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

01 Introduction

- motivations
- objectives
- background

02 Solution

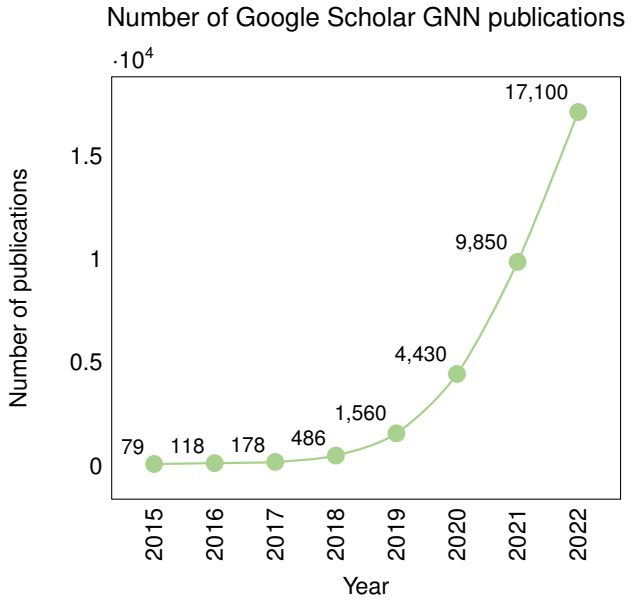
- toolchain
- experiments
  a. matrix mul acceleration
  b. GCN accelerator
  c. model accuracy

03 Conclusion

- solution analysis and comparison
- toolchain future developments

Thesis motivations and objectives

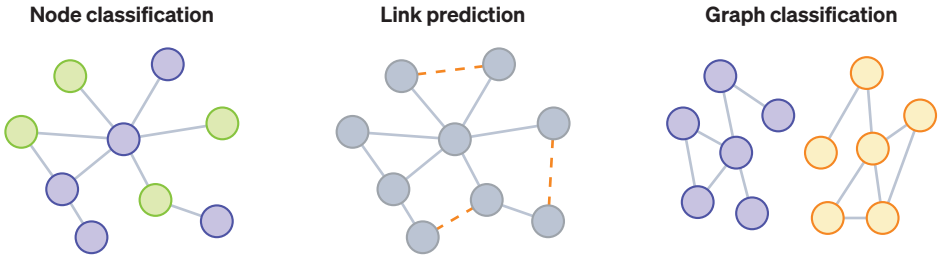
- A comprehensive toolchain for Graph Neural Network inference acceleration on FPGA architectures
- Identify customized optimizations designed to finely enhance model performance
- Enhancing the capabilities of the toolchain's elements to make them compatible
- Identify the models bottlenecks and accelerate the heaviest computational operations
- 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

Deep learning techniques that operate on graph-structured data to solve prediction tasks:



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

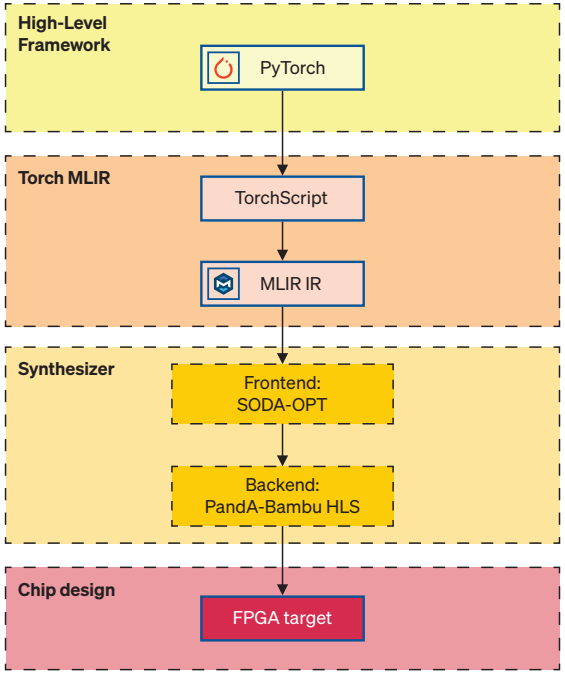
aggregates embeddings in a global feature vector

Design of the toolchain

PyTorch
Popular framework for neural network implementations.

Torch-MLIR
enables the generation of the MLIR IR.

