POLITECNICO
MILANO 1863

An FPGA toolchain for Graph Neural Network acceleration using High-Level Synthesis

ADVISOR
FABRIZIO FERRANDI

AUTHOR
DEMASI GIOVANNI

CO-ADVISORS
SERENA CURZEL, MICHELE FIORITO

Contents

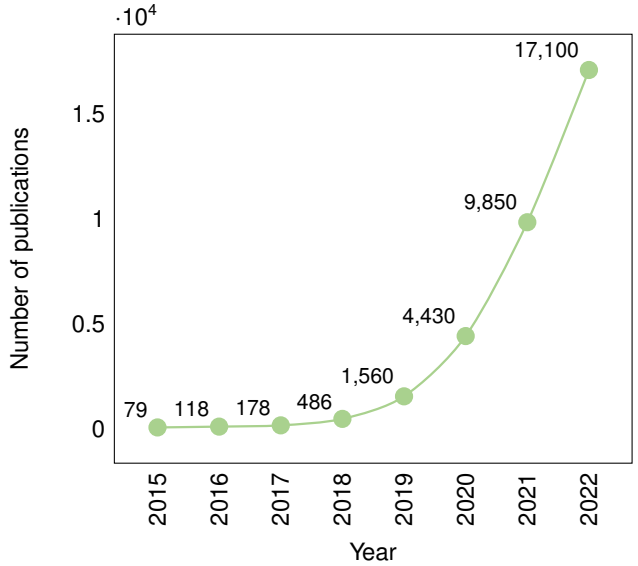
Thesis motivations and objectives

Motivations of the thesis

Optimizing and accelerating the capabilities of Graph Neural Networks is necessary due to their increasingly popularity, especially in fields characterized by vast amount of data.

Objectives of the thesis

- A comprehensive toolchain for Graph Neural Network inference acceleration on FPGA architectures
  1. Identify the models bottlenecks and accelerate the heaviest computational operations
  2. Identify customized optimizations designed to finely enhance model performance
  3. Usage of the toolchain for the generation of a GNN inference accelerator

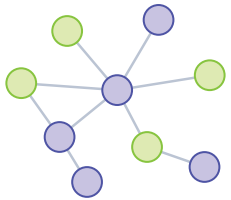


Growing popularity of Graph Neural Networks in past years on Google Scholar.

Graph Neural Networks

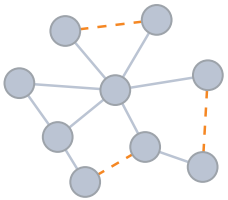
Graph Neural Networks are deep learning techniques that operate on graph-structured data to solve prediction tasks:

Node classification



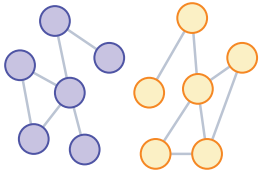
Classify nodes according to labels of their neighbours

Link prediction



Predict the evolution of connection between two entities

Graph classification



Classify the whole graph into different categories

Graph Neural Networks consist of multiple interconnected layers and typically encompasses three main stages:

pre-processing → **Iterative updates** → **readout**

transforming the input data

edge and vertex feature vectors update

aggregates embeddings in a global feature vector

Design of the toolchain

The toolchain proposed in this thesis makes use of MLIR and is based on High-Level Synthesis.

MLIR

A novel approach to construct reusable and extensible compiler infrastructure.

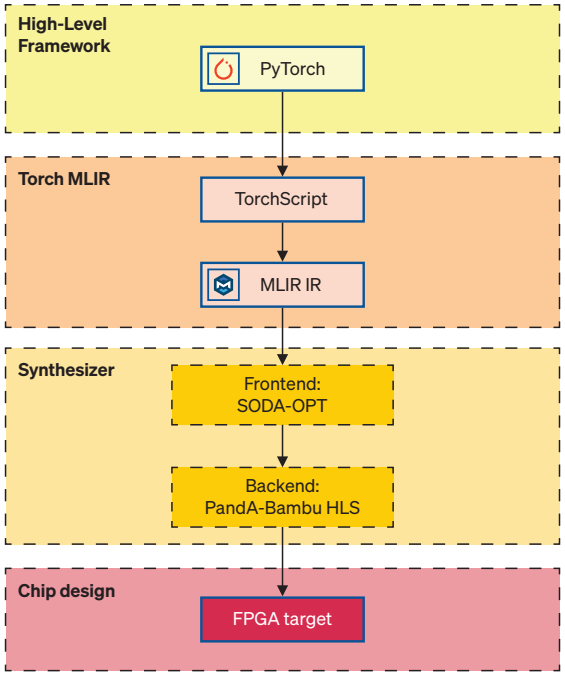
High-Level Synthesis

Simplified hardware development. Reduces design time and effort.

