POLITECNICO
MILANO 1863

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

ADVISOR
FABRIZIO FERRANDI

AUTHOR
GIOVANNI DEMASI

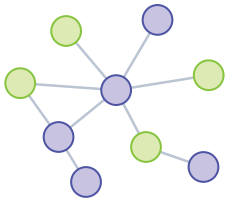
CO-ADVISORS
SERENA CURZEL, MICHELE FIORITO

Contents

Graph Neural Networks

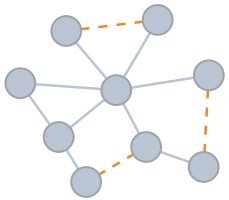
Graph Neural Networks are deep learning techniques that operate on graph-structured data to solve prediction tasks:

Node classification



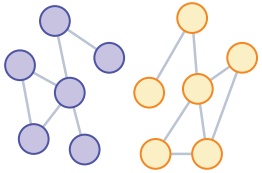
Classify nodes according to labels of their neighbours

Link prediction

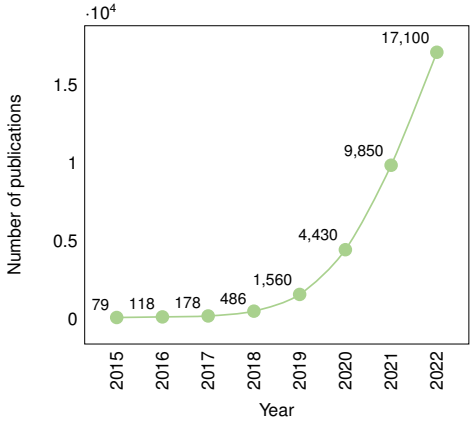


Predict the evolution of connection between two entities

Graph classification



Classify the whole graph into different categories



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

Graph Neural Networks consist of multiple interconnected layers, and they typically encompass 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

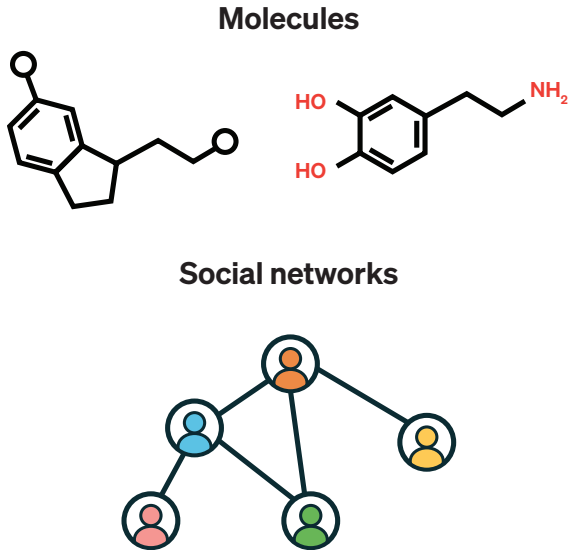
Thesis motivations and objectives

Motivations of the thesis

Optimize and accelerate 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. Analysis of state-of-the-art tools for Graph Neural Network implementation;
  2. Identify the GNN models bottlenecks and accelerate the heaviest computational tasks
  3. Design space exploration of optimizations designed to finely enhance model performance



Example of graph-structured data on which Graph Neural Networks can operate

Proposed 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.