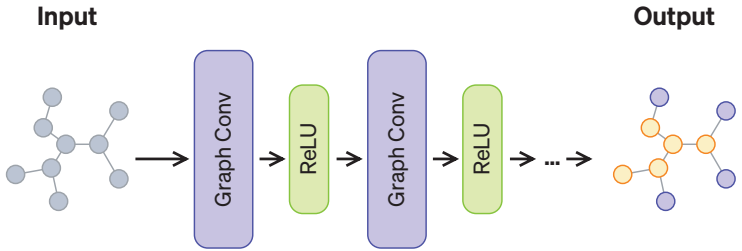
Synthesizer
The MLIR representation serves as input for the synthesizer. Frontend optimizations are applied, then the refined version proceeds to the backend, where the actual GNN accelerator is effectively produced.



FPGA Toolchain for Graph Neural Network Acceleration

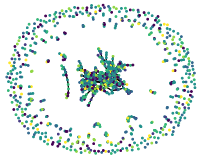
Toolchain inputs

GCN model



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

Cora dataset



- scientific publications
- multiclass classification

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

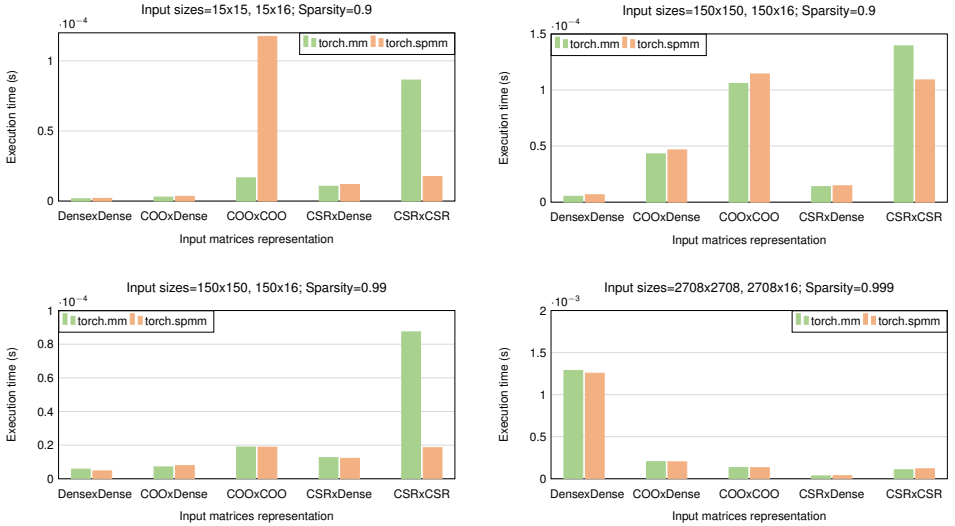
Model analysis

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



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

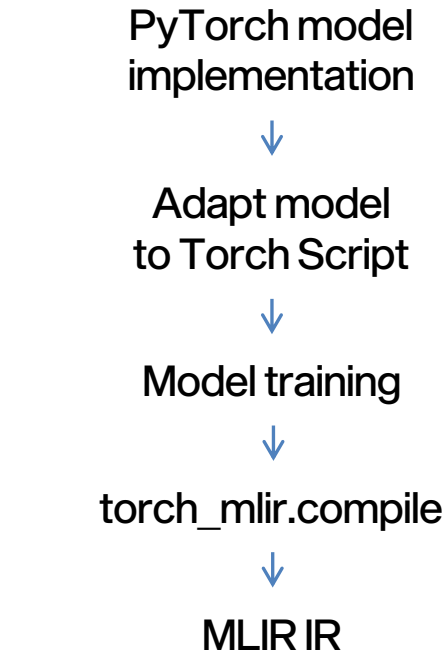
From PyTorch to MLIR

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

The model has been adapted to Torch Script, then it has been compiled using Torch-MLIR to obtain its MLIR representation.

MLIR

A novel approach to construct reusable and extensible compiler infrastructure.



From PyTorch model implementation to MLIR IR flow

Synthesis

SODA-OPT

MLIR representation
↓
Outline kernel function
↓
High level optimizations
↓
Low-Level IR

- high level optimization
- subset of MLIR optimization passes
- output represents the input of Panda-Bambu

PandA-Bambu

Low-Level IR
↓
Low level optimizations
↓
RTL IR
↓
Accelerator

- low level optimizations
- output is the final accelerator
- simplified hardware development
- reduces design time and effort

