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

Here goes the Abstract in English of your thesis followed by a list of keywords. The Abstract is a concise summary of the content of the thesis (single page of text) and a guide to the most important contributions included in your thesis. The Abstract is the very last thing you write. It should be a self-contained text and should be clear to someone who hasn't (yet) read the whole manuscript. The Abstract should contain the answers to the main scientific questions that have been addressed in your thesis. It needs to summarize the adopted motivations and the adopted methodological approach as well as the findings of your work and their relevance and impact. The Abstract is the part appearing in the record of your thesis inside POLITesi, the Digital Archive of PhD and Master Theses (Laurea Magistrale) of Politecnico di Milano. The Abstract will be followed by a list of four to six keywords. Keywords are a tool to help indexers and search engines to find relevant documents. To be relevant and effective, keywords must be chosen carefully. They should represent the content of your work and be specific to your field or sub-field. Keywords may be a single word or two to four words.

Keywords: here, the keywords, of your thesis

Abstract in Lingua Italiana

Qui va l'Abstract in lingua italiana della tesi seguito dalla lista di parole chiave.

Parole chiave: qui, vanno, le parole chiave, della tesi

Contents

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 neural 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) outperform other computing devices, such as CPUs and 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 this reason, researchers have been actively pursuing the development of new FPGA accelerators for Graph Neural Networks (GNNs) in recent times.

The conventional 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 [29], 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 (GNNs) FPGA accelerator with minimal effort required.

The suggested toolchain represents an enhancement of the SODA toolchain [3]. It operates by transforming the PyTorch model, provided as input, into a multi-level intermediate representation (MLIR) [25] utilizing Torch-MLIR [33], 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 [12].

In pursuit of the thesis goal, various optimizations were adopted throughout the process. Specifically, efforts were made to optimize specific computations in Graph Neural Networks. As these networks often deal with massive graph sizes, the computation time and memory requirements are substantial. Consequently, a significant portion of the research focuses on optimizing the computation phase of Graph Neural Networks using custom optimizations.

This analysis aims to provide valuable insights for future research endeavors, enabling the development of solutions to overcome these limitations and further enhance the proposed toolchain.

TODO: add something about results

1.1 Contributions

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.

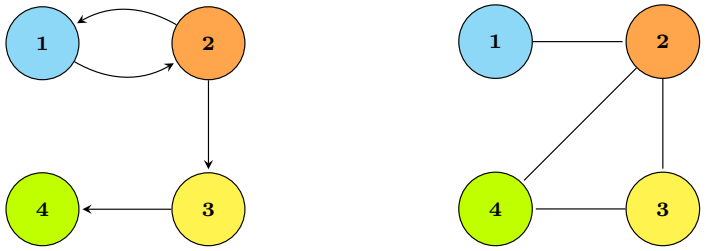
2 Background

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 and Bambu, which are integral parts of the SODA Toolchain that served as the foundation for this research.

2.1 Graphs

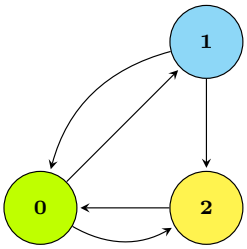
Graphs are data structures representing a collection of objects, known as vertices or nodes, and a set of edges connecting them [43]. In a graph, the edges can be either directed or undirected, as shown in Figure 2.1, and they typically connect two vertices, which may or may not be distinct. The vertices 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, we refer to their connection as a covalent bond. These bonds with distinct atomic distances can vary in nature, such as single or double bonds. Representing this complex three-



(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

dimensional object as a graph offers a practical and widely adopted abstraction, where atoms are nodes and covalent bonds act as edges [11].

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 way 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 slower 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 phase.

There are two common representations for sparse matrices: the Coordinate List (COO) format and the Compressed Sparse Row (CSR) format, which are explained 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 matrices with relatively few non-zero elements 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 $m + 1$. Figure 2.3b shows the CSR format of the adjacency matrix represented in Figure 2.2 is reported.

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 (GNNs) 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 [21]. Figure 2.4 illustrates the steady growth in the number of publications related to Graph Neural Networks (GNNs) on Google Scholar from 2015 to 2022. 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. 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 [30].