C/C++
↓
HLS tool
↓
VHDL

Synthesizer

It takes the MLIR representation as input and transform it into accelerator. Firstly, it apply high-level optimizations, then the refined version proceeds to the Bambu HLS backend, where the GNN accelerator is effectively produced.



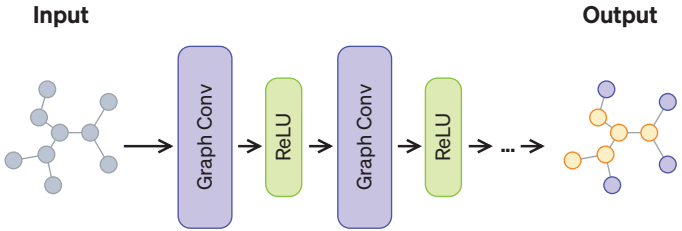
FPGA Toolchain for Graph Neural Network Acceleration

Toolchain inputs

PyTorch

Toolchain entry point, for neural network implementations.

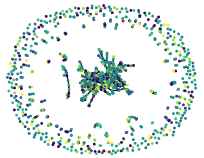
Graph Convolutional Network



Model's characteristics:

- Two convolutional layers
- Convolutional layers made of two matrix multiplications
- Task is node classification

Cora dataset



- Scientific publications
- Multiclass classification
- Subsets to contain synthesis and evaluation times

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

Datasets used for experiments: subsets of Cora dataset.

Performance analysis

Model analysis

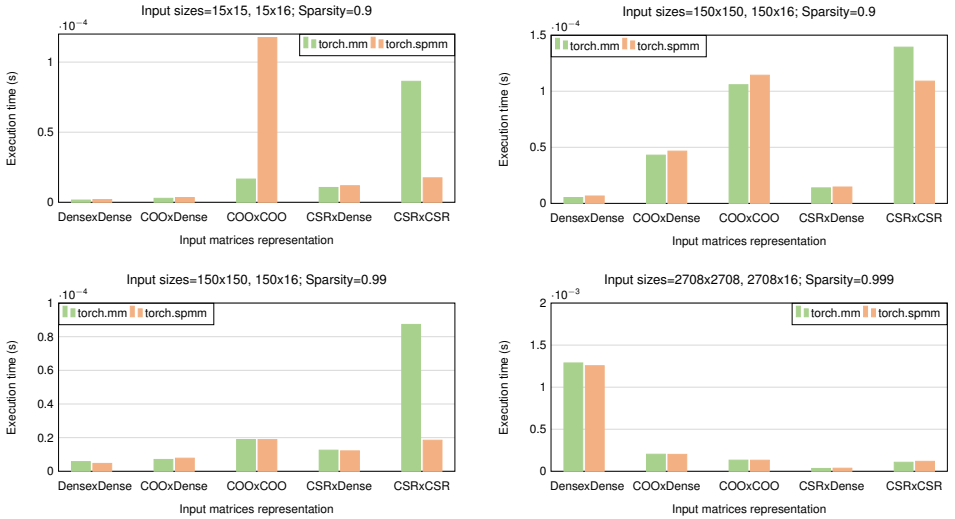
Analysis to identify the computational bottlenecks of the model

Name	%
aten::mm	50.25%
aten::addmm	36.30%
aten::add	4.64%

aten::mm=matrix multiplication; aten::add= matrix sum; aten::addmm=matrix multiplication plus matrix sum

>50% of the computational time used by matrix multiplication

PyTorch matrix multiplication analysis



PyTorch CPU execution time of matrix multiplication with dense and sparse matrix representations. COO=Coordinate list; CSR=Compressed Sparse Row