HLS

C/C++
↓
HLS tool
↓
VHDL/Verilog

FPGA-accelerated matrix multiplication

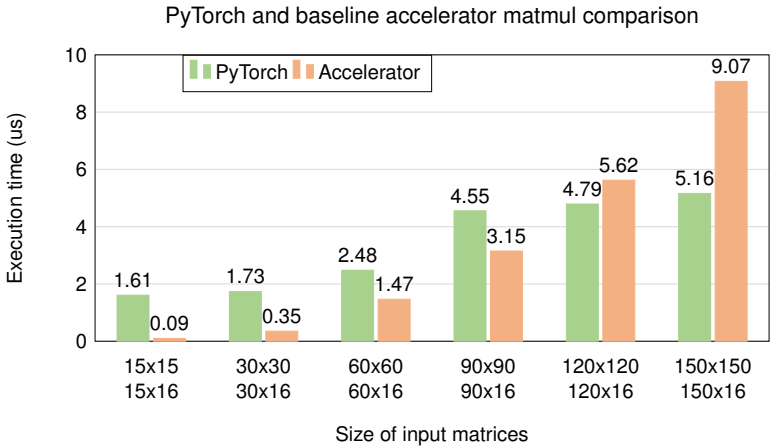
PyTorch
Intel Core i9, 8 cores, 2.3 GHz

Accelerators
AMD Virtex UltraScale+ (Alveo U280) FPGA.

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

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

Analysis about model performance to identify the computational bottlenecks



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

Synthesis optimizations

Loop unrolling

before

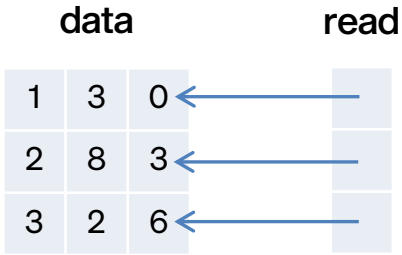
```
for i in range(0,4){
        code
}
```

after

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

- expanding completely or partially the loop
- unroll factor parameter to decide the number of loop iterations to unroll
- more chances of parallelization

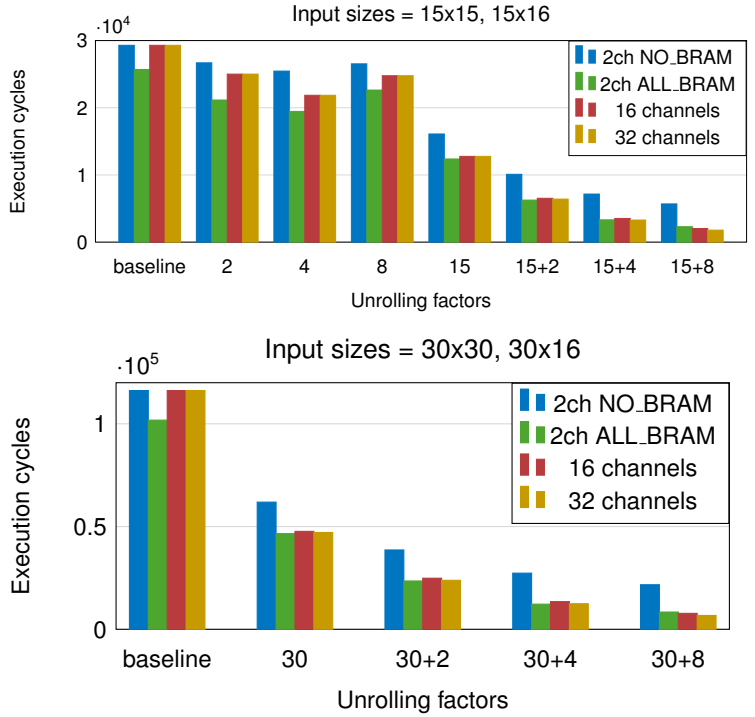
Parallel memory access



- increasing the number of memory channels
- better exploitation of available parallelism
- more parallel memory accesses but more cycles due to external memory

Optimized matrix multiplication accelerator

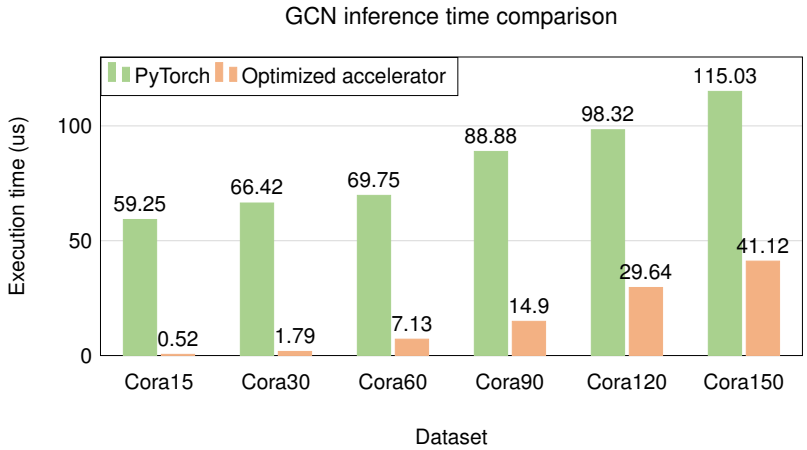
- Low available parallelization: two channels and storing all memory objects in BRAMs
- High available parallelization: thirty-two channels with external memory
- Two channels with external memory is the least favourable choice
- External memory beneficial when large volume of data can be simultaneously loaded and stored to offset the additional cycles required for external memory



Cycles comparison between configurations with different number of channels with BRAM/external memory.

