motivation and objective . . . 42

5 Design of the Toolchain **45**
5.1 PyTorch . . . 46
5.1.1 Toolchain inputs . . . 47
5.2 Torch-MLIR . . . 48
5.2.1 From PyTorch to TorchScript . . . 49
5.2.2 Torch-MLIR Compilation . . . 50
5.3 Synthesizer . . . 52
5.3.1 SODA-OPT . . . 52
5.3.2 PandA-Bambu . . . 56
5.4 Matrix multiplication algorithm . . . 57
5.4.1 Emulating global memory coalescing of CUDA . . . 58
5.5 Conclusion . . . 60

6 Experimental Procedures and Results **61**
6.1 Model analysis and profiling . . . 61
6.2 Matrix multiplication acceleration . . . 62
6.2.1 PyTorch matrix multiplication benchmark . . . 62
6.2.2 FPGA-accelerated matrix multiplication . . . 64
6.3 GCN accelerator evaluation . . . 68
6.3.1 Model accuracy . . . 70
6.4 State of the art comparison . . . 71

7 Conclusion and Future Developments **73**
7.1 Future developments . . . 74

Bibliography **77**

List of Figures **83**

List of Tables **85**

List of Algorithms **87**

List of Listings **89**

List of Symbols **91**

Acknowledgements **93**

1 Introduction

Over the past few years, deep learning has significantly revolutionized various machine learning tasks, spanning from image classification and video processing to speech recognition and natural language understanding. Traditionally, these tasks have predominantly operated within the Euclidean space, where data is typically represented. For instance, in image analysis applications, images can be considered as functions defined on the Euclidean space (plane) and sampled on a grid. Nevertheless, a growing number of applications now generate data from non-Euclidean domains [7], presenting it in the form of complex graphs with intricate relationships and interdependencies among objects. The inherent complexity of graph data has posed considerable challenges for existing machine learning algorithms. Consequently, there has been a surge of studies focusing on extending deep learning techniques to accommodate and leverage graph data.

Graph neural networks (GNNs) have been introduced in response to the growing demand for learning tasks involving graph data, which encompasses extensive relational information among its elements. These models effectively capture the interdependence among graph nodes by employing message passing mechanisms.

Optimizing and accelerating the capabilities of Graph Neural Networks is necessary due to their increasingly popularity, particularly in domains characterized by vast amounts of data, such as social networks and chemistry. In particular, inference in GNNs refers to the time the model takes to make predictions after training. The duration of the inference process determines the speed at which queries are answered, and researchers strive to minimize this time span.

In applications of deep learning that prioritize low latency, Field-programmable Gate Arrays (FPGAs) can outperform other computing devices, such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs). FPGAs offer the advantage of being fine-tuned to the application to strike the optimal balance between power efficiency and meeting performance requirements.

Due to these reasons, researchers have been actively pursuing the development of new FPGA accelerators for Graph Neural Networks in recent times.

A common approach to hardware design involves a combination of manual coding and automated processing. In particular, first the functional units are implemented in a programming language such as C/C++, then they are transformed into a Hardware Description Language (HDL) using commercial High-Level Synthesis (HLS) tools. Following functional verification, the HDL kernels are forwarded to downstream logic synthesis and physical design tools, and finally integrated into a system. However, this method demands significant effort and relies heavily on the expertise of the designers, leading to varying quality of results.

To address the challenge of accelerating GNNs on FPGAs without having extensive knowledge in hardware design, the objective of this thesis is to develop a comprehensive toolchain that, starting from PyTorch [31], a cutting-edge high-level programming framework for creating neural network algorithms based on the Python programming language, enables the automatic generation of a Graph Neural Networks FPGA accelerator with minimal effort required.

The suggested toolchain represents an enhancement of the SODA toolchain [2]. It operates by transforming the PyTorch model, provided as input, into a multi-level intermediate representation (MLIR) [26] utilizing Torch-MLIR [38], an MLIR based compiler toolkit for PyTorch programs. This MLIR representation is then passed to the SODA framework to conduct hardware/software partitioning of the algorithm specifications and architecture-independent optimizations. Following this, the framework generates a low-level IR (LLVM IR) specifically tailored for the hardware generation engine, PandA-Bambu [13].

As GNNs often deal with massive graph sizes, the computation time and memory requirements are substantial. Consequently, a significant portion of the research focuses on analysis of computational bottlenecks and optimizing the computation phase of Graph Neural Networks using custom optimizations.

The toolchain's effectiveness is validated through experimental results, demonstrating significant enhancements in performance. The generated accelerators improved inference speeds while preserving the accuracy of GNN models.

1.1 Contributions

In this thesis, a novel toolchain is introduced to accelerate GNN inference, alongside customized optimizations designed to finely enhance model performance.

The initial phase in designing the toolchain involves identifying appropriate technologies. Given its extensive usage and popularity in neural network implementations, PyTorch

serves as the basis of this design process. Following that, the selection of a technology to establish a connection between the high-level framework and the synthesizer was achieved through Torch-MLIR. This facilitated the acquisition of an MLIR representation for subsequent synthesis.

To the best of the author's knowledge, this thesis stands as one of the first documented attempts in combining Torch-MLIR with Graph Neural Networks developed in PyTorch and PyTorch Geometric.

The compatibility between PyTorch and Torch-MLIR is enabled through the TorchScript intermediate representation. This study offers general adjustments for various GNN models to make them compatible with this representation, thus enabling the conversion of the model to MLIR.

Subsequently, the following phase involved identifying the bottlenecks within the models for optimization during the synthesis stage. A substantial portion of this research is dedicated to evaluating and identifying the most effective optimizations offered by SODA-OPT and PandA-Bambu for GNNs. Both of these technologies provide various optimizations applicable during the synthesis phase, continuously integrating new features. This study includes theoretical and practical evaluations concerning the combination of their optimizations.

To enhance the inference of the GNN model employed in this thesis, an initial step involved constructing a matrix multiplication accelerator. Subsequently, the insights gained were implemented in the GNN model, generating promising outcomes. The final contribution of this thesis is an optimized GCN accelerator, demonstrating enhanced inference speeds across diverse graph structures and sizes.

1.2 Thesis structure

Chapter 1 introduced the context of the thesis, its objective, and its goals. Chapter 2 presents background about Graph Neural Networks, how they work, an explanation of the GNN types used in the thesis, and the type of tasks that they can perform, including some of their applications. Additionally, it presents the SODA framework, the starting point for this thesis's proposed toolchain. Chapter 3 contains an overview of related work; other Graph Neural Network acceleration frameworks are analyzed, underlying their differences compared to the proposed approach and their limitations. Chapter 4 formulates the problem statement, summarizes the open issues of the research objective, and explains the expected impact. Chapter 5 explains how the problem has been faced

and what technologies have been used. It contains a detailed description of the proposed toolchain and its working method. Chapter 6 lists all the performed experiments, gives the necessary information to reproduce them and contains their outcomes and the issues and limitations encountered. Finally, Chapter 7 presents overall considerations of the study, both with the main achievements obtained and the most notable obstacles faced. Along with this, potential improvements for future studies are considered.

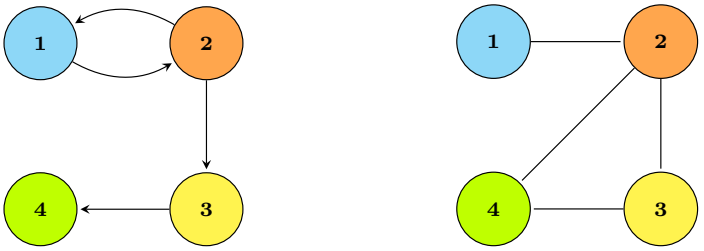
2 Background

This chapter provides essential background to understand of the thesis content and objectives. It begins by introducing the graph data structure, which is crucial for comprehending Graph Neural Networks. Additionally, the chapter provides an introduction to Graph Neural Networks, outlining their capabilities and exploring various applications. Furthermore, it introduces two essential tools, SODA-OPT and PandA-Bambu, which are integral parts of the SODA Toolchain that served as the foundation for this research.

2.1 Graphs

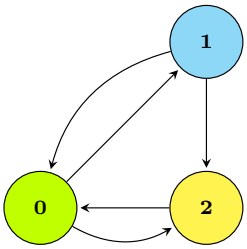
Graphs are data structures representing a collection of objects, known as vertices or nodes, and a set of edges connecting them [49]. In a graph, the edges can be either directed or undirected, as shown in Figure 2.1, and they typically connect two vertices, which may or may not be distinct. The vertices represent entities or elements, and the edges represent their relationships or connections.

Graphs serve as a versatile tool for describing diverse forms of data. For example, molecules, the fundamental units of matter, are composed of atoms and electrons arranged in three-dimensional space. In this intricate structure, all particles interact with each other. However, when a pair of atoms are stably positioned at a specific distance, their connection is referred to as a covalent bond. These bonds with distinct atomic distances can vary in nature, such as single or double bonds. Representing this complex



(a) Directed Graph (b) Undirected Graph

Figure 2.1: Example of directed and undirected graphs



(a) Graph

	0	1	2
0	0	1	1
1	1	0	1
2	1	0	0

(b) Adjacency matrix

Figure 2.2: Example of a graph and its adjacency matrix

three-dimensional object as a graph offers a practical and widely adopted abstraction, where atoms are nodes and covalent bonds act as edges [12].

Social networks provide another domain where graphs are used: in fact, they serve as valuable tools for examining patterns within the collective behavior of people, institutions, and organizations. By representing individuals as nodes and their relationships as edges, we can construct a graph that effectively captures groups of people and their interconnectedness.

2.1.1 Graph Representation

Graphs are easy to visualize, but a more formal representation is needed when implementing graph algorithms.

Adjacency matrix

The adjacency matrix of a graph is a fundamental representation that provides information about the relationships between nodes in the graph. It provides a compact and easily interpretable representation of the graph's edges and connections, which can be easily implemented in almost all programming languages using two-dimensional arrays.

The adjacency matrix of a graph is a matrix of dimensions $N \times N$ where N is the number of nodes in the graph. Each matrix cell is set to 1 if the two nodes are connected, i.e. if there is an edge starting from the node of the corresponding row to the one of the corresponding column, and zero otherwise. If the graph is undirected, each edge is bidirectional and so the matrix is symmetric. If in the graph there are no self-loops, then the main diagonal of the matrix will be with all zeros. Figure 2.2 shows a directed graph and its adjacency matrix.

The adjacency matrix consists only of ones and zeros. In real-world graph-related problems, the number of edges is usually much lower than the number of nodes, leading to an

(a) COO format (b) CSR format

Figure 2.3: COO and CSR format of Adjacency matrix in Figure 2.2

adjacency matrix with many zero elements. Such matrices with mostly zero elements are called sparse matrices. Their sparsity enables more efficient storage and manipulation, avoiding both the storage of zeros and the operations including zero elements, reducing the computational effort.

There are two common representations for sparse matrices: the Coordinate List (COO) format and the Compressed Sparse Row (CSR) format, which are described below.

Coordinate format

In the COO format, a sparse matrix is represented as a list of [row, column, data] tuples, where each tuple corresponds to a non-zero element in the matrix. The [row, column] coordinates represent the position of the non-zero element, and the value is the actual numerical value of that element. Usually, it is preferred to store the entries first by row index and then by column index, to improve random access times. In Figure 2.3 the COO format of the adjacency matrix represented in Figure 2.2 is reported. For example, by considering the first element of the three arrays, it is possible to understand that the data one is placed in position [0, 1] of the adjacency matrix.

The COO format is helpful for sparse matrices because it does not require any assumptions about the sparsity pattern and allows for efficient deletion and insertion of elements. However, it may not be the most efficient format for large and highly sparse matrices, as it may require more memory and may not support efficient row-wise or column-wise operations. In these cases, other formats, like the CSR one, are often preferred.

Compressed Sparse Row format

In the CSR format, a sparse matrix is represented using three arrays: the values array, the row pointers array (indices), and the column indices array (index pointers). The data array contains the non-zero elements of the sparse matrix stored in row-major order, the array indices contains the column position of each data, while the index pointers contains an increasing number of how many non-zero elements there are in the matrix row by row.

Given a matrix of size $m \times n$, with NNZ being the number of non-zero elements, the arrays data and indices are of length NNZ, while the array index pointers is of length $m + 1$ due to the fact that its first element is always zero. Figure 2.3b shows the CSR format of the adjacency matrix represented in Figure 2.2.

Feature matrix
Suppose to have a graph of a social network, where each node corresponds to a person and each edge to a friendship on the social media between the two nodes. If the aim is to predict the possible future friendship that could be established, maybe putting those people in the suggested friend list, having more information (features) about each node, such as the age or the gender, is helpful.

A feature vector represents the features or attributes associated with a single entity. The feature matrix of a graph contains multiple feature vectors; it represents the features or attributes associated with each node. It is commonly denoted as X and each row corresponds to a node in the graph, and each column corresponds to a specific feature or attribute of that node.

2.2 Graph Neural Networks

Graph neural networks are deep learning techniques that operate on graph-structured data. Thanks to their impressive performance, GNNs have recently gained significant popularity as a widely adopted method for graph analysis [22]. Figure 2.4 illustrates the steady growth in the number of publications related to Graph Neural Networks on Google Scholar from 2015 to 2022. The increasing trend reflects the rising interest and research activity in the field of GNNs over the years.

Graph Neural Networks are a group of neural networks which are designed to solve different tasks. Prediction tasks on graphs can generally be classified into three categories: graph-level, node-level, and edge-level predictions [32].

In a graph-level task, the objective is to predict the property or characteristic of an entire graph. For instance, when considering a molecule represented as a graph, the goal may be to predict attributes such as its probability of binding to a receptor associated with a specific disease. This assignment is comparable to image classification tasks, where the objective is to assign a label to an entire image. Similarly, in text analysis, sentiment analysis serves as a similar problem where the goal is to determine a complete sentence's overall mood or emotion in one go.

Node-level tasks involve predicting the identity or function of individual nodes within a

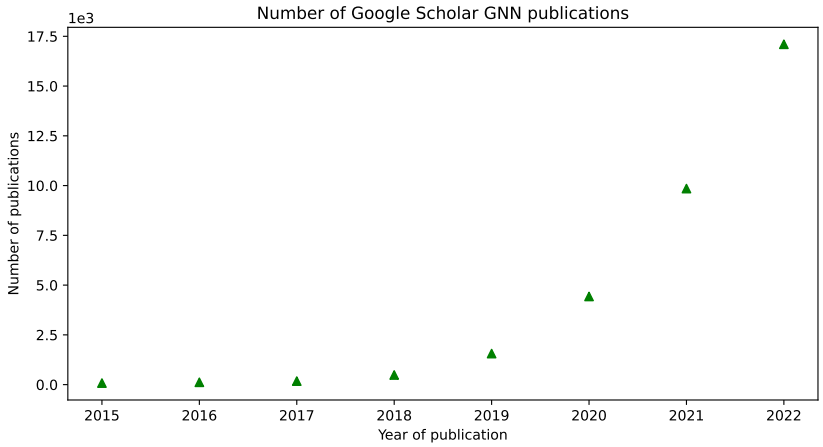


Figure 2.4: Number of GNN publications on Google Scholar per year. The data were collected by querying papers containing the specific words "Graph Neural Network" in their whole content and aggregating them on a yearly basis.

graph. One example of a node-level task is node classification in a social network. Given a social network graph where nodes represent individuals and edges represent relationships between them, the task is to predict the demographic attributes or characteristics (e.g., age, gender, occupation) of each node based on their connection patterns and features. Drawing an analogy to image processing, node-level prediction problems can be compared to image segmentation tasks, where the objective is to assign labels to each pixel in an image based on its role. Similarly, in text analysis, a comparable task would involve predicting the parts of speech for each word in a sentence, such as identifying whether a word is a noun, verb, adverb, and so on.

The remaining prediction task in graphs pertains to edge prediction. One example of an edge-level task is link prediction in a social network. Given a graph representing a social network where, as before, in node-level tasks, nodes correspond to individuals and edges represent relationships between them, the edge-level task aims to predict missing or potential connections between nodes. This can involve predicting the likelihood of a future friendship or the probability of a collaboration between individuals based on their shared characteristics or mutual connections in the network.

Graph Neural Networks are designed to process graph data and consist of multiple interconnected layers. At its core, a GNN is an algorithm that exploits the connectivity within a graph to understand and represent the relationships between nodes. By relying on the graph's structure, the GNN iteratively processes input edge, vertex, and graph feature vectors, which encode known attributes and transforms them into output feature vectors

that capture the desired predictions. Each Graph Neural Network typically encompasses three main stages: pre-processing, iterative updates and decoding or readout [1].

1. **Pre-processing**: this initial step, while optional, involves transforming the input feature vectors and graph structure representation through a pre-processing procedure.

2. **Iterative updates**: following pre-processing, the feature vectors of each edge and vertex undergo iterative updates using aggregate-combine functions. For edge updates, attributes from the edge itself, connected vertices, and the graph are aggregated and combined to generate a new edge feature vector. Similarly, vertex updates involve aggregating feature vectors from neighboring vertices $\mathcal{N}(v)$ and combining them to obtain a new feature vector. This iterative process gradually incorporates relationships between increasingly distant nodes and edges, allowing for multi-hop updates. Furthermore, the graph may coarsen through pooling [45] (i.e. selective reduction or adjustment of either the graph structure or the neighborhood set of each node) in each subsequent layer, or the neighborhood set may change via layer sampling [17] (i.e. coarsening the graph from one layer to the next, leading to a reduction in the number of nodes that need to be processed during aggregation and combination steps).

3. **Decoding or readout**: once the graph possesses a global feature vector, it is updated once upon completion of edge and node updates. The final output can be an edge/node embedding, representing specific information about each edge or node in a low-dimensional feature vector format, or a graph embedding that summarizes the entire output graph.

Performing these stages on large and sparse graphs can introduce dynamic computational data flow and numerous irregular memory access patterns.

GNNs, as previously said, are structured into layers, each representing an iteration in the update process described earlier. This layering allows information to propagate across nodes, enabling the influence of distant nodes. Consequently, the appropriate number of layers in a GNN will vary depending on the significance of relationships among distant nodes in a specific application. The commonly adopted range for the number of GNN layers is 1 to 5, as an excessive number of layers can introduce undesired problems such as feature over-smoothing, vanishing gradients, or over-fitting [27].

Different popular Graph Neural Network architectures have been proposed recently, some of which are more suitable for some tasks than others. A summary of two types of GNNs

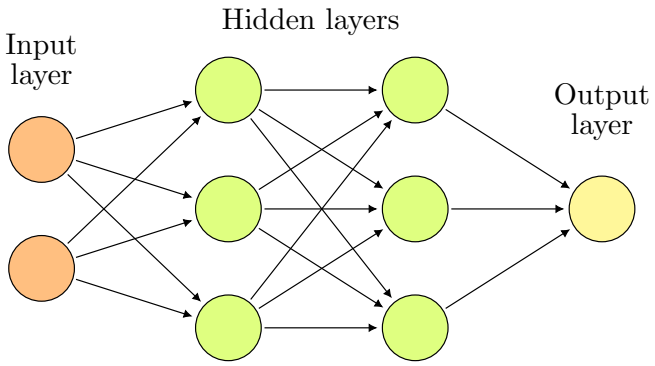


Figure 2.5: Example of a Multi-Layer Perceptron

is provided in the following subsections, preceded by an important definition.