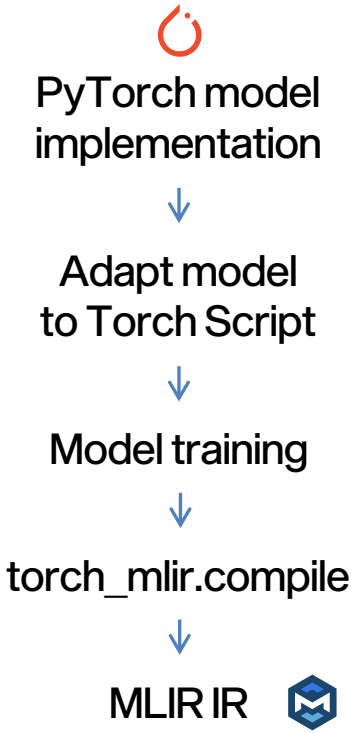
sparse matrix multiplication faster than dense for big matrix sizes with density ≤ 0.001

From PyTorch to MLIR

Torch-MLIR
The Torch-MLIR project aims to provide first class compiler support from the PyTorch ecosystem to the MLIR ecosystem.

It leverages torch script, an intermediate representation used to generate serializable and optimizable models directly from PyTorch code; it represents the bridge between PyTorch and Torch-MLIR.

After having created a set of general rules to adapt different GNN models to torch script, the GCN model has been adapted to Torch Script; then it has been compiled using Torch-MLIR to obtain its MLIR representation.



From PyTorch model implementation to MLIR IR flow

Synthesis

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

SODA-OPT

MLIR representation

↓

Outline kernel function

↓

High level optimizations

↓

Low-Level IR

- High level optimization
- Subset of MLIR passes
- Output represents the input of Panda-Bambu

PandA-Bambu

Low-Level IR

↓

Low level optimizations

↓

RTL IR

↓

Accelerator

- High-Level Synthesis tool
- low level optimizations
- output is the final accelerator

FPGA-accelerated matrix multiplication

Experimental phase setup

CPU
Intel Core i9, 8 cores, 2.3 GHz

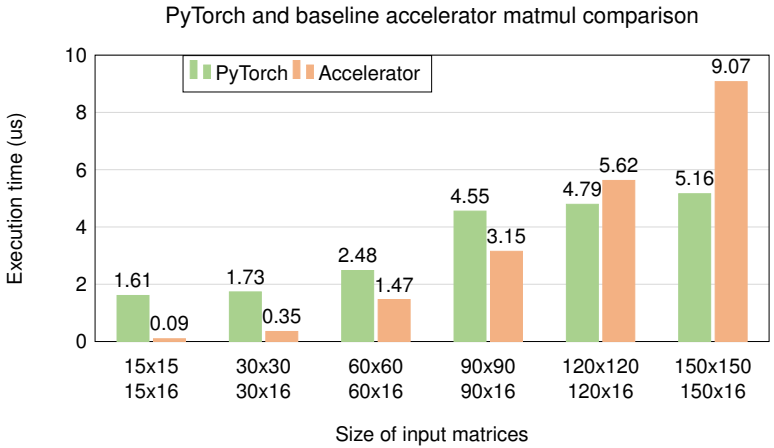
Accelerators
AMD Virtex UltraScale+ (Alveo U280) FPGA.

Analysis results

The baseline accelerator is faster than PyTorch when matrices are relatively small. The reasons behind this behaviour may lie in the fact that:

1. The generated accelerator assumes that all data is available in BRAM
2. PyTorch times have been measured using all available threads on the machine, exploiting more parallelism with respect to the accelerator

Analysis to identify performance differences between PyTorch matrix multiplication operation and the baseline accelerator.



Comparative analysis between PyTorch matrix multiplication operation and the baseline accelerator performance.

Synthesis optimizations

To make the accelerator able to exploits more parallelism, two main optimizations have been applied: loop unrolling and parallel memory accesses.

Loop unrolling

before

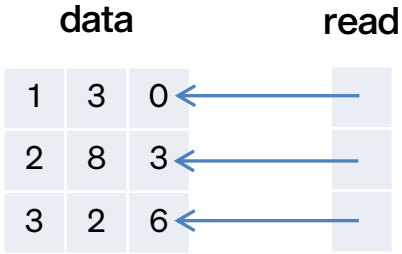
```
for i in range(0,4){
        code
}
```

after

```
for i in range(0,2){
        code
        code
}
```

- Expanding completely or partially the loop, according to unroll factor
- Unroll factor parameter to decide the number of loop iterations to unroll
- More chances of parallelization