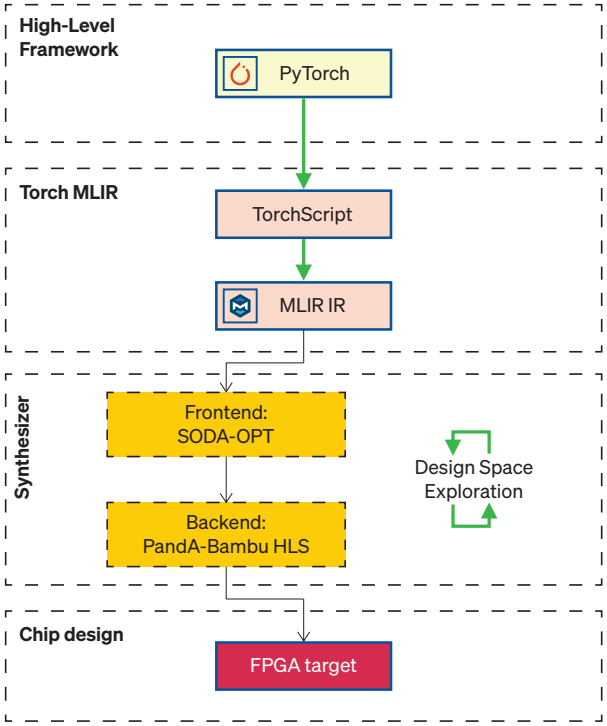
High-Level Synthesis

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 applies high-level optimizations, then the refined version proceeds to the Bambu HLS backend, where the GNN accelerator is effectively produced.



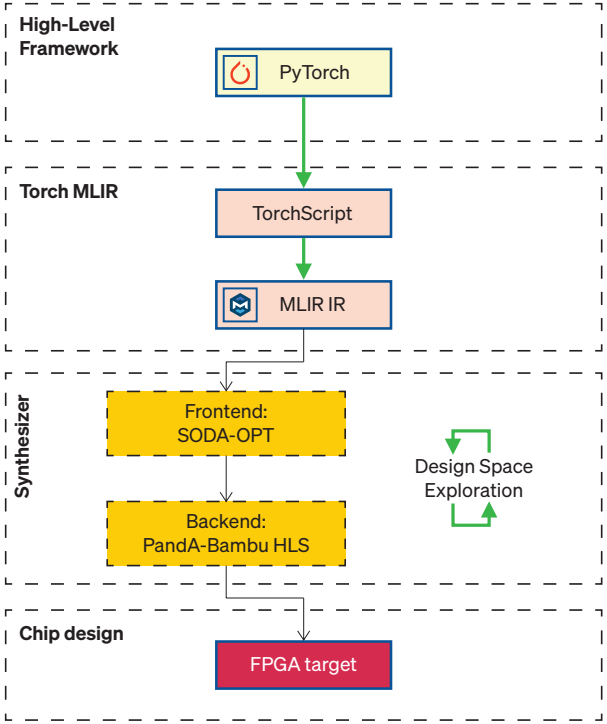
FPGA Toolchain for Graph Neural Network Acceleration. Main thesis contributions represented in green.

Contributions

The main contributions of this thesis are:

1. Analysis of the existing tools for the implementation of Graph Neural Networks;
2. Identification of missing features of Torch-MLIR and implementation of some of them;
3. Analysis of bottlenecks in the execution of a Graph Convolutional Network;
4. Exploration of the combination of SODA-OPT and PandA-Bambu optimizations, leading to the development of a novel default optimization pipeline for Graph Neural Networks.

The proposed GNN optimization pipeline allows to achieve an average of more than 3x improvement in inference time on FPGA.



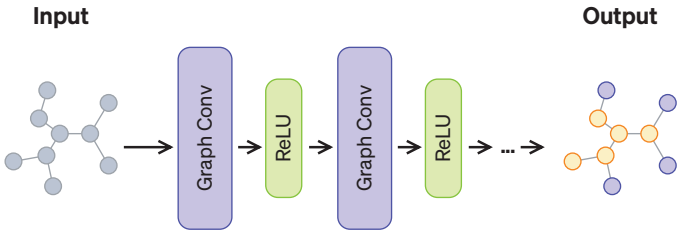
FPGA Toolchain for Graph Neural Network Acceleration. Main thesis contributions represented in green.

Toolchain inputs

PyTorch

Proposed toolchain entry point, for neural network implementations.

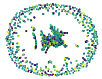
Graph Convolutional Network



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

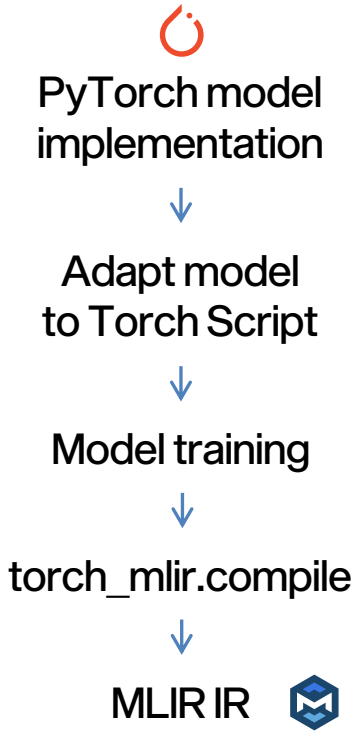
Datasets used for experiments: subsets of Cora dataset.