2.2.1 Multi-Layer Perceptron

The central component of an Artificial Neural Network is referred to as the Multi-Layer Perceptron (MLP). As depicted in Figure 2.5, it comprises multiple interconnected layers, each containing processing elements (neurons) that perform weighted sums of inputs and biases.

Input layers receives the input data, where each neuron represents a feature or attribute. Hidden layers are responsible for learning and extracting complex features from the input data. Output layers generate the final output of the network. Processing elements in each layer receive inputs from all nodes in the previous layer and send their outputs to all neurons in the subsequent layer.

2.2.2 Graph Convolutional Network

A Graph Convolutional Network (GCN) [11, 25] is a type of neural network architecture explicitly designed to operate on graph-structured data. GCNs aim to learn node representations by aggregating and combining information from neighboring nodes in the graph. The core idea behind GCNs is to perform convolution-like operations on the graph, where the convolutional filters are defined based on the graph's adjacency matrix or other graph-specific structures. This enables GCNs to capture and leverage the structural information encoded in the graph to make predictions or perform downstream tasks. GCNs have demonstrated effectiveness in various applications, including node classification, link prediction, and graph classification.

Given an undirected graph $\mathcal{G} = (V, E)$, where V represents the set of nodes (vertices),

and E represents the set of edges, with an adjacency matrix $\tilde{A} = A + I_N$, where I_N is the identity matrix, the layer-wise propagation rule in a GCN can be expressed as:

$$H^{(l+1)} = f\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{2.1}$$

Where $H^{(l)} \in \mathbb{R}^{N\times D}$ is the input node features matrix, $W^{(l)}$ is a layer-specific learnable weight matrix, $\tilde{D}$ is the degree matrix defined as $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, and $f(\cdot)$ represents a non-linear activation function applied element-wise, such as $ReLU(\cdot) = max(0, \cdot)$. The equation above demonstrates the propagation of node features through graph convolution, where the adjacency matrix $\tilde{A}$ captures the connectivity information of the graph, $\tilde{D}^{-\frac{1}{2}}$ normalizes the adjacency matrix, and $H^{(l)}W^{(l)}$ performs a linear transformation of node features. The resulting $H^{(l+1)}$ represents the updated node representations after the graph convolution operation. In practice, multiple graph convolutional layers can be stacked to capture increasingly complex relationships and further refine the node representations.

2.2.3 Graph Isomorphism Network

A Graph Isomorphism Network (GIN) [11, 43] is a type of neural network architecture designed to operate on graph-structured data by capturing graph isomorphism, which is the property of two graphs having the same structure, inspired by the Weisfeiler-Lehman (WL) graph isomorphism test [40]. The WL test involves an iterative process where it (1) aggregates the labels of nodes and their neighborhoods, and (2) hashes the aggregated labels into unique new labels. The algorithm determines that two graphs are non-isomorphic if, during any iteration, the labels of the nodes between the two graphs differ. GINs aim to learn node representations that are invariant under graph isomorphism, enabling them to generalize across different graphs with similar structures.

The learned vertex features from GIN-Conv can be directly utilized for tasks such as node classification and link prediction. The model is based on the following rule:

$$h_v^{(k+1)} = MLP^{(k)}\left(\left(1+\epsilon^{(k)}\right)\cdot h_v^{(k)} + \sum_{u\in\mathcal{N}(v)} h_u^{(k)}\right) \tag{2.2}$$

Where $h_v^{(k)}$ represents the initial node representation of node v, $\mathcal{N}(v)$ represents the neighborhood of node v, ϵ is a learnable parameter or a fixed scalar, $MLP(\cdot)$ represents a Multi-Layer Perceptron, defined in Subsection 2.2.1, and $h_v^{(k+1)}$ represents the updated node representations.

In the neighborhood aggregation process of GINs, each node's representation is updated by considering its own representation and its neighbors' representations. The neighborhood aggregation is performed through the MLP operation, followed by non-linear activation.

GINs are trained using graph-level objectives, such as graph classification or property prediction, and aim to learn invariant representations under graph isomorphism, allowing them to generalize well to unseen graphs with similar structures. However, even if the node embeddings acquired through GIN can be directly applied to tasks such as node classification and link prediction, in the case of graph classification tasks, it is necessary to use a Readout function that takes individual node embeddings as input and produces the embedding representation for the entire graph.

The Readout function is then utilized to generate the overall representation of the graph, leveraging the individual vertex representations. By concatenating the results from all iterations of GINConv, the final graph representation is obtained as:

$$h_G = CONCAT\left(READOUT\left(\left\{h_v^{(k)} | v \in G\right\}\right) | k = 0, 1, ..., K\right) \tag{2.3}$$

Where *READOUT* in 2.2 can be replaced with a sum operator in order to generalize the WL test.

2.3 MLIR

MLIR [26] represents a novel approach for constructing reusable and extensible compiler infrastructure. Addressing software fragmentation and enabling compilation for heterogeneous hardware reduces the effort to build domain-specific compilers and seamlessly connect existing compilers. MLIR empowers the design and implementation of code generators, translators, and optimizers across various levels of abstraction, application domains, hardware targets, and execution environments.

The MLIR project aims to address programming language design and implementation challenges. It achieves this by introducing new abstraction levels and built-in infrastructure to solve common compiler engineering issues. MLIR ensures extensibility through Dialects, which logically group operations, attributes, and types under a unique namespace. Dialects do not introduce new semantics but act as a mechanism for group-related functionality, such as constant folding behavior, for all operations in the dialect. They organize the ecosystem of language- and domain-specific semantics while adhering to the principle of simplicity. Unknown operations are treated with a conservative approach

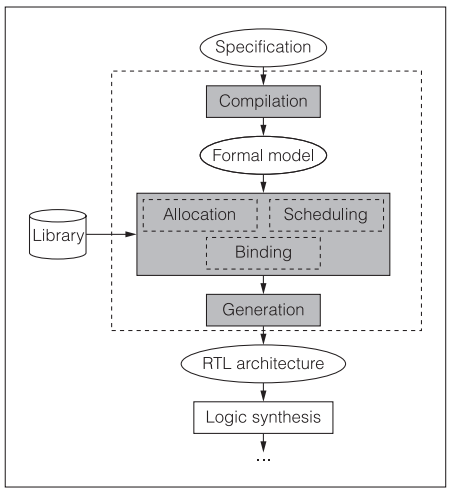


Figure 2.6: High-level Synthesis design steps [10]

by compiler passes, and MLIR offers extensive support for describing the semantics of operations to passes through traits and interfaces.

2.4 High-Level Synthesis

This thesis presents a toolchain leveraging High-Level Synthesis to automatically translate algorithmic descriptions of applications into low-level RTL (register-transfer level) descriptions or digital circuits. HLS tools simplify hardware development's most complex and time-consuming aspect by allowing users to provide programs written in common programming languages like C/C++, eliminating the need for manual VHDL/Verilog coding. HLS significantly reduces design time and effort, making the final implementation's functionality independent of hardware design knowledge.

The High-Level Synthesis process [10], depicted in Figure 2.6, begins with compiling the functional specification, transforming it into a formal representation, and applying various code optimizations. The allocation phase determines the type and quantity of hardware resources required to meet design constraints. In the scheduling phase, all operations within the specification model are scheduled into cycles. The operation's scheduling can vary based on the functional component to which it is mapped, resulting in either single-clock-cycle scheduling or multi-cycle scheduling. The binding algorithm optimizes the selection of storage units for variables carrying values across cycles and the assignment of functional units for each operation in the specification model. The next step is to generate an RTL model of the synthesized design, incorporating all the design decisions made.

2.5 LLVM

LLVM is a compiler framework that enables ongoing analysis and transformation of programs through a code representation independent of any specific language.

Many HLS frameworks rely on it, thanks to its intermediate level of abstraction. It is also the case of the HLS tool used for this thesis, which can take C/C++ as well as an LLVM representation as input.

The LLVM Intermediate Representation (IR), which resembles assembly code, is a low-level code representation. LLVM uses a language-independent type system and utilizes an infinite set of virtual registers. All instructions follow the Static Single Assignment (SSA) form, meaning each virtual register appears in only one instruction, and each instruction takes one or two input operands while producing a single result.

2.6 SODA Toolchain

SODA [2] is a software-defined accelerator synthesizer. It enables the creation of highly specialized accelerators from algorithms designed in high-level programming frameworks. The synthesizer comprises a compiler-based frontend that interfaces with high-level programming frameworks, applying advanced optimizations. It also includes a compiler-based HLS backend responsible for generating Verilog code and interfacing with external tools to compile the final design, which can be applied to application-specific integrated circuits (ASICs) or Field-Programmable Gate Arrays.

SODA's exceptional power lies in its ability to offer a fully automated end-to-end hardware compiler, eliminating the need for human intervention and any modifications to the input code. This framework seamlessly integrates with high-level Python frameworks by accepting their input descriptions, which are then translated by the frontend into a high-level intermediate representation. Leveraging MLIR, the frontend facilitates hardware/-software partitioning of algorithm specifications and performs architecture-independent optimizations. Following this, it generates a low-level IR that is utilized by the hardware generation engine, PandA-Bambu [13]. Throughout the entire SODA toolchain, compiler passes are employed to implement optimizations at all levels, greatly influencing the generated hardware designs' performance, area, and power characteristics.

2.6.1 SODA-OPT Frontend

SODA-OPT, the high-level compiler frontend of the SODA synthesizer, performs search, outlining, optimization, dispatching, and acceleration passes on the input program. Its primary objective is to prepare the program for hardware synthesis, targeting either FPGAs or ASICs. To accomplish these tasks, SODA-OPT relies on and extends the MLIR framework.

Code regions selected for hardware acceleration undergo an optimization pipeline that progressively lowers them through various MLIR dialects until they are ultimately translated into an LLVM IR format tailored explicitly for hardware synthesis. On the other hand, the host module is lowered into an LLVM IR file containing runtime calls to control the generated custom accelerators.

2.6.2 SODA Synthesizer Backend

PandA-Bambu, the SODA synthesizer backend, harnesses cutting-edge HLS techniques to produce accelerator designs using the low-level LLVM IR generated by the SODA frontend. PandA-Bambu supports multiple frontends based on standard compilers such as GCC or CLANG. It constructs an internal IR to execute HLS steps and generates designs in HDL formats, such as Verilog or VHDL. In addition to synthesizable HDL, PandA-Bambu can automatically generate testbenches for verification purposes. Using PandA-Bambu, the SODA synthesizer can target both FPGAs and ASICs.

PandA-Bambu is optimized to handle a broad range of C and C++ constructs while also being able to process LLVM IR through its internal Clang frontend. Through SODA-OPT, PandA-Bambu can be connected with MLIR code. The LLVM IR generated after SODA-OPT's high-level optimizations undergoes explicit restructuring for HLS, resulting in more efficient accelerators than direct translation from MLIR to LLVM IR.

2.7 Conclusion

This chapter has presented the foundational concepts necessary for understanding the subsequent contents of this thesis. It provided a concise overview of the broad domain of Graphs and Graph Neural Networks, explicitly focusing on the architectures of Graph Convolutional Networks and Graph Isomorphism Networks. Additionally, the chapter introduced SODA and PandA-Bambu, which will be further investigated within the context of the proposed design flow for the creation of GNNs FPGA-based accelerators.

The following chapter is dedicated to an analysis of scientific literature on hardware acceleration for Graph Neural Networks. This analysis primarily focuses on publications concerning FPGA-based implementations and design flows that leverage High-Level Synthesis techniques.

3 Related Work

Accelerating Graph Neural Networks has become a subject of interest within the research community, exploring especially FPGA accelerators. In this chapter, a comprehensive examination is conducted on cutting-edge GNN FPGA accelerators and design flows based on High-Level Synthesis to implement them. This chapter also delves into the relevant literature concerning various approaches to optimization of matrix multiplication operations, since it represents the majority of operations in a GNN.

3.1 Chapter structure

This chapter first presents the software frameworks utilized to design and train Graph Neural Networks, then provides an overview of state-of-the-art hardware accelerators, categorized based on their architecture types [1].

Particular attention is given to FlowGNN [33], an accelerator implemented using High-Level Synthesis, the same technique used by the design flow proposed in this thesis. As mentioned earlier, optimizing the matrix-matrix multiplication operation was a significant aspect of this research. Thus, a dedicated section focuses on state-of-the-art optimizations for matrix-matrix multiplication.

Finally, the chapter concludes with a summary and comparison among the accelerators presented.

3.2 Software frameworks

The challenges posed by GNN processing have led to inefficiencies in traditional deep neural network (DNN) libraries and graph processing frameworks. This is primarily due to the alternating computational phases characteristic of GNNs. While DNN libraries excel in accelerating combination operations within vertices and edges, they need help with aggregation tasks. On the other hand, graph processing libraries effectively handle irregular memory accesses during graph traversal but assume simplistic operations at the vertices, which is not the case in GNNs. Recent studies tried to bridge the gap by adapting

the DNN libraries to Graph Neural Network challenges.

The two main software frameworks trying to design and improve GNN computation are PyTorch Geometric [14] and Deep Graph Library (DGL) [39]. They both provide many examples and code for multiple GNN architectures providing optimizations for the performance improvement of both training and inference.

PyTorch Geometric is a PyTorch-based library specifically designed for deep learning on input data with irregular structures, including graphs, point clouds, and manifolds. In addition to offering comprehensive graph data structures and processing techniques, it incorporates many state-of-the-art methods from the relational learning and 3D data processing domains. PyTorch Geometric achieves impressive data throughput by utilizing sparse GPU acceleration, offering specialized CUDA kernels, and implementing efficient mini-batch handling for input examples of varying sizes. A key aspect of PyTorch Geometric involves defining a message-passing interface encompassing message and update functions for neighborhood aggregation and combination and multiple pooling operations.

DGL is a recently developed library that seamlessly integrates with TensorFlow, PyTorch, or MXNet. It introduces three essential functions: *message* for aggregating edges, *update* and *reduce* for aggregating and combining at the nodes. DGL adopts a matrix multiplication approach to enhance performance and harnesses specialized kernels designed for GPUs or TPUs. DGL intelligently selects the optimal parallelization scheme using heuristics, considering various factors, including the input graph characteristics. It distills the computational patterns of GNNs into a set of generalized sparse tensor operations, which facilitate extensive parallelization. By prioritizing the graph as the central programming abstraction, DGL enables transparent optimizations. Furthermore, through a framework-neutral design philosophy, DGL allows users to effortlessly port and leverage existing components across multiple deep learning frameworks.

The approach used by DGL outperformed PyTorch Geometric in training Graph Neural Networks, as stated in their paper [39]. However, both libraries target CPU and GPU architectures. Knowing the potentially higher computational power of FPGA, the field of hardware accelerators started gaining more and more interest, with the expectation of having GNN hardware accelerators capable of outperforming the performance of libraries targeting CPU/GPU.

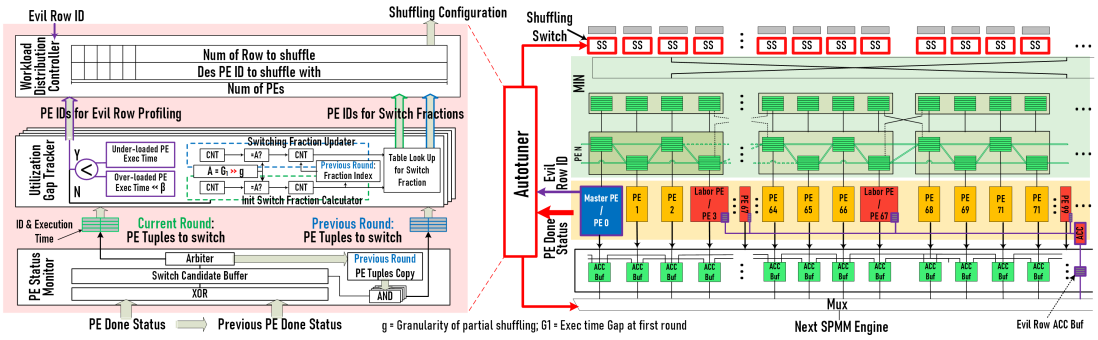


Figure 3.1: Overall architecture of SpMM engine in AWB-GCN with three rebalancing techniques: distribution smoothing, remote switching (red bordered) and evil row remapping (purple bordered) [15]

3.3 Hardware accelerators

As discussed in Section 3.2, software frameworks optimize the execution of GNNs in CPU-GPU platforms, commonly found in various computing systems, leading to substantial speed improvements in inference and training processes.

However, custom hardware accelerators could overcome the challenges of GNN computing and achieving order-of-magnitude enhancements. Consequently, numerous hardware accelerators with different architecture types have emerged, aiming to address the intensive computational demands and alternating patterns required by GNNs.

3.3.1 Unified architecture accelerators

A unified architecture refers to a design approach where the FPGA, instead of having specialized hardware modules for specific functions, has more generic PEs in which different operations are scheduled.

Autotuning-Workload-Balancing GCN (AWB-GCN), presented in [15], aims to accelerate Graph Convolutional Network inference. This accelerator actively adapts to the structural sparsity of GNNs. The authors support their design by analyzing the power-law distribution found in most graphs, positing that certain parts of the computation will exhibit density. In contrast, others will be extraordinarily sparse, leading to imbalances.

In order to tackle this problem, the architecture, shown in Figure 3.1, devises a custom matrix multiplication engine that efficiently supports skipping zeros. In particular, three hardware-based autotuning techniques to address the imbalance have been suggested: dynamic distribution smoothing, remote switching, and row remapping.

Specifically, AWB-GCN continuously monitors the sparse graph pattern, dynamically adjusts the workload distribution among many processing elements, and reuses the optimal configuration upon convergence. Data from memory is directed through a task distributor and queue (TDQ) to a collection of processing elements (PEs) and accumulators. The TDQ has two designs tailored for scenarios with moderate or high sparsity. Given AWB-GCN's emphasis on GCNs featuring linear aggregation functions, the authors suggest prioritizing combination processing, as this typically reduces the number of features and subsequently minimizes the operations performed during aggregation. Additionally, AWB-GCN incorporates a fine-grained pipelining mechanism to effectively overlap the execution of combination and aggregation, even within the same layer.

However, at the heart of the AWB-GCN architecture lies the management of load balancing at three levels of granularity: distribution smoothing to handle local utilization fluctuations among PEs, remote switching for minor crests, and row remapping for prominent crests. At the beginning of the processing, rows are evenly distributed among processing elements. Throughout each round of calculation, distribution smoothing equalizes the workloads among neighboring PEs. The architecture of AWB-GCN effectively monitors the runtime PE utilization by tracking the number of pending tasks in task queues. It continually offloads the work from more burdened PEs to their less occupied neighbors, up to 3-hop neighbors.

Remote switching is implemented to tackle regional clustering, wherein the process facilitates partial or complete workload exchanges between underutilized and overloaded PEs. An auto-tuner dynamically determines the switch fraction at runtime, relying on the PE utilization observed in each round. The accelerator retains the switch strategies employed in the current round and iteratively optimizes them based on utilization information gathered in the subsequent round. As a result, after several rounds of auto-tuning, the switch strategy that best aligns with the sparse matrix structure is attained and is then utilized for the remaining rounds, leading to nearly perfect PE utilization.