GCN accelerator

- Accelerator outperforms PyTorch
- Advantage 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 two channels with on-chip BRAMs and one full unroll
- synthesis has not impacted the accuracy of the model

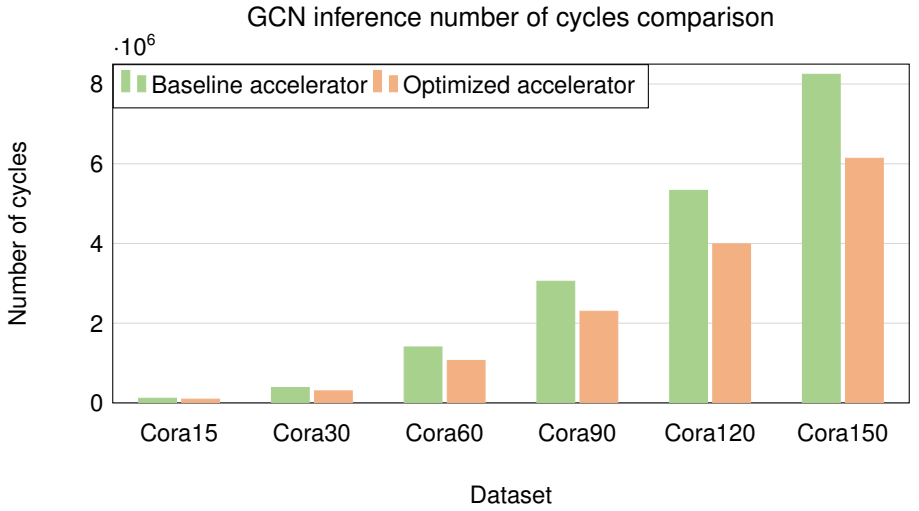
Dataset	PyTorch	Accelerator	ϵ	$\bar{\epsilon}$
Cora15	0.333	0.333	1.31E-06	0.26E-06
Cora30	0.333	0.333	8.64E-06	0.54E-06
Cora60	0.166	0.166	4.02E-06	0.25E-06
Cora90	0.555	0.555	19.47E-06	0.45E-06
Cora120	0.291	0.291	19.86E-06	0.27E-06
Cora150	0.533	0.533	23.84E-06	0.30E-06

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

GCN accelerators comparison

analysis to show how the accelerator has been affected by the loop unrolling technique with respect to the baseline performance

- optimized accelerator uses two channels and one full unrolling of the innermost loop
- the speedup increases as the size of the dataset increases
- trade-off between performance and area



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

Proposed solution analysis

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	All 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

About FlowGNN

- FlowGNN is the only state of the art solution using High-Level Synthesis
- New model and features should be implemented in C++, for significative changes it can represent a possible blocking factor
- Limited possibility of customization to specific needs, only four parallelization parameters

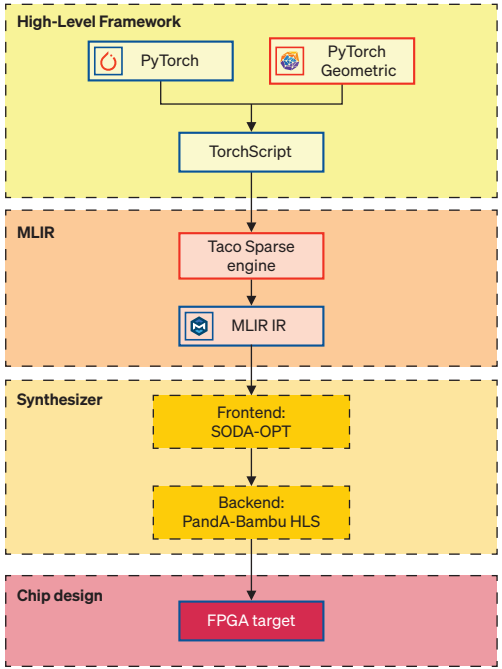
Conclusions and future developments

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