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.

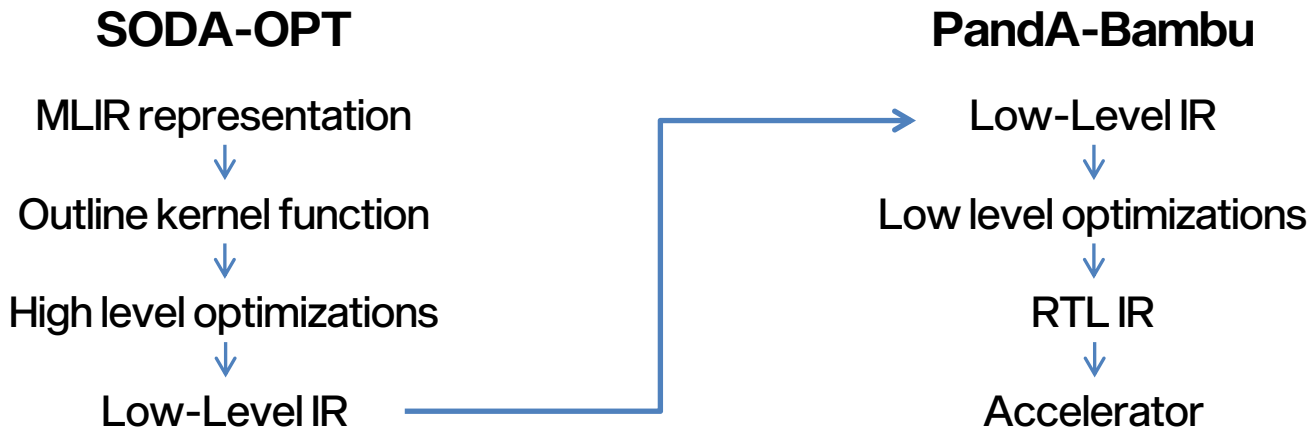
After having created **a set of general rules to adapt different GNN models to Torch script**, they have been applied to the GCN model obtaining its Torch script compatible version; then it has been compiled using Torch-MLIR to obtain its MLIR representation.



From PyTorch model implementation to MLIR IR flow

Synthesis

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



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

- High-Level Synthesis tool
- Low level optimizations
- Output is the final accelerator

GNN profiling on CPU

Model analysis

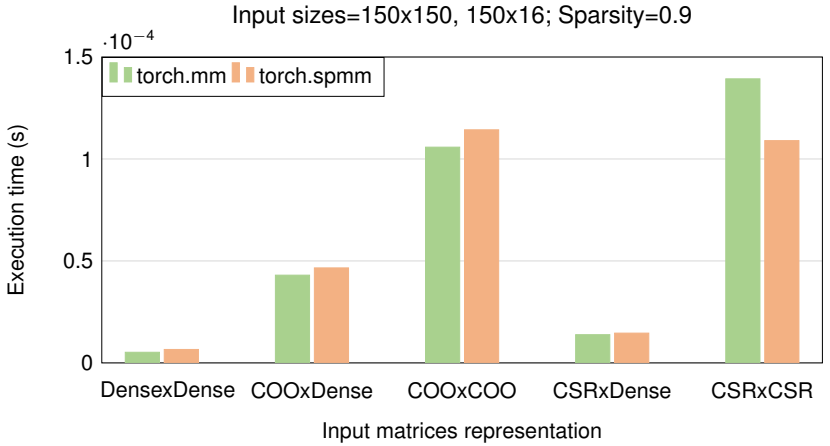
Analysis to identify the computational bottlenecks of the model

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

FPGA-accelerated matrix multiplication

Experimental phase setup

CPU
Intel Core i9, 8 cores, 2.3 GHz

Accelerators
AMD Virtex UltraScale+ (Alveo U280) FPGA.