Parallel memory access



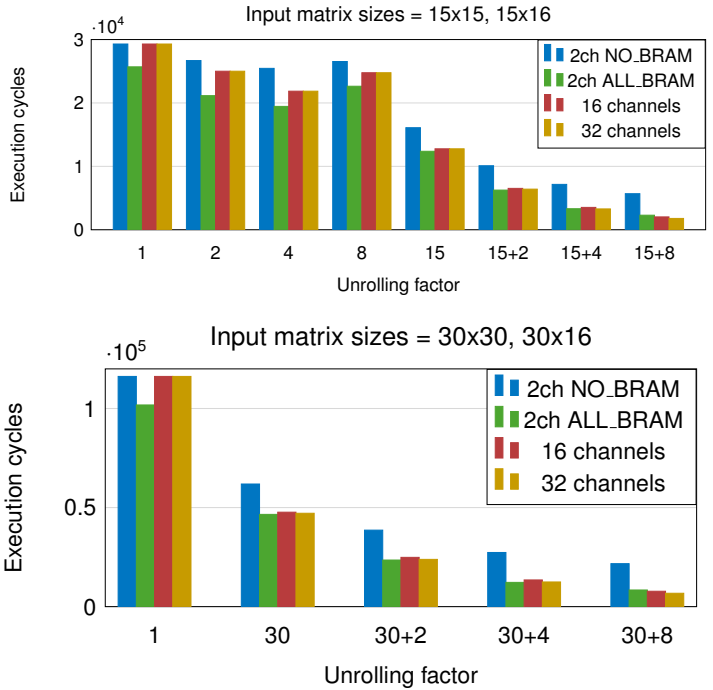
- Increasing the number of memory channels for parallel r/w
- Better exploitation of parallelism
- More parallel memory accesses but more cycles due to external memory

Optimized matrix multiplication accelerator

Analysis aiming to analyse the relation between the number of execution cycles when using BRAM/external memory and the unroll factor.

Analysis results showed that, in some situations, using external memory is better than BRAM:

- Small unroll factor: better setting is using 2 channels, storing all memory objects in BRAMs
- Large unroll factor: 32 channels with external memory is the best option
- 2 channels with external memory is always the least favourable choice
- Usage of external memory is beneficial when large volume of data can be simultaneously loaded and stored to offset the additional cycles required for external memory

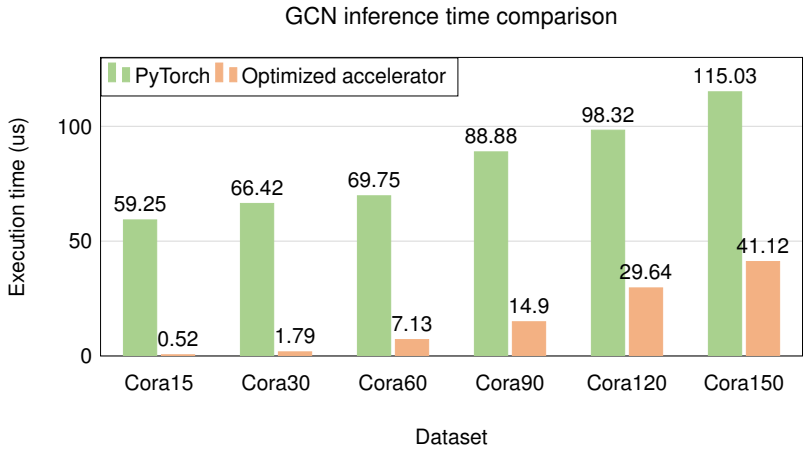


Cycles comparison between accelerator configurations using different number of channels with BRAM/external memory (NO_BRAM), for increasing unroll factor.

GCN accelerator

Application of the analysed optimizations to the Graph Convolutional Network.

- Accelerator outperforms PyTorch
- Performance improvement maintained with increasing dataset sizes



Graph Convolutional Network inference time comparison between PyTorch and FPGA implementations for different sizes of the dataset. Optimized accelerator uses 2 channels with BRAMs and one full unroll.

Analysis to measure difference in floating precision on FPGA implementations and its impact on model accuracy.

