POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

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

Master of Science in Computer Science and Engineering

Author: Giovanni Demasi

Advisor: Prof. Fabrizio Ferrandi

Co-advisors: Serena Curzel, Michele Fiorito

Academic year: 2022-2023

---

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.
A growing number of applications now generate data 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. However, the conventional approach to hardware design 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 [4], a cutting-edge high-level programming framework for creating neural network algorithms based on the Python programming language, enables the automatic generation of a GNNs FPGA accelerator with minimal effort required.

TODO: Add half column about results and contributions

2. Background

Graphs are data structures representing a collection of objects, known as vertices or nodes, and a set of edges connecting them. In a graph, the edges can be either directed or undirected, 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 are easy to visualize, but a more formal way is needed when implementing graph algorithms. The adjacency matrix of a graph provides information about the relationships between nodes in the graph. It 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, zero otherwise. A feature vector represents the features or attributes associated with a single entity.
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.
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. In a graph-level task, the objective is to predict the property or characteristic of an entire graph. Node-level tasks involve predicting the identity or function of individual nodes within a graph. The remaining prediction task in graphs pertains to edge prediction.
At its core, a GNN is an algorithm that exploits the connectivity within a graph to understand and represent the relationships between nodes. GNNs consist of multiple interconnected layers and, by relying on the graph's structure, they 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.
Different popular GNN architectures have been proposed recently, some of which are more suitable for some tasks than others. The main model used for this thesis is the *Graph Convolutional Network* (GCN) [3].

2.1. Graph Convolutional Network

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.
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:

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

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 1 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.

3. Problem Formulation

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.
To explain 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. This problem can be represented as a graph, where users items are nodes and interactions between users and items 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 enhances the efficiency and scalability of the recommendation system, resulting in quick and precise recommendations to users.

4. 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 (HLS). Figure 1 illustrates the entire design flow, with the steps involved in the GNN acceleration process.

4.1. PyTorch

The first step of the toolchain is to design and implement the Graph Neural Network model in PyTorch [4].

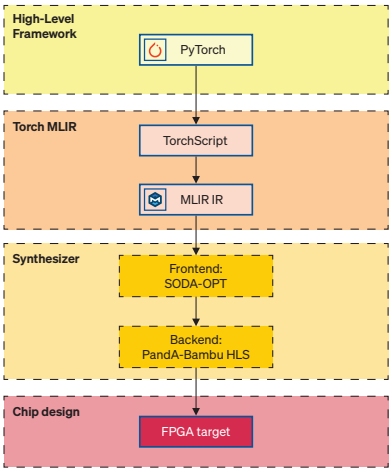

Figure 1: FPGA Toolchain for Graph Neural Network Acceleration

The main model used for this thesis is the Graph Convolutional Network **??** performing node classification on the *Cora* dataset.

4.2. Torch-MLIR

Torch-MLIR [5] offers compiler support for transitioning from the PyTorch ecosystem to the MLIR ecosystem.
For this thesis, TorchScript represented the starting point of the flow that Torch-MLIR follows to go from PyTorch to MLIR. TorchScript 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 result MLIR can use different dialects, but the one used for this thesis is the Linalg dialect, which serves as input for the next phase of the toolchain.
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. Some of the used adaptations include making jittable the GNN class, annotate the parameters of the propagate function, using some types assertion and annotate the type of the forward function.
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 input example of the model and the desired output type. The Graph Neural Network model must have been already trained, being ready for inference. The second parameter, the input example 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 is to choose which of these three dialects has to be used for the output MLIR.

4.3. Synthesizer

The synthesizer represents the final step of the toolchain, which optimizes and synthesizes the MLIR representation, targeting FPGA. This step includes SODA-OPT [1] and PandA-Bambu [2].

SODA-OPT

SODA-OPT 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. The output of SODA-OPT is an LLVM representation that serves as input to PandA-Bambu HLS.
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,
The next step consists in outlining the part of MLIR code to accelerate by adding the soda.launch and soda.terminator flags at the beginning and end of the piece of code.
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 plus a set of passes tailored for soda optimizations.
Profiling the inference function of the GCN model, the result showed that, in general, nearly 60% of the time required to make a prediction was used for the matrix multiplication operation. The result of this research showed that the most impressive improvement in terms of performance is given by the loop unrolling technique, which perfectly allows to exploit the extreme parallelism available on FPGAs.
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.

PandA-Bambu

PandA-Bambu represents the last phase of the synthesis. It receives the LLVM representation as input, and after having applied some optional low-level optimizations, it performs the typical steps of HLS, including allocation, scheduling and binding.
PandA-Bambu allows the specification of different optimizations and settings that can have a big impact on accelerator performance. The most evaluated optimization is the number of memory channels.
A comparative analysis has been performed between different number of memory channels, including the minimum, 2 channels, and the maximum, 32 channels.
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.

5. Experimental Results

6. Conclusions

A final section containing the main conclusions of your research/study have to be inserted here.

References

[1] Nicolas Bohm Agostini, Serena Curzel, Jeff Jun Zhang, Ankur Limaye, Cheng Tan,

Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Brooks, Gu-Yeon Wei, and Antonino Tumeo. Bridging python to silicon: The soda toolchain. *IEEE Micro*, 42(5):78–88, 2022.

[2] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino 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.

[3] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[4] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[5] nod.ai team Torch-MLIR team and other contributors. Torch-mlir: Mlir based compiler toolkit for pytorch programs, 2021.