Lastly, the evil-row remapping technique redistributes the evil row, i.e. a row of data that cannot be effectively smoothed or balanced through remote switching, to the most under-loaded PEs in troughs, allowing the neighboring PEs to assist. Row remapping is initiated based on demand after each round. The auto-tuner assesses the utilization gaps between the most overloaded and under-loaded PEs and decides if their gaps exceed remote switching capability. If so, row remapping is executed as a solution.

AWB-GCN proves to be a fascinating accelerator, though its generalizability beyond Graph Convolutional Network remains uncertain. On the other hand, EnGN [18] repre-

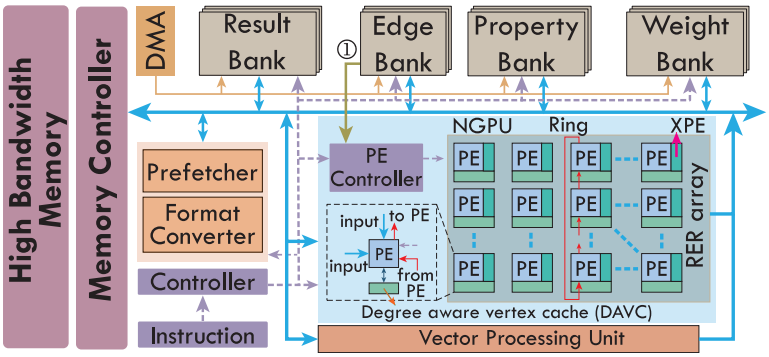


Figure 3.2: EnGN hardware architecture [18]

sents another accelerator featuring a unified architecture, with the primary goal of being adaptable for various Graph Neural Network models.

EnGN is a specialized accelerator architecture that prioritizes high-throughput and energy-efficient processing of large-scale GNNs in which the Graph Neural Network is treated as a concatenated matrix multiplication of feature vectors, adjacency matrices, and weights, all efficiently scheduled in a single data flow. An array of clustered Processing Elements (PEs) is supplied with independent banks for features, edges, and weights, enabling computation of the combination function. The hardware architecture of EnGN is shown in Figure 3.2.

EnGN accelerates the three fundamental stages of GNN propagation to handle sparsity efficiently, i.e., feature extraction, aggregate, and update, which encapsulates common computing patterns shared by typical GNNs. The authors introduce the ring-edge-reduce (RER) dataflow for the aggregation, in which each column of PEs is interconnected through a ring, and results are passed along and added based on the adjacency matrix. This process effectively addresses the poor locality of sparsely and randomly connected vertices and efficiently supports critical stages. EnGN dynamically reorders edges in each RER step to reduce redundant computations in sparsely connected nodes.

Since well-connected vertices frequently appear during computation, PE clusters have a degree-aware vertex cache that stores data for high-degree vertices. Other optimized design decisions in EnGN involve the order of matrix multiplications when the aggregation function is a sum, impacting the total number of operations.

Moreover, EnGN employs a graph tiling strategy to accommodate large graphs, optimizing the utilization of hierarchical on-chip buffers through adaptive computation reordering and tile scheduling. These optimizations collectively enhance the overall performance of EnGN for large-scale GNN processing tasks.

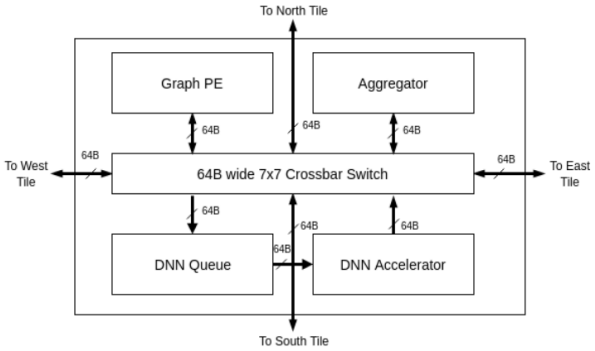


Figure 3.3: Block diagram of a tile in the GNN accelerator proposed by Auten *et al.* [3]

3.3.2 GNN acceleration using Tiled architecture

A tiled architecture refers to a design approach where the FPGA fabric is organized into a regular grid-like pattern of configurable tiles. Each tile typically consists of a set of logic cells, interconnect resources, and other functional units, and these tiles are repeated across the entire FPGA.

In contrast to most other accelerators, this work [3] presents a modular architecture for convolutional GNNs incorporating dedicated hardware units to efficiently handle the irregular data movement essential for graph computation in GNNs, while simultaneously delivering the high compute throughput required by GNN models. The fundamental building block, as shown in the block diagram of Figure 3.3 of the accelerator is a tile consisting of an aggregator module (AGG), a DNN accelerator module (DNA), a DNN queue (DNQ), and a graph PE (GPE), all interconnected via an on-chip router.

The Graph Processing Element (GPE) handles graph traversal and sequencing computation steps dependent on the underlying graph structure. The DNA executes the DNN computation within the GNN model. The AGG performs feature aggregation coordinated by the GPE based on graph traversal. The DNQ buffers memory requests and intermediate results as they are passed to the DNA. This design allows for easy scalability by interconnecting multiple tiles with memory.

The GNN accelerator program proposed by Auten *et al.* [3] represents a GNN model as a sequential set of layers. Each layer operates on a graph, applying a vertex program to generate an output graph. These layers are connected in sequence to form a complete GNN model. The initial layer takes the model input as its input graph, and subsequent layers utilize the output of the preceding layer. The last layer produces the final output graph.

3.3.3 Hybrid architectures for GNN acceleration

HyGCN [44] is a unique GCN accelerator due to its innovative hybrid architecture. This approach was inspired by the observation that GNNs exhibit two distinct execution patterns with contrasting requirements: the aggregation phase involves graph processing, displaying a dynamic and irregular execution pattern. On the other hand, the combination phase behaves more like conventional neural networks, exhibiting a static and regular execution pattern. As a result of this observation, HyGCN consists of dedicated engines for the aggregation and combination stages and a coordinating mechanism for pipelined execution of both functions.

The Combination operation at each vertex works like a neural network with a regular yet compute-intensive execution. HyGCN's architecture is based on the popular systolic array, but it incorporates multiple arrays instead of a single one to adapt to the two processing modes of the Aggregation Engine. In the combination engine, a set of systolic arrays is combined to form a systolic module, and these modules can be flexibly utilized in various ways, including independent and cooperative working modes.

In the independent working mode, the systolic modules operate autonomously, each handling the matrix-vector multiplication (MVM) operations of a small group of vertices. This mode offers the benefit of reduced vertex latency since the Combination operations for this smaller group of vertices can be processed immediately once their aggregated features are ready without waiting for additional vertices. In the cooperative working mode, a large group of vertices' aggregated features are gathered and combined. The advantage of this mode is that weight parameters can be efficiently reused by all systolic arrays, reducing energy consumption.

The aggregation engine, represented in Figure 3.4, comprises a sampler, edge scheduler, and sparsity eliminator feeding a set of SIMD (single instruction multiple data) cores. There are two processing modes for SIMD cores to handle edges in parallel.

The first mode is vertex-concentrated, where each SIMD core is assigned the workload of a single vertex. While this mode can produce aggregated features in a burst mode, the processing latency for a single vertex is prolonged, leading to workload imbalance and loss of parallelism. On the other hand, the vertex-disperse processing mode assigns the aggregation of elements in the vertex feature vector to all cores. This mode ensures that all cores are constantly busy without workload imbalance. Additionally, it enables immediate processing of each vertex in the subsequent Combination Engine while reducing the latency for a single vertex compared to processing multiple vertices together. To enhance the computation of aggregation, HyGCN uses the vertex-disperse processing

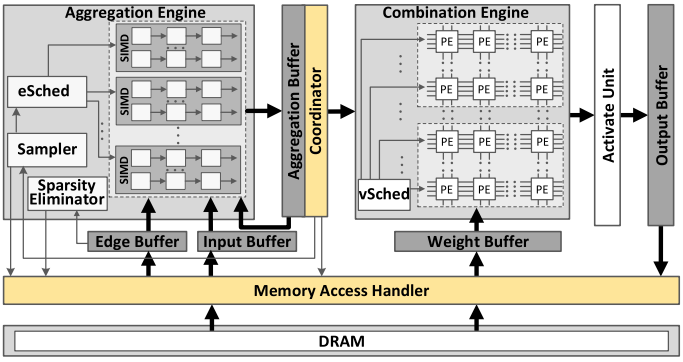


Figure 3.4: Architecture overview of HyGCN [44]

mode.

HyGCN utilizes a static graph partition method to optimize memory access to improve data reuse. The authors identified that the feature vectors of each vertex are typically large, making the exploitation of feature locality crucial. To address this, they grouped vertices in disjoint sets and processed the aggregation of their source neighbors group by the group. By following this approach, the feature accesses of all vertices in an interval were merged. This grouping allowed for overlapping neighbors within the considered interval, enabling the reuse of loaded feature data during feature aggregation. Moreover, when traversing all the neighbors of the interval, the intermediate aggregated results of the grouped vertices were stored in a buffer and could be reused during feature updates.

Sparsity is efficiently handled at the aggregation engine through effective scheduling and the sparsity eliminator, which adapts dynamically to varying degrees of sparse multiplications using a window-based sliding and shrinking approach. In particular, the authors implemented this approach to enhance data reuse and minimize redundant accesses caused by sparse graph connections. The central idea was to slide the window downward until an edge appeared in the top row and then shrink its size by moving the bottom row upward until an edge was encountered. This method effectively eliminated sparsity and improved data access efficiency.

To further optimize for varying workloads, HyGCN allows flexible grouping of SIMD cores in aggregation and PEs in combination based on the size of feature vectors. Additionally, careful attention is given to the design of the inter-engine coordinator to optimize memory accesses and enable fine-grained pipelining of execution, maximizing parallelism dynamically.

The internal structure of each tile in the accelerator proposed in [3] resembles HyGCN's, with the DNA functioning as an array for dense multiplication, the AGG as an edge-

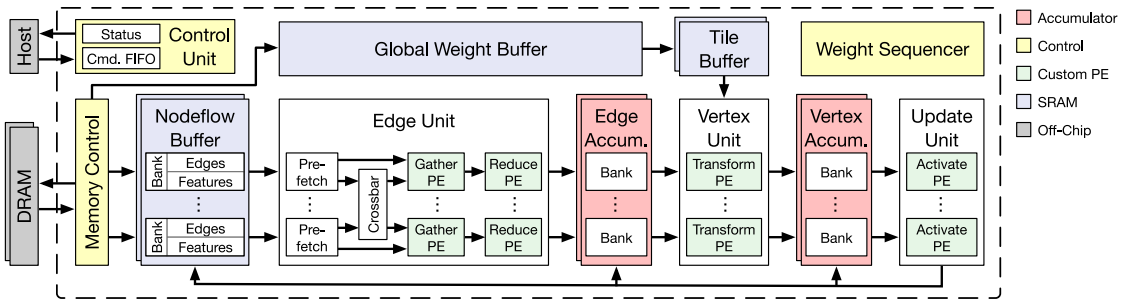


Figure 3.5: High-level overview of GRIP [24]

controlled adder, the DNQ as an inter-engine buffer, and the GPE overseeing execution. Unlike HyGCN, the accelerator introduced in [3] is less specialized and has a better potential for generalization to various Graph Neural Network models [1].

While not an authentic hybrid architecture, GRIP [24] is an accelerator that shares similar techniques with HyGCN's implementation approach. It leverages GReTA [23] (Gather, Reduce, Transform, Activate), a graph processing abstraction specifically crafted for efficient execution on accelerators. It also offers the flexibility required to implement GNN inference and holds the potential to be adaptable to various types of Graph Neural Networks.

GRIP is an accelerator designed to achieve low-latency inference. It addresses the challenges of accelerating GNNs, combining two distinct computation types: arithmetic-intensive vertex-centric operations and memory-intensive edge-centric operations. To tackle this, the accelerator divides GNN inference into fixed sets of edge- and vertex-centric execution phases, making them suitable for hardware implementation. Each unit is then specialized to handle the unique computational structure of each phase efficiently.

GRIP utilizes a high-performance matrix multiply engine and a dedicated memory subsystem for vertex-centric phases for weights to enhance data reuse. In contrast, it employs multiple parallel prefetches and reduction engines for edge-centric phases to mitigate the irregularity in memory accesses. Additionally, GRIP supports several GNN optimizations, including a novel technique called vertex-tiling, which enhances the reuse of weight data.

GRIP provides a customizable architecture, shown in Figure 3.5, with separated and custom units and accumulators for both edges (gather, reduce) and vertices (transform, activate) that allows for performing edge and node updates using user-defined functions. The control of GRIP is managed by a host system that issues commands for different operations and data transfers. The control unit dequeues these commands in order and asynchronously issues them to individual execution units or the memory controller.

GRIP comprises three core execution units: the edge unit, the vertex unit, and the update unit. The edge unit performs the edge-accumulate phase, iterating over the edges of the nodeflow, executing gather, and accumulating the result into the edge accumulator using reduce. The vertex unit performs the vertex-accumulate phase, iterating over the output vertices corresponding to the accumulated edge values, executing the transform, and accumulating the result into the vertex accumulator. The update unit performs the vertex-update phase, reading the accumulated values for each vertex and passing them to the activated PE. The result is then written to the nodeflow buffer as an updated feature or to the edge or vertex accumulator, enabling efficient data flow between different GRIP programs when executed in sequence.

As already said, GRIP allows users to customize the four PEs, which can be implemented in multiple ways based on their specific requirements. In the authors' implementation, a programmable ALU-based approach is used, splitting the edge update unit into lanes to execute vertices simultaneously. It adopts an input-stationary dataflow for the vertex update unit. The accelerator employs various optimizations, including pipelining and tiling adapted to the specific dataflows implemented, similar to other accelerators.

3.3.4 Challenges

This section presented a variety of GNN hardware accelerators, but despite many alternatives, there is still the need to face some open challenges. In particular, most of the presented solutions target only specific models of GNNs, and it is unclear how to generalize them.

Hardware accelerators face the challenge of striking the appropriate balance between performance and generalization, considering the multitude of graph types and GNN variants. Moreover, the wide variety of problems, each with varying graph and feature vector sizes, increases the complexity of the acceleration task.

Given that the field of GNN accelerators is constantly evolving, researchers are actively investigating innovative architectural and algorithmic solutions. This thesis aims to introduce a novel GNN accelerators design technique that can be tailored to specific requirements, offering an alternative to existing solutions with the advantage of a faster and automated design.

3.4 High-Level Synthesis based accelerators

As previously highlighted, one of the main challenges of GNN hardware acceleration lies in simultaneously meeting the demand for supporting novel GNN models and fast inference, as there exists a gap between the development time of efficient FPGA accelerators and the rapid evolution of new GNN models.

Unlike the designs presented in the previous section, typically designed using low-level hardware description languages by explicitly specifying the hardware components and their interconnections, this section discusses methods to generate accelerators using HLS, which automatically synthesizes high-level code into hardware, abstracting away low-level hardware details.

FlowGNN [33] is a GNN acceleration framework utilizing High-Level Synthesis, with two primary objectives. Firstly, to achieve ultra-fast GNN inference without needing graph pre-processing to meet real-time demands. Secondly, to support a wide range of GNN models with the flexibility to accommodate new models. The framework incorporates an optimized message-passing structure that applies to all models and is complemented by a diverse library of model-specific components.

This framework capitalizes on the observation that each node in a GNN layer undergoes two key steps: message passing (MP) and node transformation (NT). The message passing step is further divided into gather and scatter phases, where gather involves feature aggregation, and scatter entails message transformation and forwarding. On the other hand, node transformation usually consists of a linear layer or multi-layer perceptron, followed by node update.

To accelerate MP and NE steps, FlowGNN was designed using a message-passing style featuring two main processing elements (PEs): node transformation and message passing. The authors proposed a baseline architecture and an improved one, as shown in Figure 3.6. The baseline architecture includes three data storage buffers: one node embedding buffer and two message buffers, all of which have a $O(N)$ size, where N represents the number of nodes allowed on-chip. The two message buffers are used alternately across layers, allowing for the reuse of resources and dataflow in multiple layers. The node transformation PE handles node transformation and update within a single layer, while the message passing PE performs the subsequent scatter operation. The advantage of this approach is that the receivers of the messages can instantly update their partially aggregated message in the message buffer, enabling the merging of scatter and gather phases. Since the aggregation function is permutation invariant and the aggregation order does not matter, such a

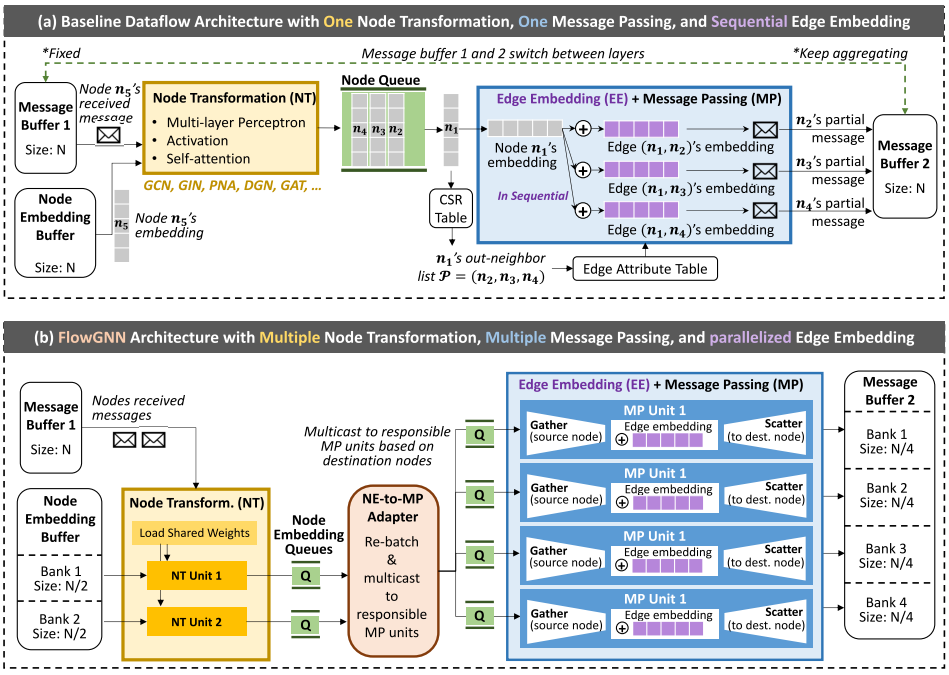


Figure 3.6: Baseline dataflow architecture and the improved FlowGNN architecture [33]

merged fashion reduces the overall process latency and minimizes memory cost.

The independence of node transformation and message passing steps across nodes and edges allows for significantly reduced processing latency by effectively pipelining these two steps. The authors referred to the most suitable approach for this task as baseline dataflow pipeline. In this implementation that significantly reduces idle cycles and minimizes resource usage, NE and MP are pipelined flexibly using a node queue. Once a node completes its NE and is prepared for message passing, its embeddings are pushed into the queue. At the same time, the MP engine reads from the queue, fetching the node embeddings for message passing.