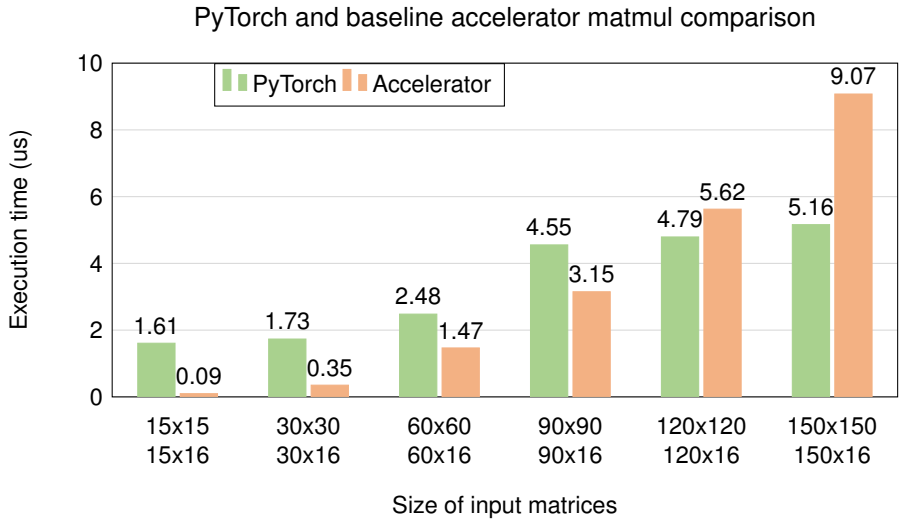
Experiment
Matrix multiplication synthesis through the proposed toolchain, without optimizations

Baseline
PyTorch CPU execution, 8 threads

Outcome
FPGA matrix multiplication accelerator, from ~17x to ~0.6x times faster, according to the size of the dataset

Analysis results
Baseline accelerator faster than PyTorch for small matrices. Possible reasons:

1. Accelerator assumes all data in BRAM
2. PyTorch exploits more parallelism by using all available threads



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

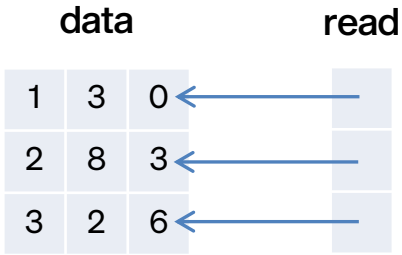
```
for i in range(0,4){
        code
}
```

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 number of mem channels for more parallel mem accesses
- Better exploitation of parallelism
- More parallel memory accesses but more cycles due to external memory

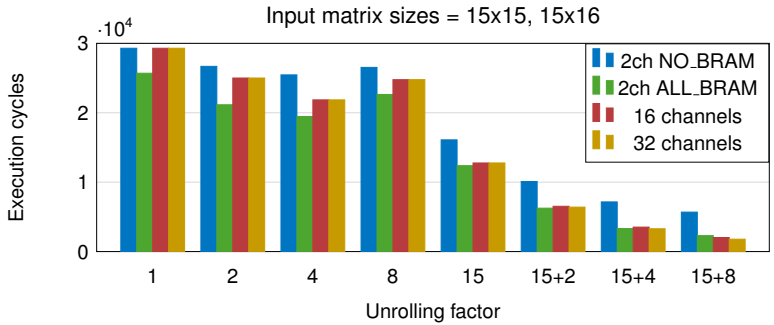
Optimized matrix multiplication accelerator

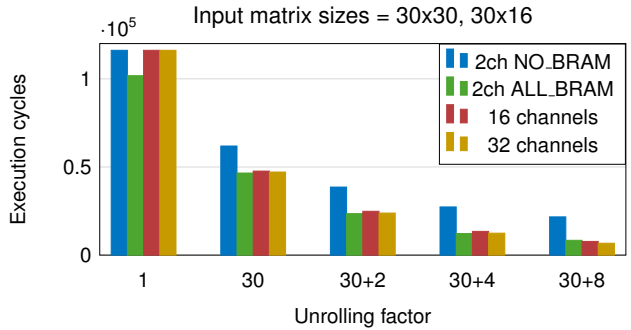
Experiment
Matrix multiplication synthesis through the proposed toolchain, with BRAM and external memory, to analyse execution cycles with different number of channels and unroll factors