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, attributes might be aimed to be predicted such as its likelihood 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

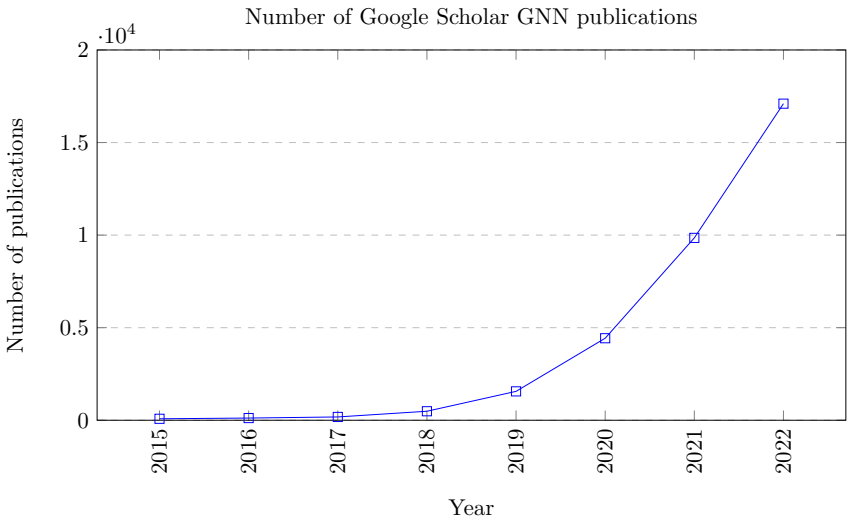


Figure 2.4: Number of GNN publications on Google Scholar per year

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 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 (GNNs) 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 [39] (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 [16] (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 [26].

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 is provided in the following subsections.

2.2.1 Graph Convolutional Network

A graph convolutional network (GCN) [10, 24] 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.

2.2.2 Graph Isomorphism Network

A Graph Isomorphism Network (GIN) [10, 37] 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 [37]. 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 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 [37].

2.3 SODA Toolchain

SODA [3] 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 pro-

gramming frameworks, applying advanced optimizations. It also includes a compiler-based 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 (FPGAs).

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 (IR). Leveraging the multi-level intermediate representation (MLIR), the frontend facilitates hardware/software partitioning of algorithm specifications and performs architecture-independent optimizations. Following this, it generates a low-level IR (LLVM IR) that is utilized by the hardware generation engine, PandA-Bambu [12]. PandA-Bambu can accept LLVM IR as input, making it a cutting-edge open-source HLS tool. 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.

2.3.1 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. MLIR is a framework that facilitates the development of reusable, extensible, and modular compiler infrastructure by defining dialects. These dialects serve as self-contained intermediate representations (IRs) that adhere to the meta-IR syntax of MLIR. By utilizing dialects, code can be modeled at different levels of abstraction, allowing for specialized representations that aid in specific compiler optimizations.

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.

2.3.2 SODA Synthesizer Backend

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. 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, Bambu can automatically generate testbenches for verification purposes. Using Bambu, the SODA synthesizer can target both FPGAs and ASICs.

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, 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.

2.4 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 (GNNs) has become a subject of intense interest within the research community, encompassing the exploration of ASIC and FPGA accelerators. In this chapter, a comprehensive examination is conducted on cutting-edge Graph Neural Networks FPGA accelerators and design flows based on High-Level Synthesis (HLS). As explained in Chapter 6, particular emphasis has been placed on optimizing matrix-matrix multiplication during this thesis research study. Consequently, this chapter also delves into the relevant literature concerning various approaches to Matmul optimization.

3.1 Chapter structure

This chapter contains several sections. Firstly, it presents the software frameworks utilized to accelerate Graph Neural Network computations. The following section provides an overview of state-of-the-art hardware accelerators, categorized based on their architecture types [1].

Subsequently, a section summarizes an accelerator implemented using High-Level Synthesis (HLS). This accelerator is separated from the hardware accelerators as it adopts HLS as the design flow proposed in this thesis.

Additionally, this chapter includes a summary of a solution that aimed to accelerate GNN using both software and hardware approaches. A section is dedicated to a state-of-the-art graph processing accelerator, implemented using HBM-equipped FPGAs.

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, especially those related to technologies similar to the ones employed in this thesis.

Finally, the chapter concludes with a comprehensive summary of the cutting-edge accelerators presented.

3.2 Software accelerators

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 research studies tried to bridge the gap by adapting the DNN libraries to overcome Graph Neural Network challenges.