While the baseline dataflow architecture can efficiently pipeline NT and MP tasks, it has several limitations. These include its ability to handle only one node and one edge at a time, limiting parallel processing capabilities. Furthermore, NT and MP operations cannot be concurrently executed within a single node, necessitating the completion of NT before initiating MP.

The authors introduce the FlowGNN architecture, shown in Figure 3.6, to overcome these limitations and significantly enhance performance through various levels of parallelism. FlowGNN's capabilities can be customized using four parameters: P_{node} determines the number of nodes that can undergo simultaneous processing by NT, P_{edge} sets the simultaneous processing of edges by MP, P_{apply} specifies the concurrent processing of node

embedding dimensions by a single NT, and $P_{Scatter}$ sets the simultaneous processing of edge embedding dimensions by a single MP.

The FlowGNN architecture incorporates multiple NT units and multiple MP units, interconnected through an NT-to-MP adapter that utilizes multiple data queues. To facilitate concurrent access, both the node embedding and message buffers are divided into several banks.

Achieving Node/Edge Parallelism necessitates that edges and their respective target nodes reside in distinct memory banks or buffers. In the absence of prior graph preprocessing, such access patterns are typically random, making parallelization unfeasible. To tackle this challenge, the authors have introduced an innovative multi-queue dataflow approach, using an NT-to-MP adapter and implementing real-time multicasting, allowing the process of P_{node} nodes in parallel as a single batch enabling concurrent NT and MP operations through dynamic node distribution. The core concept involves instantiating multiple NT and MP units, with each MP unit responsible for processing its designated set of edges and managing read/write operations for its allocated bank of node embeddings.

Parallelism in apply/scatter operations is achieved through the utilization of NT and MP units. The NT unit is responsible for conducting any per-node computations necessary for a GNN, and it comprises two consecutive processes, synchronized among nodes using ping-pong buffers to mitigate latency: accumulate and output. During the accumulate phase, each node's aggregated message is read, and a fully-connected layer is computed in an input-stationary manner, where each element of the input vector fetched is used to update the entire output vector. Subsequently, once the accumulate process is complete for an entire node, the output phase carries out any finalization tasks, such as applying an activation function, before forwarding the resulting embedding to the multicasting adapter.

The MP units are responsible for executing computations specific to each edge. Two configurations are available: NT-to-MP or MP-to-NT dataflow. In the former each MP unit manages an independent subset of destination nodes. Instead, in latter, each MP unit is allocated a subset of source nodes. Within this setup, the MP units collect partial messages from nodes within their assigned subset along the edges.

Finally, another feature provided by FlowGNN is its adaptability to different graph neural network models by offering various model-specific components. One particularly advantageous feature of FlowGNN's dataflow architecture for node/edge processing is its suitability for models with virtual nodes. As defined by [16], a virtual node acts as an artificial node connected to all other nodes in the graph, creating a shortcut for message

passing between node pairs. The authors stated that processing the virtual node can be entirely overlapped with the node embedding computation for other nodes, ensuring zero waste as long as it is handled early enough in the processing pipeline.

Another notable framework in this section is DGNN-Booster [9], an innovative Field-Programmable Gate Array accelerator framework designed for real-time inference of Dynamic Graph Neural Networks (DGNNs) using High-Level Synthesis. Unlike the other accelerators mentioned and outside the scope of this research, DGNN-Booster focuses on DGNNs, which are Graph Neural Networks tailored for dynamic graph structures and features. However, it is worth mentioning that DGNN-Booster implements GNNs using a message-passing mechanism based on FlowGNN at a lower level of parallelism.

3.5 Software-Hardware co-design accelerators

Software-hardware co-design is an integrated approach which involves designing both software and hardware components collaboratively to enhance system performance and efficiency. Unlike traditional design methods where software and hardware are developed independently and later integrated, co-design seeks to unify these two elements from the beginning of the design process. This design strategy evaluates which tasks are suitable for offloading to software and which should remain in hardware, considering the associated overheads.

The work conducted by Zhang *et al.* [47] introduces a combined software and hardware approach for accelerating Graph Convolutional Networks. Their study was initiated with the recognition that hardware acceleration of Graph Convolutional Network inference poses challenges stemming from the vast size of the input graph, the heterogeneous workload of GCN inference involving sparse and dense matrix operations, and the irregular information propagation along the edges during computation

The primary objective of this accelerator is to accelerate GCN models, with a particular focus on the critical computational kernels: feature aggregation AX and feature transformation XW. In these kernels, A represents the adjacency matrix, X denotes the feature matrix, and W represents the weight matrix.

The proposed algorithm-architecture co-optimization for accelerating large-scale Graph Convolutional Network inference on FPGA involves several key steps. First, the authors implemented a data partitioning scheme for GCN inference to accommodate real-world datasets with huge dimensions for A and X. This approach ensures that both the adjacency matrix and the feature matrix can fit on-chip while mapping the computational

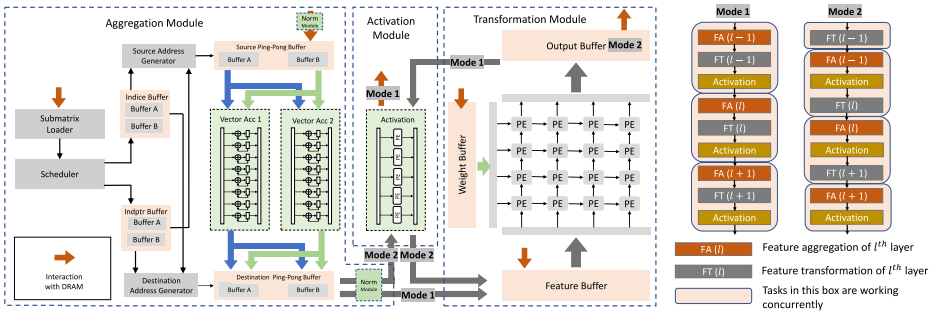


Figure 3.7: Hardware architecture with two execution modes proposed by Zhang *et al.* [47]

kernels onto the FPGA.

Then, the graph undergoes a two-phase preprocessing algorithm involving sparsification and node reordering. The sparsification phase eliminates edge connections of high-degree nodes by merging familiar neighbors to reduce the memory accesses that a graph with more edges can require during the aggregation stage. The node reordering phase effectively groups adjacent nodes to enhance on-chip data reuse.

The pre-processed graph is then fed into a hardware accelerator implemented in an FPGA that efficiently pipelines GCN's two major computational kernels: aggregation and transformation. As outlined in [1], the design, shown in Figure 3.7, distinguishes itself from other approaches in several ways. The aggregator module adopts a double-buffering technique to hide addition latency and leverages node- and feature-level parallelism. Moreover, the accelerator supports two modes of operation depending on the order of matrix multiplications, leading to different pipelining strategies. In order to accommodate these modes, the modules are interconnected from the aggregate module to the combination modules and vice versa.

FlowGNN's NE/MP pipeline, introduced in Section 3.4, shares a similar concept with the task scheduling approach of BoostGCN [48]. BoostGCN presents a framework, shown in Figure 3.8, using an algorithm-architecture co-optimization scheme tailored to enhance GCN inference on FPGA. The authors introduced a groundbreaking hardware-aware Partition-Centric Feature Aggregation (PCFA) scheme that capitalizes on 3-D partitioning alongside the vertex-centric computing paradigm. This innovation significantly boosts on-chip data reuse while minimizing the overall data communication volume with external memory.

Furthermore, they devised a novel hardware architecture that enables seamless pipelined execution of the computation phases of inference: feature aggregation and feature update. They developed a low-overhead task scheduling strategy to tackle any potential pipeline

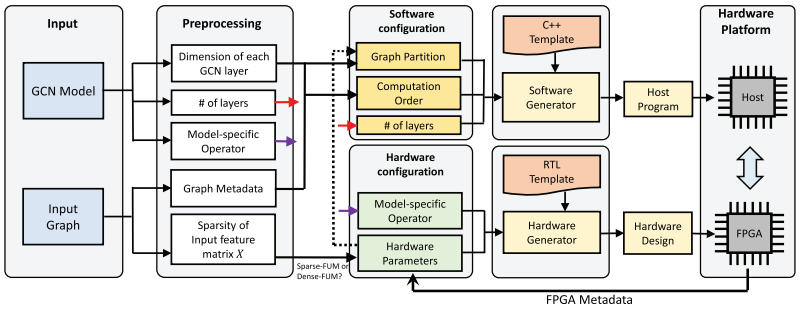


Figure 3.8: BoostGCN framework overview [48]

stalls arising from these phases.

The authors delivered a comprehensive GCN acceleration framework on FPGA, complete with meticulously optimized RTL (Register-Transfer Level) templates. This framework can generate hardware designs based on personalized configurations and is adaptable to diverse GCN models. BoostGCN's overall system architecture comprises external memory and FPGA components. Feature Aggregation Modules (FAMs) handle feature aggregation on the FPGA board, while Feature Update Modules (FUMs) manage feature updates. Intermediate results generated by FAMs are cached in the Internal Buffer, and the Memory Controller manages data transmissions between external memory and hardware modules.

The authors propose a specific approach to optimize task scheduling and minimize pipeline stalls for FUM and FAM. This involves arranging intervals based on their vertex degrees and prioritizing intervals with more minor vertex degrees for execution first. Furthermore, they allocate a buffer in external memory to store aggregated feature vectors produced by FAMs in case FUM is not yet prepared to consume new aggregated feature vectors. Subsequently, FUM can retrieve the aggregated feature vectors from external memory when ready.

As indicated in [33], while the scheduling approaches of FlowGNN and BoostGCN share some similarities, there are notable differences. Firstly, BoostGCN relies on sorting vertices by degrees on the CPU to establish an execution order, whereas FlowGNN processes the nodes on-the-fly in FPGA in an adaptive manner. Secondly, BoostGCN employs a buffer in external memory, while FlowGNN utilizes an on-chip FIFO to queue the nodes ready for message passing.

GCoD [46] is another accelerator that follows the dedicated Algorithm and Accelerator Co-Design approach. It is a framework combining Graph Convolutional Network algorithm and accelerator design to improve GCNs' inference efficiency significantly.

GCoD incorporates a split-and-conquer GCN training strategy at the algorithm level, dividing graphs into denser or sparser local neighborhoods without sacrificing model accuracy. This approach leads to adjacency matrices with mainly two levels of workload, enabling more effortless acceleration.

GCoD's Split and Conquer Algorithm aims to tackle the high sparsity and irregularity in GCNs' adjacency matrices through subgraph classification, enforcing regularity at different granularities. Nodes with similar degrees are clustered into classes, and each class is further divided into subgraphs with similar edge counts. This approach fosters regular and efficient hardware acceleration, with each sub-accelerator processing one subgraph.

Additionally, Group Partitioning manages uniformly grouped subgraphs of the same class, reducing boundary connections to enforce sparser patterns. This grouping strategy simplifies hardware designs and communication among sub-accelerators, further enhancing processing efficiency.

The authors design a specialized two-pronged accelerator on the hardware level, with separate engines for processing denser and sparser workloads, maximizing overall utilization and acceleration efficiency. The GCoD accelerator is designed with two separate computing branches, each dedicated to processing the denser and sparser workloads resulting from the GCoD algorithm's adjacency matrices.

The Denser Branch utilizes an array of parallel sub-accelerators to process the enforced regular dense subgraphs along the diagonal line of the adjacency matrices. This approach efficiently handles the more intense workload while maintaining workload balancing through proportional resource allocation among the sub-accelerators.

Meanwhile, the Sparser Branch efficiently handles the remaining irregular but lightweight sparser workloads, mostly on-chip. This design minimizes frequent and large-volume data movements from the off-chip memory, improving overall processing efficiency.

In each sub-accelerator within the branches, there are dedicated Buffers that enhance local reuse opportunities, a Sparse/Dense Matrix Multiplication Engine (SpMM) capable of handling both dense and sparse matrix multiplication, element-wise Activation Units for non-linear activation operations, and sampling Units for efficient node sampling scheduling.

3.6 Graph processing acceleration

This section delves into the state-of-the-art accelerators for graph processing. Although not directly tailored for graph neural networks, graph processing is a fundamental aspect of GNN acceleration, particularly for models like Graph Convolutional Networks. As mentioned earlier in this chapter, numerous accelerators have prioritized graph processing to enhance GNN performance.

GraphLily [20] is a graph linear algebra overlay designed to accelerate graph processing on FPGAs equipped with high-bandwidth memory (HBM). Given the low compute-to-memory access ratio and irregular data access pattern, memory access often limits graph processing. HBM's exceptional bandwidth, with multiple channels servicing memory requests concurrently, has the potential to enhance graph processing performance significantly.

GraphLily supports a diverse set of graph algorithms using the GraphBLAS [21] programming interface, which formulates graph algorithms as sparse linear algebra operations. GraphBLAS establishes a fundamental collection of matrix-based graph operations, enabling the implementation of a broad array of graph algorithms across various programming environments.

The accelerator provides efficient, memory-optimized implementations for two widely-used GraphBLAS kernels: sparse-matrix dense-vector multiplication (SpMV) and sparse-matrix sparse-vector multiplication (SpMSpV). The SpMV accelerator is specifically designed to fully utilize the HBM bandwidth, facilitating efficient pull-based graph processing. To achieve this, the authors introduced a novel sparse matrix storage format that explicitly captures and encodes both intra-channel and inter-channel memory-level parallelism, effectively harnessing the parallel capabilities of the accelerator. Its design comprises multiple PE clusters, each connected to one HBM channel.

The SpMSpV accelerator complements the SpMV accelerator, explicitly catering to push-based graph processing, which is particularly advantageous for highly sparse input vectors. Its architecture is different from the SpMV one. It comprises a vector loader, a matrix loader, and an arbitrated crossbar. The vector loader is responsible for loading the non-zero elements of the sparse input vector from HBM. The matrix loader loads packets of the corresponding columns of the sparse matrix from DDR, decoding them into separate streams. Finally, the arbitrated crossbar dispatches these streams based on the row IDs to an array of PEs, each accessing independent banks of the output buffer.

GraphLily incorporates a middleware that presents each accelerator as a module, effec-

tively linking the GraphBLAS interface and the overlay. This modular approach enables users to construct graph algorithms by specifying the required modules and scheduling their execution order. Each module provides a set of APIs that facilitate data transfers between the host and the device and between different devices. Host-to-device and device-to-host data transfers occur only once before or after the iterations of the graph algorithm, ensuring their costs are amortized. Meanwhile, device-to-device data transfers facilitate the exchange of intermediate results during the iterations, minimizing the need for frequent data transfers to the host and back.

3.7 Matrix multiplication optimization

Significant efforts have been dedicated to accelerating matrix multiplication, resulting in numerous libraries designed for various platforms. Basic Linear Algebra Subprograms (BLAS) specify low-level routines for executing fundamental linear algebra operations, including matrix multiplication. OpenBLAS [42] is a prominent optimized BLAS library tailored for CPU usage. At the same time, cuBLAS [30] serves as a specialized library providing GPU-accelerated implementations of BLAS.

One example of FPGA-accelerated BLAS library is GraphLily, discussed in Section 3.6, designed to exploit FPGA's parallelism and hardware capabilities for efficient matrix operations. Various custom implementations are crafted to suit specific FPGA platforms and applications.

Particular attention through a limited selection of state-of-the-art techniques will be given to the optimization of matrix multiplication using MLIR. This focus arises because SODA, the framework introduced in Section 2.6, which represents part of the core of this thesis, represents an extension of the MLIR framework.

3.7.1 Matrix multiplication optimization in MLIR

The work presented in [6] aimed to reimagine the optimization approach of OpenBLAS in a compiler-oriented fashion using MLIR. MLIR was explicitly designed to offer a unified, modular, and extensible infrastructure, facilitating the gradual lowering of dataflow compute graphs, potentially through loop nests, to high-performance target-specific code.

The authors of [6] chose to base their exploration on the matrix-matrix multiplication (matmul) algorithm. This choice is because matrix-matrix multiplication is an excellent routine for demonstrating code optimization practices in tutorials and a crucial operation in various domains. Consequently, it is often the first task for which developers create an

optimized implementation when working with a new architecture.

The authors used an Intel i7-8700K CPU at 3.70GHz frequency for all experiments and achieved a nearly 3x improvement in performance through their first optimization, which involved using the cache tiling strategy employed in OpenBLAS. This strategy carefully tiles the matrices to exploit reuse at different cache levels. The primary objective is to ensure that the vector of Fused Multiply-Add (FMA) units, designed to perform a mathematical operation known as a fused multiply-add, which combines a multiplication and an addition in a single instruction, remains full, avoiding waiting for loads.

The explicit copying or packing technique, where accessed data is first copied or packed into contiguous buffers and then indexed for computation, is commonly employed when dealing with code involving multidimensional arrays that exhibit reuse. By employing such copying techniques, the reduction or near elimination of conflict misses, Translation Lookaside Buffer (TLB) misses, and improved hardware prefetching performance can be achieved. This approach, in combination with tiling, allows for the exploitation of reuse in multiple directions when the data accessed fits in a higher level of the memory hierarchy. However, it also addresses the issue where data accessed for a tile is no longer contiguous in the original matrix/tensor. This leads to conflict misses, TLB misses, and more prefetch streams, potentially negating some of the gains even with high reuse. As a result, this approach yields a substantial performance improvement of nearly 1.5x.

Another applied optimization involves the unroll-and-jam of the innermost two loops, followed by scalar replacement in MLIR post unroll-and-jam. This process converts reduced memref (the in-memory representation of a tensor in MLIR) locations into scalars (single-element memrefs) and hoists them, eliminating redundant loads and lifting invariant loads out of loops. This step significantly improved overall performance, resulting in an impressive 10x speedup.

The final technique employed was vectorization. It consists in organizing data in vectors and processing multiple elements of these vectors in parallel, and it yielded a remarkable 4.5x improvement. Combining all the abovementioned techniques and carefully selecting the appropriate tiling parameters to optimize register and cache utilization, the overall performance achieved was only 10% less than OpenBLAS.

3.8 Conclusion

The analysis of the state of the art in previous sections yields several conclusions. Firstly, the one-size-fits-all approach does not apply to GNNs, and different applications will

likely require distinct design approaches. It is evident that specific accelerators are more suitable and have been specifically designed to accelerate particular models.

Secondly, a quantitative comparison among accelerators is challenging due to the absence of a standard baseline system and a GNN benchmark suite encompassing a representative set of algorithms, datasets, and design objectives. To address this issue, initiatives like the Open Graph Benchmark (OGB) [19], which will be discussed in Chapter 6, aim to provide a representative set of graphs and GNNs for benchmarking purposes.

4 Problem Formulation

This chapter aims to formulate the problem rigorously, explaining the thesis's objective, motivation, and research questions that guided this work.

4.1 Graph Neural Network acceleration

Graph neural network acceleration refers to designing and implementing hardware accelerators and co-processors to speed up the training or inference of GNNs. A GNN accelerator aims at optimizing the execution of GNN computations, which involve iterative message-passing between nodes in a graph to update their representations based on neighboring nodes' features. Hardware acceleration aims to improve the performance, efficiency, and capabilities of computing systems by offloading specific tasks or computations to specialized hardware components.

In particular, this thesis focuses on improving GNNs inference time, by designing specialized hardware to efficiently perform the computation-intensive operations involved in GNNs, such as matrix multiplications, aggregations, and non-linear activation functions.