Baseline
FPGA matrix multiplication accelerator with BRAM and 2 memory channels

Outcome
FPGA matmul accelerators exploiting increased unroll factors and external memory, with 2, 16 and 32 memory channels

Analysis results
External memory beneficial for large unroll factor: parallel load of many data offsets the additional cycles required for external memory





GCN accelerator

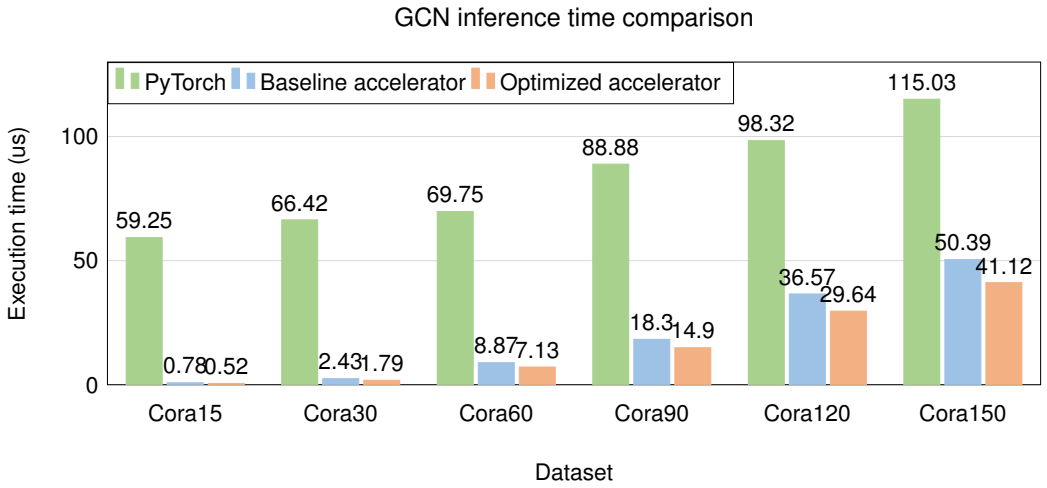
Experiment
Automated synthesis of the GCN model with the new optimization pipeline from Design Space Exploration.

Baseline
PyTorch CPU execution, 8 threads

Outcome
FPGA baseline and optimized accelerators; optimized accelerator uses 2 memory channels with BRAMs and one full unroll, and it achieves an average of more than 3x improvement in inference time.

Analysis results

- Accelerator outperforms PyTorch
- Performance improvement maintained with increasing dataset sizes
- Synthesis has not impacted the accuracy of the model



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	Proposed 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

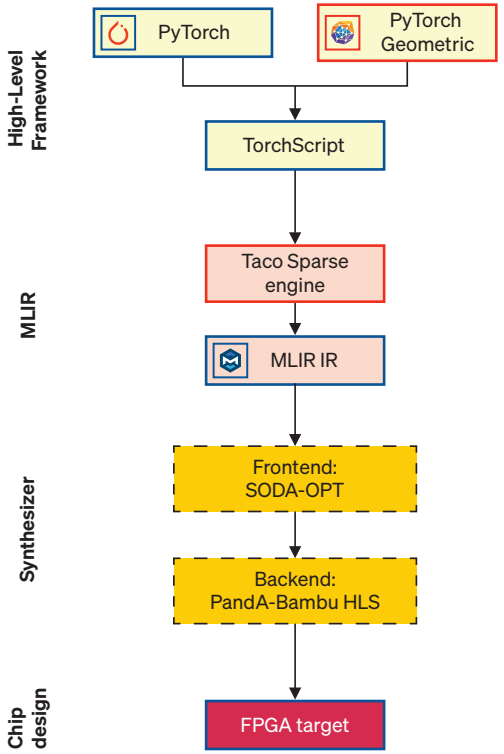
Conclusions and future developments

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 in red

POLITECNICO
MILANO 1863

Thanks for your attention