The two main software frameworks trying to accelerate Graph Neural Networks computation are PyTorch Geometric [13] and Deep Graph Library [34]. They both provide a lot of examples and code for multiple GNN architectures providing optimizations that could work for the acceleration 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 relational learning and 3D data processing domains. PyTorch Geometric achieves remarkable data throughput by introducing efficient handling of mini-batches containing input examples of varying sizes and efficiently handling sparsity through specialized GPU scatter and gather kernels, which operate on all edges and nodes concurrently, as opposed to relying on sparse matrix multiplication kernels. A key aspect of PyG 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. Specifically, both sampled dense-dense and sparse matrix multiplications and options for node, edge, or feature parallelization are considered. DGL intelligently selects the optimal parallelization scheme using heuristics, considering various factors, including the input graph. 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 [34]. However, both libraries target CPU and GPU architectures. Knowing the extreme 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 CPU-GPU targeting libraries.

3.3 Hardware accelerators

As discussed in Section 3.2, software accelerators 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, the research field has raised questions about the feasibility of custom hardware accelerators in overcoming 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 fabric is configured to be versatile and flexible, allowing it to handle various applications and tasks. Instead of having specialized and fixed hardware modules for specific functions, a unified architecture enables the FPGA to reconfigure its resources to dynamically adapt to different computation requirements.

[14] presents Autotuning-Workload-Balancing GCN (AWB-GCN) to accelerate Graph Convolutional Network inference. This accelerator endorses a proactive adaptation to the structural sparsity inherent in 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 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 to the most underloaded 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 represents

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

EnGN [17] 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.

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.

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. This approach enhances EnGN's capability to handle substantial graphs effectively.

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.

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 [4] 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 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. Each tile's internal structure resembles HyGCN's [1], with the DNA functioning as an array for dense multiplication, the AGG as an edge-controlled adder, the DNQ as an inter-engine buffer, and the GPE overseeing execution.

The GNN accelerator program proposed by Auten *et al.* 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.

Unlike HyGCN, the accelerator introduced in [4] is less specialized but has a better potential for generalization to various Graph Neural Network models [1].

3.3.3 Hybrid architectures for GNN acceleration

HyGCN [38] 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 functions 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 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 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 within the same interval and processed the aggregation of their source neighbors interval by the interval. 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.

While not an authentic hybrid architecture, GRIP [23] is an accelerator that shares similar techniques with HyGCN's implementation approach. It leverages GReTA [22] (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 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.

3.4 High-Level Synthesis based accelerators

As previously highlighted, the main challenge of GNN hardware acceleration lies in simultaneously meeting the demand for novel GNN models and fast inference, as there exists a gap between the difficulty in developing efficient FPGA accelerators and the rapid pace of creating new GNN models.

To address this challenge, in [2] GenGNN has been introduced, a GNN acceleration framework utilizing High-Level Synthesis (HLS), 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 embedding (NE). 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 embedding encompasses node transformation and update.

To achieve this goal, GenGNN was designed using a message-passing style featuring two main processing elements (PEs): node embedding and message passing. The 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 embedding 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 merged fashion reduces the overall process latency and minimizes memory cost.

The independence of node embedding (NE) and message passing (MP) 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 streaming-based pipelining, which can be achieved using a streaming-based FIFO (first-in-first-out) memory queue. 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.

GenGNN enhances its architecture's adaptability to different graph neural network models by offering various model-specific components. One particularly advantageous feature of GenGNN's streaming-based pipelining for node/edge processing is its suitability for models with virtual nodes. As defined by [15], 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.

Finally, another feature provided by GenGNN is the large graph extension. The authors implemented a prefetcher to accommodate large graphs that cannot be stored on-chip.

This prefetcher retrieves consecutive nodes' degrees from DRAM and stores them in an on-chip FIFO buffer. As the message passing PE requires, it loads each subsequent node's degree, prompting the prefetcher to refill the buffer. This clever mechanism effectively conceals the latency associated with fetching from the off-chip degree table, ensuring that the message passing PE behaves similarly to handling small graphs. Moreover, the authors adopted packed data transfer by typecasting off-chip array pointers into the desired size pointer types, facilitating the transfer of more significant bits between DRAM and the system with every clock cycle.

It can be observed that GenGNN's NE/MP pipeline shares a similar concept with the task scheduling approach of BoostGCN [42].

BoostGCN presents a framework 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 two distinct computation phases. They developed a low-overhead task scheduling strategy to tackle any potential pipeline 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 [2], while the scheduling approaches of GenGNN 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 GenGNN processes the nodes on-the-fly in FPGA in an adaptive manner. Secondly, BoostGCN employs a buffer in external memory, while GenGNN utilizes an on-chip FIFO to queue the nodes ready for message passing.