A small example explains the potential impact of this thesis's objective. Let us consider a recommendation system, in which the goal is to predict what items a user might be interested in based on their past interactions and preferences. An example is Netflix suggesting what to watch next based on previously watched movies and ratings. This problem can be represented as a graph, where users and movies (items) are nodes and interactions between users and movies are edges. A GNN is an optimal choice for modeling recommendation systems, as it can effectively capture relationships and interactions between users and items. However, as the number of users and items increases, the computational complexity of GNNs can become a significant bottleneck. One way to address the bottleneck issue is by using a specialized GNN accelerator, which is purpose-built to handle GNN computations on extensive graphs efficiently. By exploiting parallelism and data locality in GNN operations, it enables faster and more energy-efficient processing of the graph.

Utilizing the GNN accelerator, the recommendation system can offer real-time recommendations to users, even on edge devices with limited computational capabilities. Ultimately, the GNN accelerator enhances the efficiency and scalability of the recommendation system, resulting in quick and precise recommendations to users. It also reduces computational and memory overhead.

4.2 Motivation and objective

As mentioned in Chapter 3, different state-of-the-art accelerators exist and use different approaches to improve GNNs' performance. Even if various alternatives are available, almost all GNN acceleration research has implicitly focused on either developing highly efficient schemes tailored for specific GNN models or aiming for generality and flexibility to accommodate various types of GNNs with less efficiency.

The primary challenge driving the research in this thesis lies in creating a framework that optimizes performance and efficiency while retaining the necessary flexibility to adapt to diverse graph sizes, characteristics, and GNN algorithms.

The main research questions that led to this thesis can be summarised as follow:

1. How is it possible to design hardware accelerators that exploit the unique characteristics of GNN computation?
2. How is it possible to synthesize accelerators, starting from high-level programming languages, without being a hardware design expert?
3. What are the bottlenecks of GNNs that can slow down their inference time?
4. What are the most effective low-level optimizations for GNNs that can improve their efficiency without sacrificing model accuracy?
5. Can an automated design be generalized to different GNN models and datasets while ensuring high performance?

Consequently, the main objectives of the thesis are:

1. Investigate existing GNN models and identify bottlenecks that hinder their inference efficiency.
2. Develop an FPGA toolchain for GNN acceleration, allowing seamless integration with various GNN models and datasets.
3. Explore low-level optimizations for GNNs to improve hardware performance.

4. Synthesize hardware accelerators tailored to GNNs, leveraging parallelism and memory optimizations to accelerate graph computations.
5. Implement and evaluate the proposed GNN accelerators on FPGA.

5 Design of the Toolchain

The main contribution of this thesis is represented by the design of a toolchain for Graph Neural Network acceleration on FPGA leveraging High-Level Synthesis.

This chapter explains in detail how the toolchain has been designed and how it can be used to build GNN accelerators to enhance inference performance.

The core component of the toolchain is the synthesizer, enclosing SODA-OPT [2] and PandA-Bambu [13]. The primary objective of this thesis is to enhance SODA-OPT and PandA-Bambu to synthesize GNN models written in high-level frameworks, such as PyTorch, to FPGA architectures.

Figure 5.1 illustrates the entire design flow, with the steps involved in the GNN acceleration process. In particular, firstly, the GNN model is implemented in PyTorch, one of the most popular and powerful frameworks for Neural Network implementations. Subsequently, the model is passed as input to Torch-MLIR, a crucial middle step that enables the generation of the MLIR representation. This intermediate representation serves as input for the synthesizer, where, once the frontend optimization is complete, the refined version proceeds to the backend, where the actual GNN accelerator is effectively produced, ready to enhance inference performance on FPGA architectures.

The following sections provide a comprehensive and in-depth exploration of each step within the proposed design flow. This detailed breakdown highlights the various possibilities inherent in the toolchain and outlines the recommended procedures necessary to achieve the optimal outcome for GNN acceleration. A final section explains how matrix multiplication acceleration has been addressed, focusing on a theoretical analysis of the algorithm and the heuristics followed in the phase preceding the experiments. The thesis aims to equip researchers and practitioners in Graph Neural Networks with the necessary insights and understanding to harness the full potential of this toolchain and unleash the power of FPGA acceleration.

In conclusion, this thesis represents a significant advancement in Graph Neural Network acceleration. By designing a refined toolchain and bridging the gap between high-level

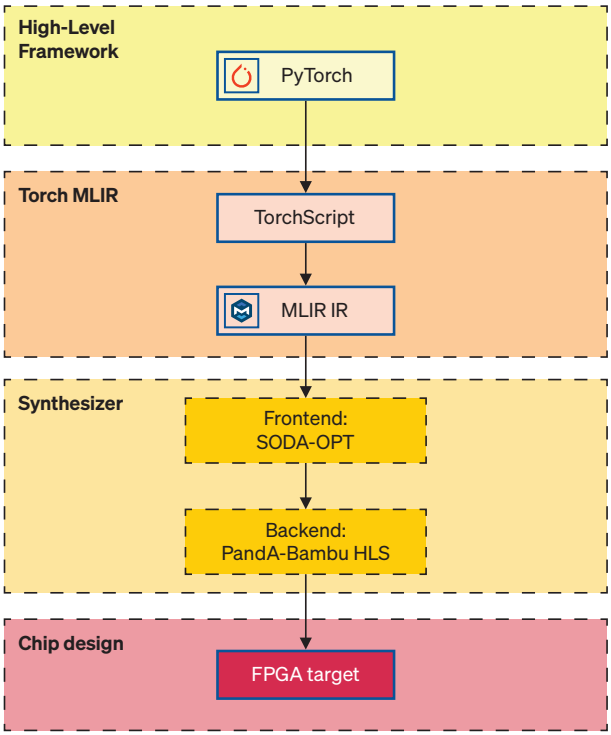


Figure 5.1: FPGA Toolchain for Graph Neural Network Acceleration

frameworks and FPGA architectures, this research contributes to the broader field of artificial intelligence. It reinforces the potential of FPGA-based accelerators in revolutionizing the inference performance of GNN models. The implications of this work offer a solid foundation for further exploration and advancements in the field of hardware acceleration for deep learning applications.

5.1 PyTorch

PyTorch [31] is an open-source deep learning framework widely used for building and training artificial neural networks for various machine learning tasks.

The first step of the toolchain is to design and implement the Graph Neural Network model in PyTorch. Doing so involves defining the GNN model architecture and writing the necessary forward pass to compute node and graph-level representations. Once having defined the model, the next step is training the GNN using standard PyTorch techniques, such as defining a loss function, setting up an optimizer, and performing backpropagation to optimize the model parameters.

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3 from pygcn.layers import GraphConvolution
4
5 class GCN(nn.Module):
6     def __init__(self, nfeat, nhid, nclass, dropout):
7         super(GCN, self).__init__()
8
9         self.gc1 = GraphConvolution(nfeat, nhid)
10        self.gc1 = GraphConvolution(nhid, nclass)
11        self.dropout = dropout
12
13    def forward(self, x, adj):
14        x = F.relu(self.gc1(x, adj))
15        x = F.dropout(x, self.dropout,
16                      training=self.training)
17        x = self.gc2(x, adj)
18        return F.log_softmax(x, dim=1)
```

Listing 5.1: Class of GCN model

```
1     def forward(self, input, adj):
2         support = torch.mm(input, self.weight)
3         output = torch.spmm(adj, support)
4         if self.bias is not None:
5             return output + self.bias
6         else:
7             return output
```

Listing 5.2: Forward function of GCN layer

5.1.1 Toolchain inputs

Two main models have been used as input for the developed toolchain: a Graph Isomorphism Network from OGB [19, 34], written using PyTorch Geometric [14], and a Graph Convolutional Network [25, 37], written using PyTorch [31].

Most research and experiments have been conducted using the GCN model. The GCN class, shown in Listing 5.1, is characterized by two Graph Convolutional layers, and the forward function of each layer, shown in Listing 5.2, is implemented through two matrix multiplications.

OGB provides different datasets that can be used with their models. The one used for

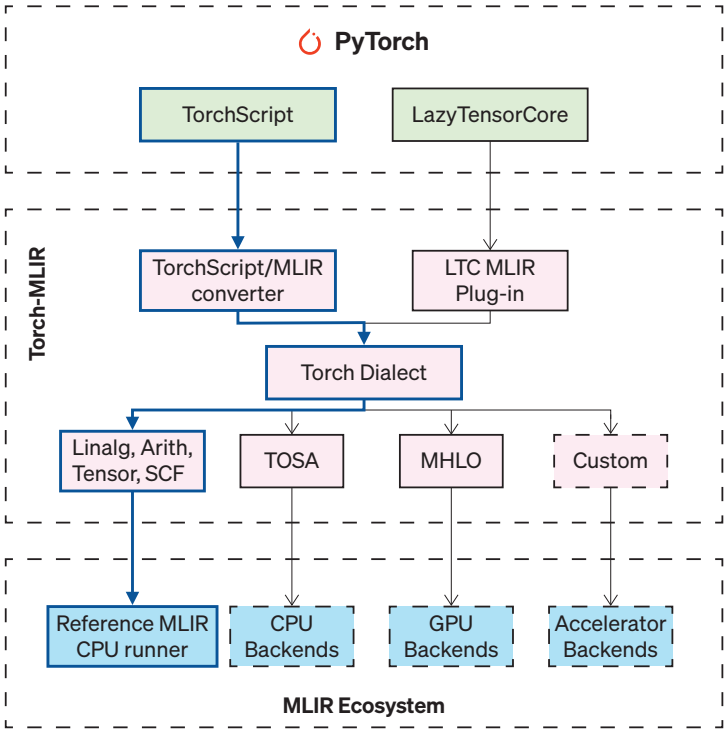


Figure 5.2: Torch-MLIR flow

this thesis is called *ogbg-molhiv*, a molecular property prediction dataset. In each graph representing a molecule, nodes correspond to atoms, and edges represent chemical bonds. The input node features consist of nine dimensions, encompassing information like atomic number, formal charge, and whether the atom is part of a ring. The binary classification task for GIN consists in achieving precise predictions of target molecular properties, for example, determining whether a molecule inhibits HIV replication or not.

The dataset used for the GCN model is the *Cora* one. This dataset contains 2708 scientific publications, categorized into one of seven classes. The citation network contains 5429 links. Each publication in the dataset is represented by a binary-valued word vector, indicating the absence or presence of the corresponding word from a dictionary of 1433 unique words. The task is a multiclass classification, in which, given a paper, the objective is to classify it into one of the seven classes correctly.

5.2 Torch-MLIR

Torch-MLIR [38] offers compiler support for transitioning from the PyTorch ecosystem to the MLIR ecosystem.

The steps Torch-MLIR follows to go from PyTorch to MLIR are shown in Figure 5.2. In

particular, the flow followed in this thesis has been highlighted with blue arrows. There are two starting points of the flow: TorchScript and LazyTensorCode. The one used for this research, which is also the most tested one, is TorchScript. TorchScript [36] offers a way to generate serializable and optimizable models directly from PyTorch code.

The TorchScript representation is then converted to MLIR using the built-in conversion of Torch-MLIR. The resulting MLIR IR can use different dialects that can be selected by the user, but the one used for this thesis is the Linalg dialect, which serves as input for the next phase of the toolchain.

5.2.1 From PyTorch to TorchScript

Since Torch-MLIR implicitly uses the TorchScript representation to go from PyTorch to MLIR, the first part of the research consisted of a deep analysis of the Graph Neural Network models to make them compatible with TorchScript.

This task required more effort for the GIN model as it uses functions from PyTorch Geometric than the GCN model, which only uses pure PyTorch operations. The following adaptations have been applied to both models, and they can be applied to make any GNN compatible with TorchScript.

- The GNN layer class, if created as a subclass of the Message Passing class, must be marked as Jittable whenever it is used.

```
self.convs.append(GINConv(emb_dim).jittable())
```

- The propagate function, if used, need its parameters to be explicitly annotated using one of the two available options: through the definition of a dictionary or through a comment.

```
propagate_type = {'x': Tensor, 'edge_attr': Tensor}

# propagate_type: (x: Tensor, edge_attr: Tensor)
```

- It can happen that TorchScript is not able to recognize the correct type of variables. In this case, it is necessary to use an assertion to explicitly declare that the variable is an instance of the correct type.

```
assert isinstance(edge_embedding, Tensor)
```

- A common approach to speed up the training and inference steps is to use batched data. Unfortunately, TorchScript does not support forward functions that take as

input a batch. For this reason, the forward function must receive Tensors as input, thus the batch must be split into its component.

```
def forward(self, batched_data):
```

```
def forward(self, x, edge_index, edge_attr):
```

- The parameters of the forward function must be explicitly annotated with their type. If not declared, it is assumed to be of type Tensor.

```
def forward(self, x: Tensor, edge_index: Tensor,
            edge_attr: Tensor) -> Tensor:
```

- TorchScript always expects an integer literal for the index, this is because indexing is only supported with integer literals. For this reason, cycles that do not use integer literals must be changed into enumeration.

```
for idx, layer in enumerate(self.convs):
```

5.2.2 Torch-MLIR Compilation

Once having designed, implemented, made compatible with TorchScript, and trained the GNN model in PyTorch, it is possible to use the torch_mlir.compile API to obtain the MLIR representation of the model. In particular, this API takes three parameters as input: the GNN model, an example input of the model and the desired output type. The Graph Neural Network model must have been already trained, and frozen, ready for inference. The second parameter, the example input of the model, is an arbitrary input similar to the one that would be given for inference purposes. It is required because, by default, the implicit Jit function called by Torch-MLIR to script the model and obtain a script module involves compiling the forward method and recursively compiling any methods, submodules, and functions called within the forward method. This results in a JIT IR which is converted to the torch dialect, which is almost in a 1:1 correspondence. The torch dialect is then lowered into one of the three available output dialects: linalg, tosa, mhlo. The purpose of the last parameter of torch_mlir.compile is to choose which of these three dialects has to be used for the output MLIR.

An additional parameter that can be used is related to the tracing. There are two ways in which it is possible to obtain a TorchScript representation: torch.jit.script and torch.jit.trace. The compile API of Torch-MLIR uses the first one by default. Instead, if the option use tracing is set to True, JIT tracing is used. The behavior of the two functions is slightly different.

Tracing traces the execution of the code for the given inputs to generate a TorchScript representation. It only captures functions and modules that lack untracked external dependencies (e.g., perform input/output or access global variables). Tracing exclusively captures operations performed when the specified function is executed with the provided tensors. Consequently, the resulting ScriptModule will consistently execute the same traced graph for any input.

In conclusion, tracing can be a valid option in some cases, such as when there is no need to record any control-flow like if-statements or loops, but the scripting is preferred, and it is guaranteed to work in a more wide set of cases. A call example of the compile Python API of Torch-MLIR is reported below.

```
module = torch_mlir.compile(gnn_model, (x, features, adj),
                            output_type="linalg-on-tensors")
```

Once having obtained the compiled module, the expected behavior is to use one of the backends provided by torch-MLIR to run inference. This is not the flow followed in this thesis, because, as represented in the accelerator design flow in Figure 5.1, it is time to export the Linalg representation for the next phase. This can be done by simply saving the model to an MLIR file, as shown below.

```
with open("gnn_model.mlir", "w", encoding="utf-8") as outf:
    outf.write(str(module))
```

Only the GCN model implemented in PyTorch reached this phase of the toolchain. During the research, much effort has been spent in trying to add support for the PyTorch Geometric framework to the toolchain. Even if some improvements have been achieved in this regard, such as the implementation of support of the constant of Tuple type, there are still open points to work on. For this reason, at the actual state, the proposed design flow only supports PyTorch as a high-level framework

In particular, Torch-MLIR does not support the aten.scatter_add operation, which, at the actual state, cannot be lowered to MLIR. This operation is extensively used by PyTorch Geometric, leading to the incompatibility of the two tools.

Another current limitation of the framework derives from the fact that Torch-MLIR does not support the sparse tensor type. Each sparse tensor implemented in PyTorch, with the relative sparse operations, is lowered to MLIR to a dense tensor, losing all its representation's advantages. A promising option to avoid this is Taco [29] with its PyTaco APIs for Python. MLIR-PyTACO [5] is an end-to-end use case for the sparse tensor compiler, which employs a Python interface to process the PyTACO language, creating MLIR

linalg . generic operations annotated with sparse tensor information to depict tensor computations. It subsequently invokes the MLIR sparse tensor code generator to produce an executable program, which is executed by the MLIR execution engine, returning the result back to the Python layer. It is important to clarify that, as introduced in [4], sparse tensors are supported by MLIR, with a dedicated sparse tensor dialect that uses intuitive annotations for different sparse tensor representations, such as CSR and COO. What is not supported yet is the lowering through Torch-MLIR. However, even if MLIR-PyTACO is still premature and in testing phase, interesting features could be brought by its advancement. In fact, part of this work successfully exported the MLIR of a sparse matrix multiplication generated by MLIR-PyTACO. It has been then used as input for the sparse compiler to obtain the LLVM representation, which has been synthesized using PandA-Bambu. However, actually PandA-Bambu does not support the struct data type as the return type of a function and, for this reason, no simulation of such operation had been performed.

Even without the sparse tensor representation, using the proper optimizations provided by the toolchain in combination with the higher computational performance of FPGAs, still makes it possible to accelerate the GNN operations, as will be stated in the next Chapter.

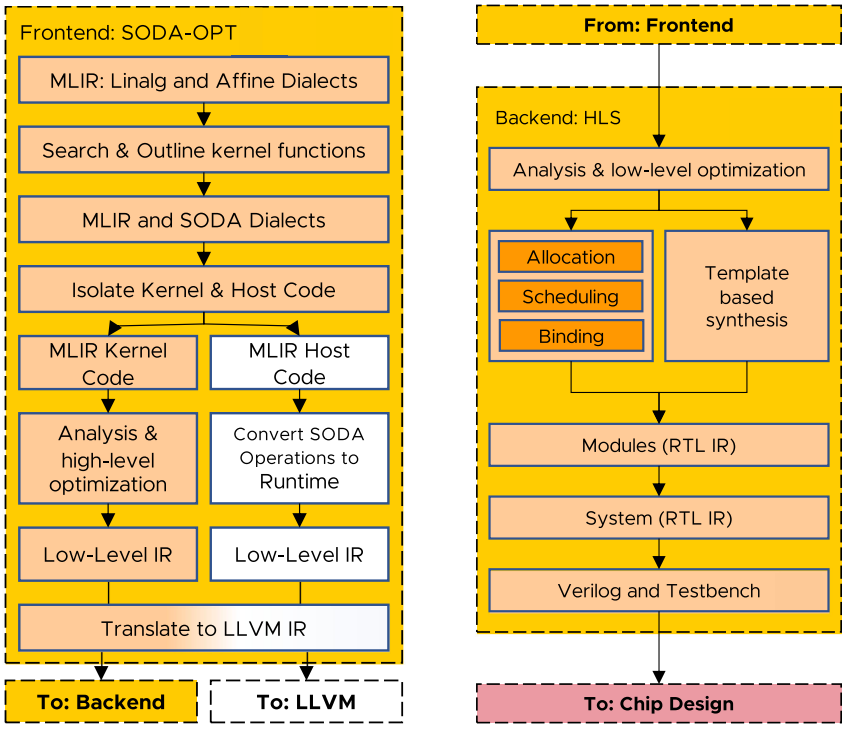
5.3 Synthesizer

The synthesizer represents the final step of the toolchain, which optimizes and synthesizes the MLIR representation, targeting FPGA. This step includes SODA-OPT and PandA-Bambu, both introduced in Section 2.6. The following Subsections provide insight into what is happening internally to these two components.

5.3.1 SODA-OPT

SODA-OPT, as shown in Figure 5.3a, receives as input the MLIR representation of the model. This step is primarily responsible for applying optimizations that can be exploited in the next step. In particular, a subset of MLIR passes can be used to do so. The output of SODA-OPT is an LLVM representation that serves as input to PandA-Bambu HLS.

Despite the remarkable capabilities of SODA-OPT, it should be noted that it does not support the entire set of dialects utilized within the MLIR ecosystem, such as the ml_program or the Tensor dialect. Consequently, an additional step is required wherein the representation obtained from Torch-MLIR is lowered to remove unsupported dialects. In particular,



(a) Compiler frontend (b) High-level synthesis backend

Figure 5.3: Synthesizer: SODA-OPT and PandA-Bambu overview [2]

once having successfully exported the MLIR representation using the torch_mlir.compile API, a set of mlir-opt passes have been used to remove the unsupported dialects by lowering them to supported ones, such as memref.

The next step consists in outlining the part of MLIR code to accelerate. In general, the aim is to accelerate the whole function. To do so, it is enough to modify the MLIR file by adding soda.launch and soda.terminator flags at the beginning and end of the function, thus after the start of the forward method and before the return statement.

SODA-OPT provides various passes that can be used to apply optimization to the outlined code. In particular, it provides a subset of MLIR passes and a set of custom passes. The SODA-OPT passes are continuously evolving, trying to keep up with rapid advancement and innovation within the domain of MLIR.

An essential part of this research has been the analysis of the GNN model, the understanding of its bottlenecks, and the consequent identification of the optimization having the biggest impact on performance, without increasing too much the area of the accelerator.

The first part of this analysis has been conducted on PyTorch. Profiling the inference function of the GCN model, the result showed that, in general, nearly 60% of the time