- Synthesis has not impacted the accuracy of the model
- Average error is stable; not affected by the dataset size
- Low probability of misclassification

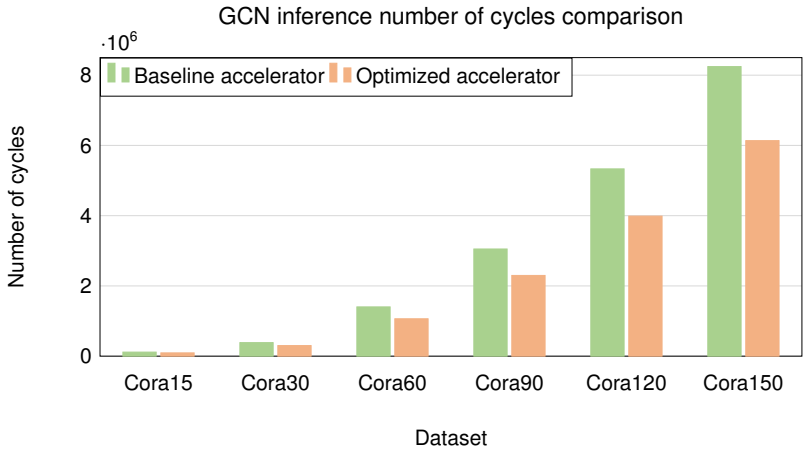
Dataset	PyTorch	Accelerator	ϵ	$\bar{\epsilon}$
Cora15	0.333	0.333	1.31E-06	0.26E-06
Cora30	0.333	0.333	8.64E-06	0.54E-06
Cora60	0.166	0.166	4.02E-06	0.25E-06
Cora90	0.555	0.555	19.47E-06	0.45E-06
Cora120	0.291	0.291	19.86E-06	0.27E-06
Cora150	0.533	0.533	23.84E-06	0.30E-06

Accuracy comparison between PyTorch and Accelerator. ϵ=total error; $\bar{\epsilon}$=average error.

GCN accelerators comparison

Analysis to show how the accelerator using 2 channels with BRAMs and one full unroll of the innermost loop has been affected by loop unrolling technique, compared to the baseline performance with 2 channels BRAMs and unroll factor equal to 1.

- The baseline accelerator does not exploit parallelism at all
- The optimized accelerator exploits parallelism more and more as the size of the dataset increases the speedup increases as the size of the dataset increases
- Trade-off between performance and area: more parallel operations means more computational units



Graph Convolutional Network number of cycles comparison between baseline and optimized accelerators for different sizes of the dataset.

State of the art analysis

The proposed toolchain offers new and improved ways to accelerate Graph Neural Network inference time on FPGA architectures.

Characteristic	State of the art	FlowGNN	Toolchain
Supported models	Mostly focused on a specific model, only some solutions are generalizable	Set of implementations in C++ that can be modified	Any models that can be implemented in PyTorch
Customization	Mostly not customizable, only few propose some settable parameters	Four configurable parallelization parameters	Optimization during synthesis, fine-tuning of model performance
Hardware design	Mostly required, only one solution uses HLS tools	Use of HLS, no need of low-level programming	Use of HLS, no need of low-level programming

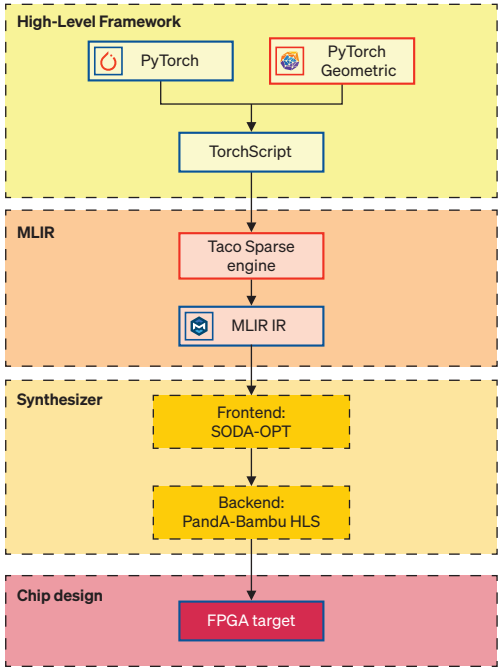
Conclusions and future developments

Accomplishments

- FPGA toolchain for GNN inference acceleration
- Enhanced compatibility of toolchain elements
  a. General set of rules to make models compatible with torch script
  b. Support of constant of tuple Type in Torch-MLIR
  c. Identification of area of improvements for PyTorch Geometric support
- Matrix multiplication and GNN accelerators with customized optimizations to finely enhance their performance

Future developments

- Support of PyTorch Geometric
- Support of sparse tensor operations, possible solution is PyTaco



Possible future developments to the proposed toolchain

POLITECNICO
MILANO 1863

Thanks for your attention