Another notable framework in this section is DGNN-Booster [9], an innovative Field-Programmable Gate Array (FPGA) accelerator framework designed for real-time inference of Dynamic Graph Neural Networks (DGNNs) using High-Level Synthesis (HLS). 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. Hence, a detailed description is not provided here. However, it is worth mentioning that DGNN-Booster implements GNNs using a message-passing mechanism based on GenGNN at a lower level of parallelism.

3.5 Software-Hardware co-design accelerators

The work conducted by Zhang *et al.* [41] introduces a combined software and hardware approach for accelerating Graph Convolutional Networks (GCNs). Their study was initiated with the recognition that hardware acceleration of Graph Convolutional Network (GCN) 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 expedite GCN models, with a particular focus on accelerating 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 (GCN) 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 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 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.

GCoD [40] 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 balanc-

ing 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.

Similar to GraphLily, as discussed in Section 3.6, GCoD also leverages High Bandwidth Memory (HBM). Specifically, each sub-accelerator communicates with an off-chip HBM through direct memory access to enhance access efficiency.

3.6 Graph processing acceleration using HBM-equipped FPGAs

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. Also, a part of this research follows this approach, which will be extensively discussed in Chapter 6.

GraphLily [19] 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 [20] 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's 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, effectively 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. Prior to proceeding, it is essential to outline that Basic Linear Algebra Subprograms (BLAS) specify low-level routines for executing fundamental linear algebra operations, including matrix multiplication. OpenBLAS [36] is a prominent optimized BLAS library tailored for CPU usage. At the same time, cuBLAS [28] serves as a specialized library providing GPU-accelerated implementations of BLAS.

Also, several FPGA-accelerated BLAS libraries exist, designed to exploit FPGA's parallelism and hardware capabilities for efficient matrix operations. One example is GraphLily,

discussed in Section 3.6, but other options exist. Various custom implementations are crafted to suit specific FPGA platforms and applications.

However, in this Section, particular attention will be given to the optimization of matrix multiplication using MLIR. This focus arises because SODA, the framework introduced in Section 2.3, 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, a novel intermediate representation, 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 this paper chose to base their example 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 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 FMA/add mul pipeline 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, 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, 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) [18], which will be discussed in Chapter 6 as it was also used for part of this thesis' experiments, aim to provide a representative set of graphs and GNNs for benchmarking purposes.

Secondly, 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.

In conclusion of this chapter, Table 3.1 summarizes the discussed accelerators, providing an overview of their most important features.

Name	Features
Auten *et al.*	- Tiled architecture - Specialized hardware units
AWB-GCN	- Three hardware-based autotuning techniques - Adapting to varying GNN workloads
BoostGCN	- Partition-Centric Feature Aggregation scheme - Centralized load balancing scheme
EnGN	- Unified architecture - Aggregation via Ring-Edge Reduction (RER)
GCoD	- Algorithm and accelerator Co-Design - Dedicated two-pronged accelerator
GenGNN	- Using High-Level Synthesis (HLS) - Highly optimized message passing architecture
GRIP	- Uses the GReTA abstraction - Vertex-tiling optimization
HyGCN	- Hybrid architecture, aggregate/combine phases - Sparsity reduction via window sliding/shrinking
Zhang *et al.*	- Software and hardware co-design - Double buffering, node and feature parallelism

Table 3.1: Summary of discussed Graph Neural Network accelerators

4 Problem Formulation

Problem formulated in a clear way, what we did and how, with open issues and thesis goals.

5 FPGA Toolchain for Graph Neural Network Acceleration

Introduction of the way I faced the problem, with the motivation for the followed approach. Explanation of the toolchain in a clear way.

6 Experimental Results

Chapter dedicated to the outcome of the results, what I have obtained and what limitations have been encountered. Explaining the still open issues and research suggestions.

7 Conclusions and Future Developments

Final chapter containing the main conclusions of my research and possible future developments.

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

[3] 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.

[4] 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.

[5] 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.

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

[11] 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`.

[12] 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.

[13] 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`.

[14] 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`.

[15] 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`.

[16] 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`.

[17] 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`.

[18] 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`.

[19] 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.

[20] 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`.

[21] 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`.

[22] 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.

[23] 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`.

[24] 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`.

[25] 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.

[26] 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`.

[27] 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.

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

[29] 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`.

[30] 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.

[31] 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`.

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

[33] 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`.

[34] 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`.

[35] 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`.

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

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

[38] 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`.

[39] 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`.

[40] 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 Inter-*

national Symposium on High-Performance Computer Architecture (HPCA), pages 460–474, 2022. doi: 10.1109/HPCA53966.2022.00041.

[41] 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.

[42] 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.

[43] 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

List of Tables

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...