```
affine.for %arg3 = 0 to 15 {
  affine.for %arg4 = 0 to 16 {
    affine.for %arg5 = 0 to 15 {
    %0 = affine.load %arg0[%arg3, %arg5] : memref<15x15xf32>
    %1 = affine.load %arg1[%arg5, %arg4] : memref<15x16xf32>
    %2 = affine.load %arg2[%arg3, %arg4] : memref<15x16xf32>
    %3 = arith.mulf %0, %1 : f32
    %4 = arith.addf %2, %3 : f32
    affine.store %4, %arg2[%arg3, %arg4] : memref<15x16xf32>
    }
  }
}
```

Listing 5.3: Matrix multiplication in MLIR affine dialect

required to make a prediction was used for the matrix multiplication operation as will be shown in Chapter 6. For this reason, an important part of this thesis is represented by research on how to optimize matrix multiplication using SODA-OPT passes to then accelerate the GCN inference.

A matmul operation in the linalg dialect is lowered to affine by SODA-OPT, resulting in three nested affine loops shown in Listing 5.3.

Among all optimizations available in SODA-OPT the biggest improvement in terms of performance is achieved applying the loop unrolling technique, which involves breaking down a loop into multiple smaller iterations, effectively reducing the overhead of loop control instructions and potentially exposing more opportunities for other optimizations. There are two available options: full unrolling and partial unrolling. In the former the entire loop is expanded so that each iteration becomes a separate iteration, resulting in a significant increase in the size of the code. In the latter, instead, only a subset of the loop iterations is expanded into multiple iterations. A parameter, called *unroll factor*, can be set to decide the number of loop iterations that are combined into a single iteration. Listing 5.4 shows the effect of a partial unrolling, with unrolling factor equal to 5, to the matrix multiplication introduced in Listing 5.3.

The loop unrolling technique perfectly allows to exploit the extreme parallelism available on FPGAs, because the loop bodies are expanded allowing the parallel execution of multiple iterations of the original loop. The right choice is not to continuously unroll until having no more loops in the code. The solution is to pick the right trade-off between performance reduction and the area of the matrix multiplication accelerator.

```
affine.for %arg3 = 0 to 15 {
  affine.for %arg4 = 0 to 16 {
    affine.for %arg5 = 0 to 15 step 5 {
      %0 = affine.load %arg0[%arg3, %arg5] : memref<15x15xf32>
      %1 = affine.load %arg1[%arg5, %arg4] : memref<15x16xf32>
      %2 = affine.load %arg2[%arg3, %arg4] : memref<15x16xf32>
      %3 = arith.mulf %0, %1 : f32
      %4 = arith.addf %2, %3 : f32
      affine.store %4, %arg2[%arg3, %arg4] : memref<15x16xf32>
      %5 = affine.apply affine_map<(d0) -> (d0 + 1)>(%arg5)
      %6 = affine.load %arg0[%arg3, %5] : memref<15x15xf32>
      %7 = affine.load %arg1[%5, %arg4] : memref<15x16xf32>
      %8 = affine.load %arg2[%arg3, %arg4] : memref<15x16xf32>
      %9 = arith.mulf %6, %7 : f32
      %10 = arith.addf %8, %9 : f32
      affine.store %10, %arg2[%arg3, %arg4] : memref<15x16xf32>
      %11 = affine.apply affine_map<(d0) -> (d0 + 2)>(%arg5)
      %12 = affine.load %arg0[%arg3, %11] : memref<15x15xf32>
      %13 = affine.load %arg1[%11, %arg4] : memref<15x16xf32>
      %14 = affine.load %arg2[%arg3, %arg4] : memref<15x16xf32>
      %15 = arith.mulf %12, %13 : f32
      %16 = arith.addf %14, %15 : f32
      affine.store %16, %arg2[%arg3, %arg4] : memref<15x16xf32>
      %17 = affine.apply affine_map<(d0) -> (d0 + 3)>(%arg5)
      %18 = affine.load %arg0[%arg3, %17] : memref<15x15xf32>
      %19 = affine.load %arg1[%17, %arg4] : memref<15x16xf32>
      %20 = affine.load %arg2[%arg3, %arg4] : memref<15x16xf32>
      %21 = arith.mulf %18, %19 : f32
      %22 = arith.addf %20, %21 : f32
      affine.store %22, %arg2[%arg3, %arg4] : memref<15x16xf32>
      %23 = affine.apply affine_map<(d0) -> (d0 + 4)>(%arg5)
      %24 = affine.load %arg0[%arg3, %23] : memref<15x15xf32>
      %25 = affine.load %arg1[%23, %arg4] : memref<15x16xf32>
      %26 = affine.load %arg2[%arg3, %arg4] : memref<15x16xf32>
      %27 = arith.mulf %24, %25 : f32
      %28 = arith.addf %26, %27 : f32
      affine.store %28, %arg2[%arg3, %arg4] : memref<15x16xf32>
    }
  }
}
```

Listing 5.4: Unrolled matrix multiplication in affine dialect with unrolling factor 5

SODA-OPT, after having identified the key code regions, having outlined them into separate MLIR modules, and having applied the transformation passes to the MLIR input, optimizes the portion of code selected for hardware acceleration with a progressive lowering through different MLIR dialects. As a final result, the input code is translated into an LLVM intermediate representation intentionally restructured for hardware synthesis.

5.3.2 PandA-Bambu

PandA-Bambu represents the last phase of the synthesis. As represented in Figure 5.3b, it receives the LLVM representation as input, and after having applied some optional low-level optimizations, it performs the typical steps of HLS introduced in Section 2.4.

The LLVM intermediate representation taken as input from PandA-Bambu is received by the Clang compiler frontend which builds an internal IR to perform the HLS steps. After having applied the specified optimizations, the generated design in an HDL is given as output. As a result, after traversing through each stage of the proposed toolchain's process, PandA-Bambu's output represents the final output, an accelerator tailored to target and maximize performance on cutting-edge FPGA architectures.

PandA-Bambu allows the specification of different optimizations and settings that can have a big impact on accelerator performance. Some optimization techniques have been explored, including in particular an evaluation of the impact of varying the number of memory channels.

In particular, an experimental analysis has been performed varying the number of memory channels, between the minimum, 2 channels, and the maximum, 32 channels. The latter option uses an external memory for the accelerator, which allows to better exploit the high level of parallelism that can be achieved using the loop unrolling technique, but at the same time, more cycles and area for loading data are required. It is the case of a trade-off between the number of channels and the required number of data load cycles. The conducted study revealed that there is a point, represented by a specific number of parallel operations, after which using an external memory with 32 channels becomes beneficial.

PandA-Bambu also offers the possibility to apply low-level loop unrolling, but this option has been disabled to be able to evaluate in isolation the performance impact of the SODA-OPT loop unrolling technique and avoid that loops are unrolled automatically if the option was not enabled in SODA-OPT. PandA-Bambu provides also the possibility to export some files that have been used for this research. One important file is represented by the HLS graph, which shows the computation states and transitions, the number of cycles

Algorithm 5.1 Naive matrix multiplication algorithm

```
1: Data: A[R][P], B[M][N]
2: Result: C[R][N]
3: if P == M then
4:    for m = 0; m < R, m++ do
5:       for r = 0; r < N, r++ do
6:          C[m][r] = 0
7:          for k = 0; k < M, k++ do
8:             C[m][r]+ = A[m][k] * B[k][r]
9:          end for
10:      end for
11:   end for
12: end if
```

needed by each operation to complete, and other information that has been useful to study and understand the impact of both SODA-OPT and PandA-Bambu optimization settings.

5.4 Matrix multiplication algorithm

This section introduces some heuristics followed during the matrix multiplication acceleration experiments. They served as a guide to be followed during the preliminary study phase before conducting experiments. This allowed for a better understanding of which optimizations might be most appropriate and would yield the most significant positive impact on performance.

The naive implementation of the row-by-column multiplication, whose pseudocode is shown below, is characterized by three nested loops. The multiplication is possible only in the case the number of column of the first matrix is equal to the number of row of the second one.

Figure 5.4 highlight how an element of the output matrix is computed, using a row of the first matrix and a column of the second one. In a general matrix multiplication, shown in Equation 5.1, each element of the new matrix can be computed accordingly to Equation 5.2.

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix} \tag{5.1}$$

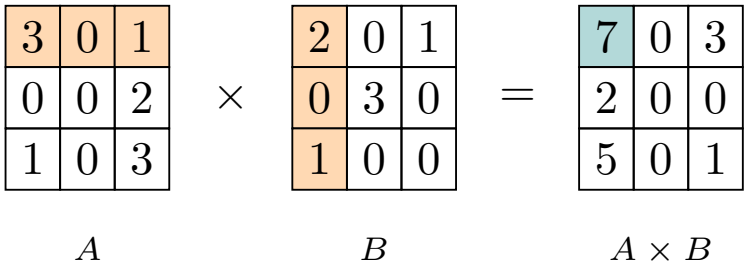


Figure 5.4: Row-by-column matrix multiplication example

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj} \tag{5.2}$$

Then, the number of cycles needed by an accelerator generated by PandA-Bambu without optimizations can be calculated using Equation 5.3. The notation used refers to Equation 5.1, but it can be referred to Algorithm 5.1 by considering $n = M \wedge m = R \wedge p = N$ The first part of the equation computes the total number of iterations, while the second half computes the number of cycles needed to perform multiplications and additions, and to load and store data. The number of cycles needed to load and store data are computed by adding one cycle for each *ch* operands read, and one cycle for each *ch* operand written, with *ch* equal to the number of memory channels. From Listing 5.3, it can be seen that there are three load and one store, for a total of four memory operations.

$$cycles = (n \cdot m \cdot p) \cdot \left(cycles_{mul} + cycles_{add} + \frac{4}{ch} \right) \tag{5.3}$$

Let us consider two matrices, the first of size 15×15 and the second of size 15×16. The FPGA model used for the experimental phase uses three cycles for floating-point addition and two cycles for floating-point multiplication, both single precision. By applying the Equation 5.3, the expected number of cycles for computing such matrix multiplication, using two memory channels is 25,200.

5.4.1 Emulating global memory coalescing of CUDA

The work in [8] iteratively optimize an implementation of matrix multiplication written in CUDA, and it is a useful resource to understand which techniques can be applied also to FPGAs. One of the most interesting optimizations that this work tried to apply to FPGA architecture is global memory coalescing, a memory optimization technique employed in parallel computing, specifically within the context of GPU programming and

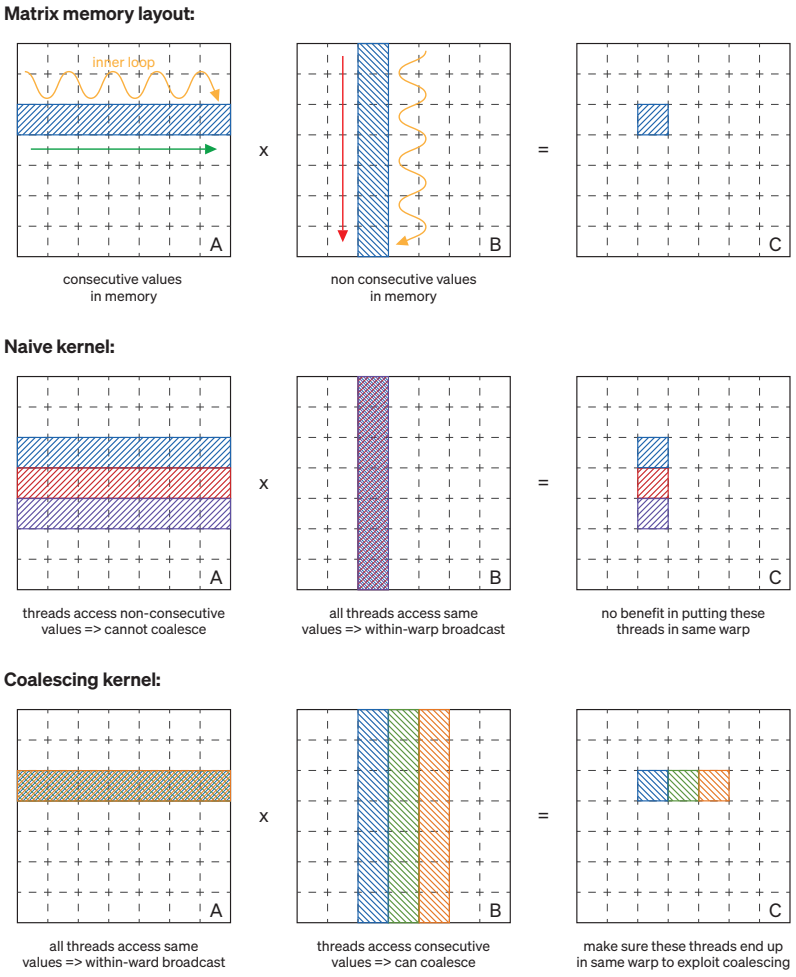


Figure 5.5: Matrix multiplication memory coalescing [8]

CUDA. Its objective is to enhance the efficiency of memory access patterns and data transfers when multiple threads or processing units concurrently access global memory. This is accomplished through the reduction of memory access latency and bandwidth consumption, organizing memory accesses in a manner that optimizes the utilization of the memory hierarchy and the bus bandwidth of GPUs and similar parallel processors. During execution, the threads within a block are organized into entities known as warps, which are subsequently allocated to warp schedulers, i.e. the physical cores responsible for executing instructions. Each multiprocessor houses four warp schedulers and each warp comprises 32 threads. The arrangement into warps is determined by a value called threadId in a consecutive sequence, ensuring that threads with neighboring threadId values are grouped together within the same warp.

The author of this work enables coalescing changing how positions of the result matrix C are assigned to threads. This change in the global memory access pattern is illustrated in Figure 5.5.

To reproduce the behaviour of the coalescing kernel shown in Figure 5.5, multiple memory channels has been used. Using burst read and write operations allows for transferring multiple data elements in a single memory access, enhancing memory coalescing. This optimization enriched the structure of the accelerator, by loading and storing multiple data in parallel and thus using more PEs in parallel to perform operations such as multiplications and additions.

5.5 Conclusion

This chapter presented the main contribution of this thesis, an FPGA toolchain to create GNN hardware accelerators starting directly from PyTorch high-level framework, with the possibility to use different optimizations to improve performance depending on application bottlenecks.

The most crucial advantage of the proposed toolchain is that it allows obtaining an accelerator without any knowledge of hardware design and implementation.

Moreover, analyzing two representative GNN models it has been highlighted that most of the computation is composed of matrix multiplication operations. Available opt passes in SODA-OPT and PandA-Bambu can be tailored for matmul, their effect is shown through experiments in the next chapter.

6 Experimental Procedures and Results

This chapter presents all the experiments performed about matrix multiplication acceleration and GNN acceleration along with the achieved results.

All the CPU experiments have been conducted using an Intel Core i9, with 8 cores and a frequency of 2,3 GHz. On the other hand, the synthesis experiments targeted an AMD Virtex UltraScale+ (Alveo U280) FPGA. The whole experimental phase evaluated only the computational time of the accelerators, and results are based on the assumption that all data are loaded from BRAMs, requiring two cycles to read data and one cycle to store data.

6.1 Model analysis and profiling

As already anticipated in Section 5.1, the GCN model is implemented in PyTorch, and it is characterized by two convolutional layers, a ReLU and a dropout functions. The forward function of each layer contains two matrix multiplications, and one of the two is a sparse multiplication.

The first step to understand how to accelerate the PyTorch GCN model was to analyze and profile it. Table 6.1 shows the results extracted from a run of the PyTorch profiler, which

Name	**Self CPU %**	**Self CPU**	**CPU total %**	**CPU total**
aten::mm	50,25%	1,012ms	89,72%	1,807ms
aten::addmm	36,30%	731,0μs	37,04%	746,0μs
aten::add	4,67%	94,0μs	4,67%	94,0μs
aten::mm matrix multiplication; aten::add matrix sum; aten::addmm matrix multiplication plus matrix sum.				

Table 6.1: Excerpt of GCN model inference profiling result

measures the ATen operations used at a lower level of PyTorch. In particular, ATen [35] is a tensor library that serves as the foundation for most Python and C++ interfaces within PyTorch. It provides a central Tensor class that encompasses a multitude of operations, with the Tensor class dynamically selecting the appropriate one based on its type.

Among the profiled operations, aten::mm performs a multiplication between two tensors, aten::addmm performs a multiplication between two tensors and then another tensor is added to the result and finally aten::add returns the sum of two tensors provided as input. The distinction between *self CPU time* and *total CPU time* lies in the fact that self CPU time does not contain the time spent in child operator calls, whereas total CPU time contains it, considering that operators can invoke other operators. It is clear that the bottleneck and the most time-consuming operation is the matrix multiplication. In particular, more than 50% of the self CPU time is used by matrix multiplication, while, considering the child operator calls, this percentage represents nearly the 90%. This result clearly justifies a special focus on matrix multiplication acceleration.

6.2 Matrix multiplication acceleration

Matrix multiplication, introduced in Section 5.4, is a well-known algorithm. It consists of multiplying two compatible matrices to obtain the result matrix. A lot of work has been done to try to improve its performance on different architectures [6, 8]. The following subsections explains how this thesis improved matmul performance exploiting the proposed toolchain.

6.2.1 PyTorch matrix multiplication benchmark

PyTorch provides different matrix representations and different matrix multiplication functions. The one considered in this Subsection are *torch.mm* and *torch.spmm*. The former function multiplies two dense matrices, but it also supports COO and CSR representation. The latter, instead, is typically used for sparse matrix multiplications, in which one of the two matrices, or both, are saved using sparse representations.

Table 6.2 represents a benchmark for the dense matrix multiplication between a first matrix of size 15×15 and a second matrix of size 15×16, both composed by float32 elements. The chosen sizes for input matrices are not arbitrary; they have been specifically selected due to their extensive use in the following experiments, requiring an accurate benchmark for comparison purposes. In particular, the table shows five different runs, each composed by a different number of executions. Then, the final time is computed as

Executions	Run1	Run2	Run3	Run4	Run5	Avg. time
2E06	1.587E-06	1.568E-06	1.567E-06	1.594E-06	1.572E-06	1.578E-06
4E06	1.598E-06	1.585E-06	1.592E-06	1.602E-06	1.599E-06	1.595E-06
6E06	1.601E-06	1.614E-06	1.603E-06	1.603E-06	1.608E-06	1.606E-06
8E06	1.608E-06	1.607E-06	1.600E-06	1.614E-06	1.617E-06	1.609E-06
10E06	1.614E-06	1.603E-06	1.603E-06	1.613E-06	1.607E-06	1.608E-06

Table 6.2: Benchmark of *torch.mm* PyTorch function. Five runs using different number of executions and their average, unit of measure is second.

Function	Sizes	Sparsity	Runs	Dense×Dense	COO×Dense	COO×COO	CSR×Dense	CSR×CSR
torch.mm	15×15, 15×16	0.9	10E06	**1.599E-06**	2.844E-06	1.658E-05	1.058E-05	8.632E-05
torch.spmm	15×15, 15×16	0.9	10E06	1.934E-06	3.385E-06	1.750E-05	1.183E-05	1.746E-05
torch.mm	150×150, 150×16	0.9	1E06	**5.220E-06**	4.303E-05	1.058E-04	1.384E-05	1.393E-04
torch.spmm	150×150, 150×16	0.9	1E06	6.593E-06	4.659E-05	1.143E-04	1.461E-05	1.090E-04
torch.mm	150×150, 150×16	0.99	1E06	5.752E-06	7.039E-06	1.887E-05	1.256E-05	8.730E-05
torch.spmm	150×150, 150×16	0.99	1E06	**4.678E-06**	7.797E-06	1.883E-05	1.214E-05	1.847E-05
torch.mm	2708×2708, 2708×16	0.999	100E03	1.288E-03	2.030E-04	1.325E-04	**3.377E-05**	1.073E-04
torch.spmm	2708×2708, 2708×16	0.999	100E03	1.255E-03	2.012E-04	1.316E-04	3.661E-05	1.186E-04

Table 6.3: Comparison between dense and sparse PyTorch matmul functions, computed as average of multiple runs. Times unit of measure is seconds.

the average of the five different runs. Additionally, this average has been computed five times using increasing number of executions. The results shows that the variance of the runs of the last experiments is lower than the others, fixing to ten millions a sufficient number of executions for good accuracy and stability.

As expected, given the globally high amount of executions, the five average execution times are similar between them, and the variance decreases as the number of executions increases. In conclusion, the time needed by a dense matrix multiplication between two matrices of the given size can be considered equal to $1.608\mu s$.

Since the GCN model uses both dense and sparse matrix multiplication functions, Table 6.3 shows the times needed by both functions according to different representations of the two input matrices A and B. Different experiments have been performed using randomly generated input matrices; they are composed by float32 elements and different input sized and levels of sparsity have been analyzed.

It is clear that sparse matrix multiplication, both with dense and sparse matrix repre-

sentations, does not increase performance on CPU architecture when the size of input matrices is small. The disadvantage of using sparse matrix multiplication on CPU becomes more evident as the size of the input matrices increases but the level of sparsity remains constant. However, when the size of input matrices increases considerably and the level of sparsity is very high, using sparse representations, in particular CSR, can bring a significant advantage.

Even if the following experiments do not take advantage of sparse matrix computations, the utilization of custom optimizations for the analyzed GCN has enabled the acceleration of inference, balancing the absence of sparse matrix computations.

6.2.2 FPGA-accelerated matrix multiplication

Before applying any optimization, it is necessary to understand the difference between PyTorch matrix multiplication operation and the baseline accelerator. To obtain the matmul accelerator, the process began with following the initial steps of the toolchain using the GCN model. After having made the model compatible with TorchScript, its linalg representation has been obtained through Torch-MLIR compilation. Subsequently, only the matmul code of the generated MLIR representation has been outlined through the use of SODA annotation. Following this, the LLVM-generated file has been synthesized with PandA-Bambu using baseline attributes, including the −fno−unroll−loops flag, two memory channels, and the ALL_BRAM option. Figure 6.1 shows the result of this analysis, which reveal the fact that the cycle counts required by the accelerator for each experiment align perfectly with the anticipated expectations outlined in Section 5.4. Specifically, dividing the cycle count of the first experiment of the table and the final one by the total count of loop iterations effectively confirms a consistent result of approximately 7 in both instances, confirming that the number of cycles per each loop iteration stayed constant, as shown by the second part of Equation 5.3. However, this constant behavior limits the utilization of the possibilities presented by FPGA technology, which encompass substantial parallelization potential and concurrent memory accesses.

The PyTorch times have been recorded by averaging five measurements each of ten millions executions. The baseline accelerator is much faster than PyTorch when matrices are relatively small. The reason behind this behavior is that the generated accelerator simplifies data loading by assuming that all data is available in BRAM. By default, the system uses two channels and stores all memory objects in BRAMs. However, an alternative approach, explored in this thesis, involves employing a greater number of memory channels while utilizing external memory. The advantage of the accelerator in terms of

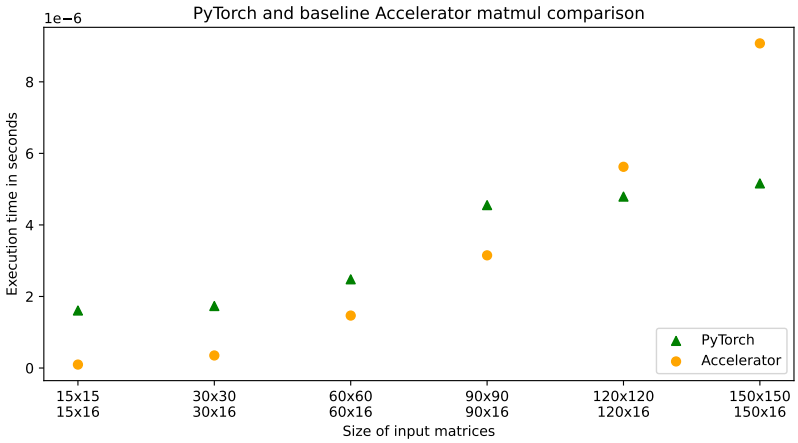


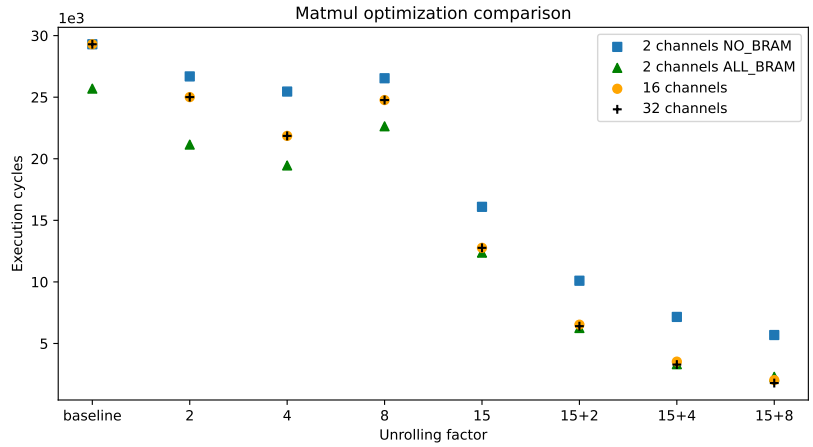
Figure 6.1: Performance comparison between PyTorch matmul function and accelerator

performance decreases as the size of the input matrices increase, until reaching a point in which the accelerator becomes slower than PyTorch solution.

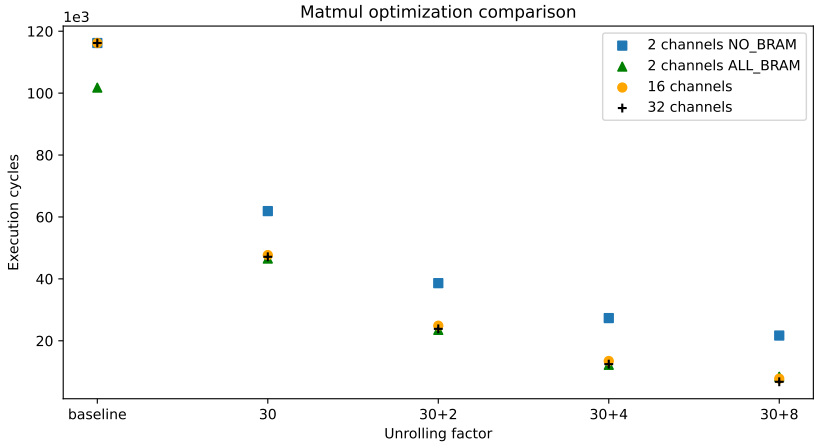
This behaviour can be attributed to the fact that PyTorch times have been recorded using all the eight available threads on the machine. So, it is expected that the `torch.mm` function exploits more parallelism with respect to the accelerator, and this advantage is more evident when the potential level of parallelism increases. For this reason, the optimizations discussed in Section 5.3.1 and in Section 5.3.2 need to be applied to exploit more parallelism.

SODA-OPT, as introduced in Subsection 5.3.1, offers the possibility to make different types of unrolling, among which the full unroll which completely unroll the innermost loop, and the partial unroll up to an arbitrary factor. Moreover, PandA-Bambu introduces several optimization options, one of which, as already anticipated above, involves expanding the number of memory channels in use by exploiting an external memory instead of a BRAM.

The combination of loop unrolling and multiple memory channels offers both advantages and disadvantages, resulting in a trade-off. Loop unrolling enhances parallelization, thereby reducing computational time. However, especially when utilizing more than two memory channels, it increases the number of parallel processing elements, leading to a larger area footprint. Additionally, external memory allows for up to 32 memory channels, enabling the simultaneous loading of 32 variables. Nonetheless, accessing data from external memory requires more load cycles compared to accessing internal memory, depending by the access time of DDRs. For the performed experiments both internal and external memory requires two cycles to load data and one cycle to store data, but not



(a) Input matrices 15×15, 15×16



(b) Input matrices 30×30, 30×16

Figure 6.2: Matrix multiplication optimization comparison

using BRAM changes the structure of the accelerator, resulting in a rise of the nnumber of cycles.

Figure 6.2 show the performance of two matrix multiplication accelerators operating on different sizes of the input matrices. Figure 6.2a show results of a multiplication between two matrices of size 15×15 and 15×16, while Figure 6.2b shows results of a multiplication between two matrices of size 30×30 and 30×15. In both figures, on the x-axis there is the unrolling factor used and on the y-axis the resulting number of cycles. The baseline computation does not use unrolling, instead when there are two factors it means that the first one can be considered as a full unroll of the innermost cycle, while the second one is the unrolling factor applied to the original second innermost cycle.

In both cases, when the parallelization is not high, thus the unrolling factor is small,

using two channels and storing all memory objects in BRAMs is the best option since a minor number of cycles is required to perform the computation. Using 16 channels seems to be similar to use 32 channels, but when parallelization is high the latter option uses fewer cycles. As expected, the findings indicate that opting for two channels with external memory is the least favorable choice. This is primarily because employing external memory becomes advantageous when a large volume of data can be simultaneously loaded and stored to offset the additional cycles required for external memory access.

The best trade-off is achieved using an unrolling factor for which the number of cycles needed by the accelerator with 32 memory channels is lower than the number of cycles needed by the accelerator with 2 memory channels. In the first case, in Figure 6.2a, this objective is achieved with two unrolling factors of 15 and 8, while in the second case is achieved with unrolling factors of 30 and 8.

These results can be used to set a heuristic useful to identify the number of parallel unrolled loop iterations that makes the use of 32 channels with external memory preferable to the use of 2 channels with BRAM:

$$i \cdot j \cdot k \geq 2 \cdot \sqrt{M \cdot N \cdot R} \tag{6.1}$$

where M, N and R are the sizes of the three nested loops, as defined in Algorithm 5.1, and i, j and k are their respective loop unrolling factors, following the rule $j \neq 1 \iff i = M \wedge k \neq 1 \iff j = N$.

Equation 6.1 captures the relations of the two results. In fact, using 15 and 8 as unrolling factors means having $15 \cdot 8 = 120$ parallel loop iterations. Instead, using 30 and 8 as unrolling factors means having $30 \cdot 8 = 240$ parallel loop iterations. Additionally, the total amount of possible parallel loop iterations in a matrix multiplication between two matrices of sizes 15×15 and 15×16 is equal to $15 \cdot 16 \cdot 15 = 3,600$. Meanwhile, the total amount of possible parallel loop iterations in a matrix multiplication between two matrices of sizes 30×30 and 30×16 is $30 \cdot 16 \cdot 30 = 14,400$. Even if the two number of parallel loop iterations representing the changing point of the trade-off, 120 and 240, are different, they are related by Equation 6.1.

Equation 6.1 should not be taken as an infallible rule, it is the outcome of this experimental phase and should be used as a discriminant to decide when to use thirty-two memory channels instead of two. In conclusion, the generalized rule, outcome of this comparative analysis, conducted using different sizes of input matrices, is that the number of parallel unrolled loop iterations that justify the use of 32 channels is given by Equation 6.1.

Name	Nodes	Words	Links	Task	Classes
Cora	2708	1433	5429	Multiclass classification	7
Cora15	15	15	3	Multiclass classification	7
Cora30	30	30	4	Multiclass classification	7
Cora60	60	60	8	Multiclass classification	7
Cora90	90	90	18	Multiclass classification	7
Cora120	120	120	22	Multiclass classification	7
Cora150	150	150	37	Multiclass classification	7

Table 6.4: Cora sub-dataset used for GCN infernce

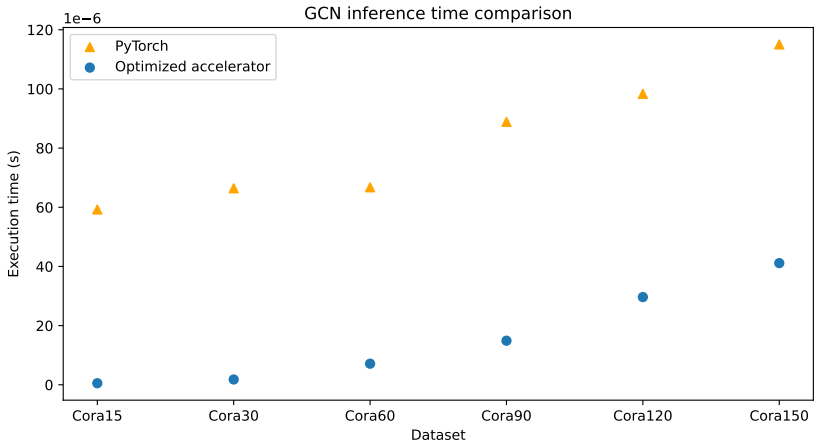


Figure 6.3: GCN inference PyTorch-Accelerator comparison

6.3 GCN accelerator evaluation

In this Subsection, the toolchain and the proposed optimizations are applied to the GCN model.

Figure 6.3 shows the result of a comparative analysis between the PyTorch CPU time and the optimized FPGA accelerator time to perform GCN inference. All the experiments have been performed using a subset of the Cora dataset introduced in Subsection 5.1.1, as detailed in Table 6.4. The PyTorch times have been acquired averaging one million of execution time measurements using the PyTorch built-in benchmark API.

Both accelerators have been obtained from the same PyTorch model, which has been lowered using Torch-MLIR. After obtaining the MLIR representation and outlining the

Dataset	Optimizations	Cycles	Slices	Luts	Registers	DSPs	BRAMs	Frequency(MHz)	SpeedUp
Cora15	Baseline	115,852	8,307	32,927	32,195	16	256	147.77	-
Cora15	Full Unroll	93,705	8,338	36,265	35,164	44	256	179.08	1.23
Cora30	Baseline	385,874	7,457	30,379	26,925	16	256	158.25	-
Cora30	Full Unroll	301,800	12,448	50,907	50,558	74	256	168.12	1.27
Cora60	Baseline	1,402,860	6,928	30,115	24,287	16	256	158.07	-
Cora60	Full Unroll	1,064,580	21,726	94,025	77,956	134	256	149.20	1.31
Cora90	Baseline	3,051,630	6,769	29,899	25,966	16	256	166.69	-
Cora90	Full Unroll	2,298,510	32,046	154,917	108,770	194	256	154.20	1.32
Cora120	Baseline	5,332,200	8,045	30,441	25,878	16	256	145.79	-
Cora120	Full Unroll	3,987,840	43,187	218,178	135,961	254	256	134.49	1.33
Cora150	Baseline	8,244,570	7,499	29,390	26,015	16	264	163.61	-
Cora150	Full Unroll	6,136,470	27,002	115,463	143,916	26	264	149.20	1.34

Table 6.5: GCN inference time comparison

entire forward function, the model has been synthesized using SODA-OPT and PandA-Bambu, employing both baseline and optimized configurations. Figure 6.4 and Table 6.5 show how the accelerator has been affected by the unrolling technique with respect to the baseline performance. The optimized settings uses two channels and one full unrolling of the innermost loop, and it obviously requires more area than the baseline accelerator, but it is computationally faster; being matrix multiplication the most time-consuming operation of the GCN, thanks to loop unrolling technique, the speedup increases as the size of the dataset, and thus the number of nodes in the graph, increases. The used optimized setting with two channels and BRAM has been preferred to the alternative considered setting with thirty-two channels and two external memory, because it already achieves high speedup without increasing too much area requirements, representing the best trade-off for the considered experiment.

The results of this final evaluation are incredibly encouraging, showing significant improvement obtained by the optimized accelerator using two channels with on-chip BRAMs. The speedup of the accelerator is still being affected by the size of the input matrices, but this was an expected result since the unrolling factor used is contained. The accelerator's computational time is significantly lower with respect to the one measured on CPU with PyTorch, and for bigger dataset sizes the technique presented in Subsection 6.2.2 can be used, accelerating even more the computational performance of the accelerator.

The area of the accelerator, instead, is obviously affected by the sizes of the input matrices and of the unrolling factor. This because more the matrices are big, more the parallel loop iterations will be and thus the parallel processing elements.

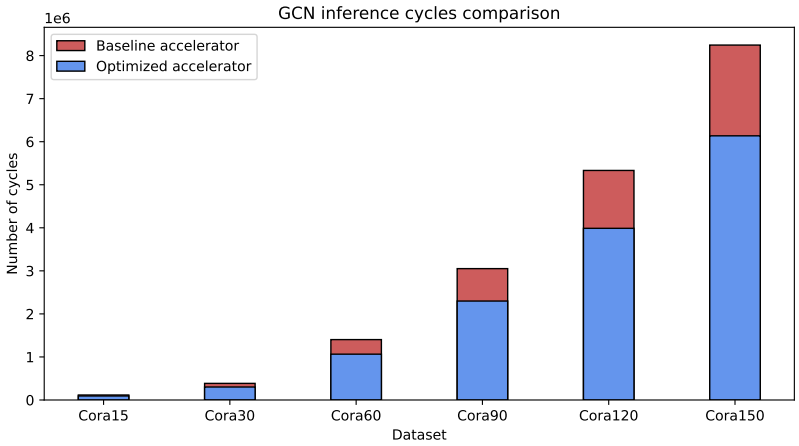


Figure 6.4: GCN inference number of cycles comparison

However, the possibilities offered by the proposed toolchain are various. It is possible to use less memory channels with a lower loop unrolling factor to still have a big positive impact on performance, but containing the accelerator area requirements. The Equation 6.1 can be adapted also to this specific case, by considering the number of load and store and the size of the different loop iterations, becoming a valid indicator to decide which optimization to apply according to the dataset size.

6.3.1 Model accuracy

An evaluation of the model accuracy has been conducted for both the PyTorch-based implementation and the FPGA-accelerated version to ensure that there has been no impact on accuracy.

This comparative analysis has been made possible through the possibility offered by PandA-Bambu which allows for the generation of custom simulation calls using C files. Specifically, the process involved training the model in PyTorch, conducting inference to generate outputs, and saving all inputs and inference results in a binary file. The binary file was then utilized to load all the data within a C file. The forward kernel function was employed to execute inference on the FPGA accelerator using the same inputs as the PyTorch version. The resulting inference output was saved and subjected to analysis, allowing for a comprehensive comparison with the PyTorch implementation.

The results presented in Table 6.6 indicate that the model accuracy has been maintained in the FPGA accelerator. Nonetheless, the slight variation in the floating-point data precision between the PyTorch inference output and the FPGA accelerator may potentially result in a difference of the accuracy, especially when dealing with very large datasets.

Dataset	PyTorch accuracy	Accelerator accuracy	Floating distance
Cora15	0.000	0.000	0.000
Cora30	0.000	0.000	0.000
Cora60	0.000	0.000	0.000
Cora90	0.000	0.000	0.000
Cora120	0.000	0.000	0.000
Cora150	0.000	0.000	0.000

Table 6.6: Model accuracy comparison between PyTorch and FPGA accelerator

6.4 State of the art comparison

The proposed design flow offers the flexibility of utilizing customized settings tailored to diverse applications, which presents a distinct advantage in contrast to the optimized pipeline offered by SODA-OPT [2]. The latter confines users to employ only the full unrolling optimization technique, which can represent a limitation when dealing with extensive loop iterations. This limitation prevents the fine-tuning of the balance between area utilization and performance, particularly when handling significant large loops, potentially leading to resource exhaustion and errors. In contrast, the design flow introduced in this thesis provides a more refined approach to customizing the trade-off between area and performance. It eliminates the need for mandatory double full unrolling when a single one is not sufficient, offering the flexibility to employ a partial unrolling factor.

A noteworthy point of comparison involves FlowGNN [33], which stands out among the state-of-the-art technologies introduced in Chapter 3 as the only solution employing the HLS technique, which is also used in this thesis.

The toolchain proposed in this thesis, as well as the dataflow of FlowGNN, offers support for a diverse range of GNN models. However, FlowGNN is limited to just four configurable parallelization parameters and focuses on delivering model-specific components, which limit the customization of the accelerator even further. In contrast, the proposed toolchain offers many optimization passes from both SODA-OPT and PandA-Bambu, allowing users to finely customize and optimize the accelerator's capabilities.

Another significant advantage of the proposed toolchain is its ability to generate the accelerator directly from the PyTorch implementation, eliminating the need for low-level programming. This sets it apart from FlowGNN, which offers pre-built models in C++.

The authors, through a small example, states that the provided models can be slightly changed to be adapted to different features. However, To do this a minimum of experience with C++ is needed, and if the changes are substantial it could be a blocking factor. On the contrary, the proposed toolchain, in case pre-existing models would not be sufficient, only requires knowledge in PyTorch, one of the most used the high-level framework for GNN implementations.

7 Conclusion and Future Developments

This thesis tackled the challenge of accelerating Graph Neural Network inference by leveraging High-Level Synthesis techniques targeting FPGAs.

An HLS-based toolchain was designed to obtain a GNN inference accelerator starting directly from PyTorch. PyTorch stands as one of the foremost high-level frameworks for neural network implementation, extensively recognized and employed within the community, making the proposed toolchain suitable for different applications.

The results of this research significantly contribute to the field of GNN acceleration, introducing a new perspective about how it is possible to obtain hardware accelerators for GNNs even without having any hardware design knowledge. The toolchain offers different possibilities and provides various optimization passes that can be used to fine-tune the accelerator capabilities.

Within the spectrum of optimizations made available by this toolchain, this thesis primarily delved into two key techniques: the loop unrolling technique in SODA-OPT and the parallel memory access of PandA-Bambu with an external memory of thirty-two channels. These optimizations allowed to achieve encouraging and promising results in accelerating the inference of the GCN model analyzed. By studying and understanding the model bottlenecks, it is possible to achieve consistent improvements.

Lastly, this study has also made substantial contribution in enhancing Torch-MLIR. A new feature, the support of the constant of Tuple type, has been added and different areas of improvement have been identified, where more work would be needed to implement functionalities for the complete support of PyTorch Geometric. Before this research, no examples were available on how to use Torch-MLIR with Graph Neural Networks, and the compatibility of PyTorch Geometric and Torch-MLIR was still an unexplored area.

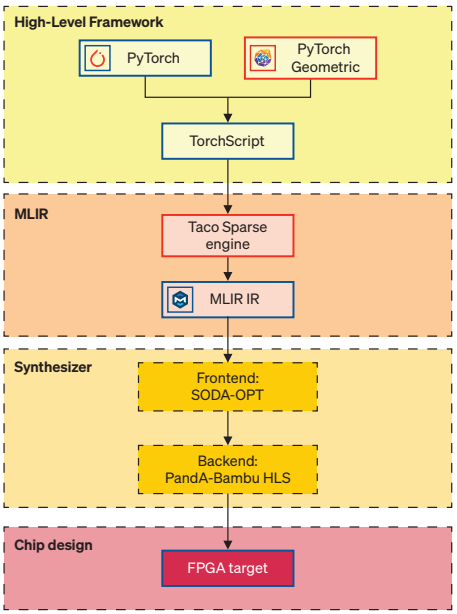


Figure 7.1: Future FPGA Toolchain for Graph Neural Network Acceleration

7.1 Future developments

This thesis represents a significant advancement in the field of GNN acceleration. Nonetheless, there exist opportunities for further enhancement through the implementation of novel features.

Primarily, future works could build upon the innovative contributions of this thesis in terms of enhancing compatibility between PyTorch Geometric and Torch-MLIR. PyTorch Geometric offers valuable features and pre-established classes for implementing Graph Neural Networks. Integrating this advanced framework into the toolchain would mark another significant step forward. This thesis identified the needed area of improvement, representing a possible starting point for these future works.

Another improvement that can be applied to the presented toolchain is the support of sparse tensors. At the actual state, Torch-MLIR is not capable of lowering PyTorch sparse tensors to the Linalg dialect, even if it already supports them. For this reason, some work would be needed on Torch-MLIR to implement this feature, possibly using PyTaco, which would bring a significant advantage also in the resulting accelerators, having the possibility to exploit faster computation with sparse tensors. An additional enhancement that can be integrated into the proposed toolchain involves extending support to sparse tensors. Currently, Torch-MLIR cannot convert PyTorch sparse tensors into the Linalg dialect, even if MLIR [26] already has a way to represent them [4]. Consequently, implementing this feature in Torch-MLIR would require dedicated efforts, producing substantial bene-

fits in the toolchain resultant accelerators. Even if the resulting accelerators, as discussed in Chapter 6, greatly increase performance and reduce the computational time, this enhancement would enable leveraging faster computations through the use of sparse tensors by further improving the performance of the accelerator.

Figure 7.1 shows how the toolchain could appear with the implementation of the above-mentioned future developments. In particular, PyTorch Geometric would be part of the supported high-level frameworks, and the engine used by PyTaco to annotate sparse tensors would be adapted and used to convert the TorchScript representation of the model into MLIR with sparse tensor dialect, ready to be received as input by SODA-OPT.

In conclusion, ongoing support and dedicated efforts are consistently aimed at advancing SODA-OPT [2] and PandA-Bambu [13] through the integration of novel functionalities. The continuous evolution of these tools represents an alternative way of progress over that the classic hardware design techniques, as the introduction of fresh SODA-OPT passes and PandA-Bambu optimizations have the potential to open novel opportunities for refining and optimizing the synthesized hardware accelerators.

Bibliography

[1] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *CoRR*, abs/2010.00130, 2020. URL https://arxiv.org/abs/2010.00130.

[2] N. B. Agostini, S. Curzel, J. J. Zhang, A. Limaye, C. Tan, V. Amatya, M. Minutoli, V. G. Castellana, J. Manzano, D. Brooks, G.-Y. Wei, and A. Tumeo. Bridging python to silicon: The soda toolchain. *IEEE Micro*, 42(5):78–88, 2022. doi: 10.1109/MM.2022.3178580.

[3] A. Auten, M. Tomei, and R. Kumar. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi: 10.1109/DAC18072.2020.9218751.

[4] A. Bik, P. Koanantakool, T. Shpeisman, N. Vasilache, B. Zheng, and F. Kjolstad. Compiler support for sparse tensor computations in MLIR. *ACM Transactions on Architecture and Code Optimization*, 19(4):1–25, sep 2022. doi: 10.1145/3544559. URL https://doi.org/10.1145%2F3544559.

[5] A. B. Bixia Zheng and other Torch-MLIR contributors. Mlir-pytaco: An end-to-end use case for the sparse tensor compiler, 2022. URL https://mlir.llvm.org/OpenMeetings/2022-02-10-PyTACO.pdf.

[6] U. Bondhugula. High performance code generation in MLIR: an early case study with GEMM. *CoRR*, abs/2003.00532, 2020. URL https://arxiv.org/abs/2003.00532.

[7] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond euclidean data. *CoRR*, abs/1611.08097, 2016. URL http://arxiv.org/abs/1611.08097.

[8] S. Böhm. How to optimize a cuda matmul kernel for cublas-like performance: a worklog, 2022. URL https://siboehm.com/articles/22/CUDA-MMM.

[9] H. Chen and C. Hao. Dgnn-booster: A generic fpga accelerator framework for dynamic graph neural network inference, 2023.

[10] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009. doi: 10.1109/MDT.2009.69.

[11] A. Daigavane, B. Ravindran, and G. Aggarwal. Understanding convolutions on graphs. *Distill*, 2021. doi: 10.23915/distill.00032. https://distill.pub/2021/understanding-gnns.

[12] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. *CoRR*, abs/1509.09292, 2015. URL `http://arxiv.org/abs/1509.09292`.

[13] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo. Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330, 2021. doi: 10.1109/DAC18074.2021.9586110.

[14] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019. URL `http://arxiv.org/abs/1903.02428`.

[15] T. Geng, A. Li, T. Wang, C. Wu, Y. Li, A. Tumeo, and M. C. Herbordt. AWB-GCN: hardware acceleration of graph-convolution-network through runtime workload rebalancing. *CoRR*, abs/1908.10834, 2019. URL `http://arxiv.org/abs/1908.10834`.

[16] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017. URL `http://arxiv.org/abs/1704.01212`.

[17] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017. URL `http://arxiv.org/abs/1706.02216`.

[18] L. He. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *CoRR*, abs/1909.00155, 2019. URL `http://arxiv.org/abs/1909.00155`.

[19] 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. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 22118–22133. Curran Associates, Inc.,

2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/fb60d411a5c5b72b2e7d3527cfc84fd0-Paper.pdf.

[20] Y. Hu, Y. Du, E. Ustun, and Z. Zhang. Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021. doi: 10.1109/ICCAD51958.2021.9643582.

[21] J. Kepner, P. Aaltonen, D. A. Bader, A. Buluç, F. Franchetti, J. R. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. E. Moreira, J. D. Owens, C. Yang, M. Zalewski, and T. G. Mattson. Mathematical foundations of the graphblas. *CoRR*, abs/1606.05790, 2016. URL http://arxiv.org/abs/1606.05790.

[22] A. Keramatfar, M. Rafiee, and H. Amirkhani. Graph neural networks: A bibliometrics overview. *Machine Learning with Applications*, 10:100401, 2022. ISSN 2666-8270. doi: https://doi.org/10.1016/j.mlwa.2022.100401. URL https://www.sciencedirect.com/science/article/pii/S2666827022000780.

[23] K. Kiningham, P. Levis, and C. Re. GReTA: Hardware Optimized Graph Processing for GNNs. In *Proceedings of the Workshop on Resource-Constrained Machine Learning (ReCoML 2020)*, March 2020.

[24] K. Kiningham, C. Ré, and P. A. Levis. GRIP: A graph neural network accelerator architecture. *CoRR*, abs/2007.13828, 2020. URL https://arxiv.org/abs/2007.13828.

[25] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL http://arxiv.org/abs/1609.02907.

[26] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.

[27] Q. Li, Z. Han, and X. Wu. Deeper insights into graph convolutional networks for semi-supervised learning. *CoRR*, abs/1801.07606, 2018. URL http://arxiv.org/abs/1801.07606.

[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)*, pages 1–9, 2020.

[29] m. o. t. C. r. g. l. b. P. S. A. a. M. C. Members of Prof. Fredrik Kjolstad's research group at Stanford University and other contributors. Taco: The tensor algebra compiler, 2016. URL `http://tensor-compiler.org`.

[30] NVIDIA. cublas library. URL `https://developer.nvidia.com/cublas`.

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019. URL `http://arxiv.org/abs/1912.01703`.

[32] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. doi: 10.23915/distill.00033. https://distill.pub/2021/gnn-intro.

[33] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao. Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference, 2022.

[34] O. G. B. team and other contributors. Gnn models from open graph benchmark, 2020. URL `https://github.com/snap-stanford/ogb/tree/master/examples/graphproppred/mol`.

[35] P. Team. Aten: a tensor library, 2019. URL `https://pytorch.org/cppdocs/api/namespace_at.html#namespace-at`.

[36] P. Team. Torchscript docs, 2019. URL `https://pytorch.org/docs/stable/jit.html`.

[37] M. W. Thomas N. Kipf and other contributors. Gcn model in pytorch, 2017. URL `https://github.com/tkipf/pygcn`.

[38] n. t. Torch-MLIR team and other contributors. Torch-mlir: Mlir based compiler toolkit for pytorch programs, 2021. URL `https://github.com/llvm/torch-mlir`.

[39] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019. URL `http://arxiv.org/abs/1909.01315`.

[40] B. Weisfeiler and A. Leman. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series*, 2(9):12–16, 1968.

[41] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey

on graph neural networks. *CoRR*, abs/1901.00596, 2019. URL `http://arxiv.org/abs/1901.00596`.

[42] Z. Xianyi and other contributors. Openblas library. URL `http://www.openblas.net`.

[43] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks?, 2019.

[44] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. Hygcn: A GCN accelerator with hybrid architecture. *CoRR*, abs/2001.02514, 2020. URL `http://arxiv.org/abs/2001.02514`.

[45] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. *CoRR*, abs/1806.08804, 2018. URL `http://arxiv.org/abs/1806.08804`.

[46] H. You, T. Geng, Y. Zhang, A. Li, and Y. Lin. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 460–474, 2022. doi: 10.1109/HPCA53966.2022.00041.

[47] B. Zhang, H. Zeng, and V. Prasanna. Hardware acceleration of large scale gcn inference. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 61–68, 2020. doi: 10.1109/ASAP49362.2020.00019.

[48] 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)*, pages 29–39, 2021. doi: 10.1109/FCCM51124.2021.00012.

[49] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018. URL `http://arxiv.org/abs/1812.08434`.

List of Figures

2.1 Example of directed and undirected graphs 5
2.2 Example of a graph and its adjacency matrix 6
2.3 COO and CSR format of Adjacency matrix in Figure 2.2 7
2.4 Number of GNN publications on Google Scholar per year 9
2.5 Example of a Multi-Layer Perceptron 11
2.6 High-level Synthesis design steps [10] 14

3.1 Overall architecture of SpMM engine in AWB-GCN [15] 21
3.2 EnGN hardware architecture [18] 23
3.3 Block diagram of a tile in the GNN accelerator proposed by Auten *et al.* [3] 24
3.4 Architecture overview of HyGCN [44] 26
3.5 High-level overview of GRIP [24] 27
3.6 Baseline dataflow architecture and the improved FlowGNN architecture [33] 30
3.7 Hardware architecture with two execution modes proposed by Zhang *et al.* [47] 33
3.8 BoostGCN framework overview [48] 34

5.1 FPGA Toolchain for Graph Neural Network Acceleration 46
5.2 Torch-MLIR flow 48
5.3 Synthesizer: SODA-OPT and PandA-Bambu overview [2] 53
5.4 Row-by-column matrix multiplication example 58
5.5 Matrix multiplication memory coalescing [8] 59

6.1 Performance comparison between PyTorch matmul function and accelerator 65
6.2 Matrix multiplication optimization comparison 66
6.3 GCN inference PyTorch-Accelerator comparison 68
6.4 GCN inference number of cycles comparison 70

7.1 Future FPGA Toolchain for Graph Neural Network Acceleration 74

List of Tables

6.1 Excerpt of GCN model inference profiling result 61
6.2 Benchmark of *torch.mm* PyTorch function 63
6.3 Comparison between dense and sparse PyTorch matmul functions 63
6.4 Cora sub-dataset used for GCN infernce 68
6.5 GCN inference time comparison 69
6.6 Model accuracy comparison between PyTorch and FPGA accelerator . . . 71

List of Algorithms

5.1 Naive matrix multiplication algorithm 57

List of Listings

5.1 Class of GCN model 47
5.2 Forward function of GCN layer 47
5.3 Matrix multiplication in MLIR affine dialect 54
5.4 Unrolled matrix multiplication in affine dialect with unrolling factor 5 . . . 55

List of Symbols

Notation	Description
$\mathcal{G} = (V, E)$	The input graph for the GNN
V	Set of vertices of the graph
E	Set of edges of the graph
$\mathcal{N}(v)$	Set of neighbors of vertex v
$A \in \mathbb{R}^{N \times N}$	Adjacency matrix of $\mathcal{G}$ (N : number of nodes)
$\tilde{D}$	Degree matrix of the graph
$W^{(l)}$	Weight matrix of the neural network (l : layer)
$H^{(l)}$	Input node features matrix (l : layer)
h_v	Node representation of node v
ϵ	Learnable parameter or fixed scalar
I	Identity matrix

Acknowledgements

Acknowledgements here...