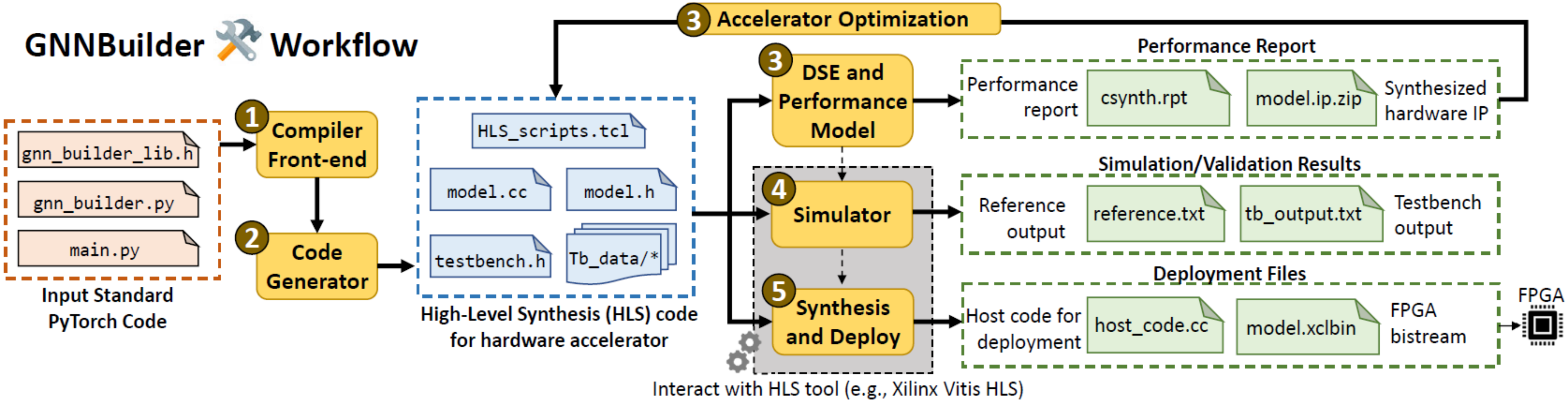
GNNBuilder

An Automated Framework for Generic Graph Neural Network Accelerator Generation, Simulation, and Optimization



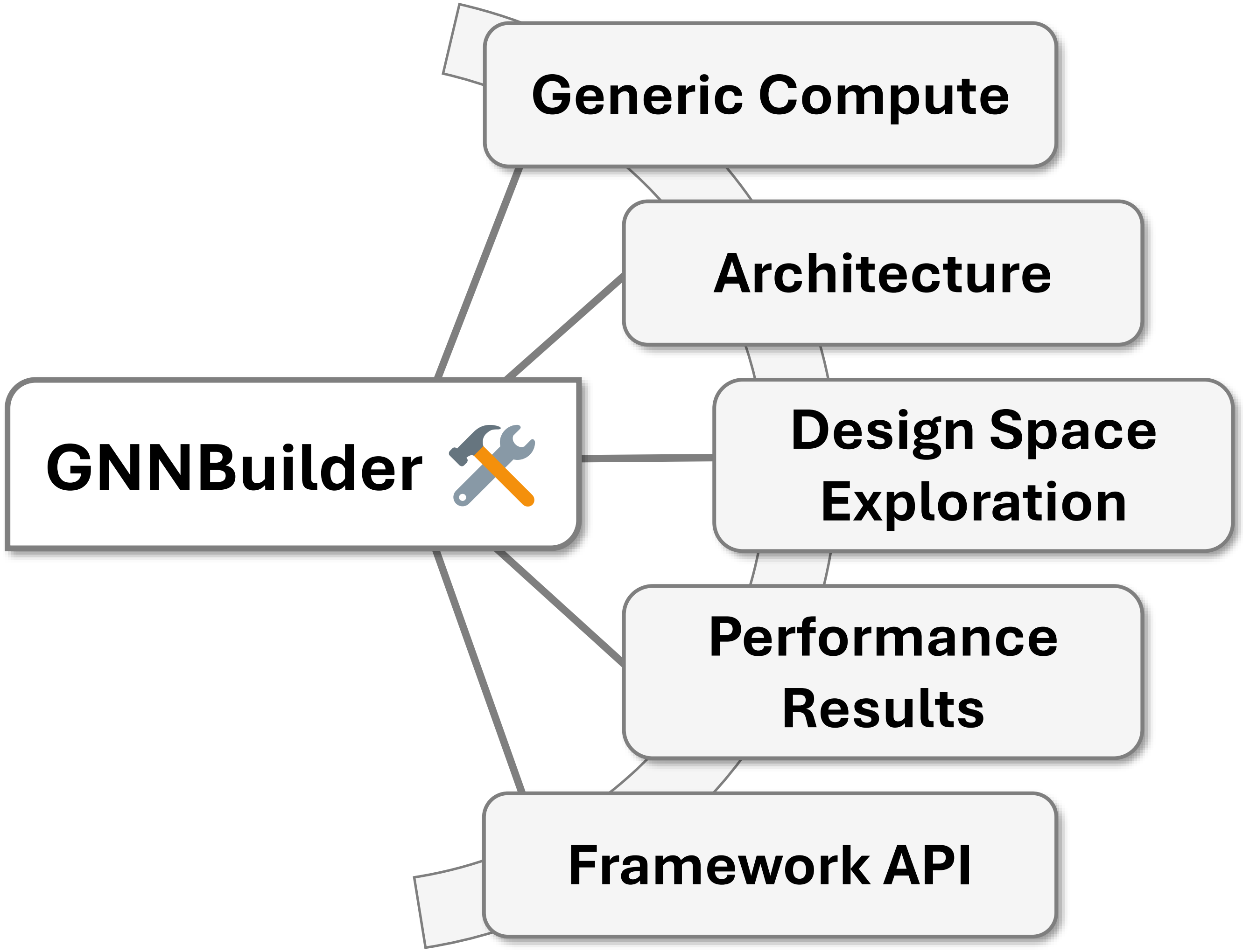
Stefan Abi-Karam[1,2], Cong Hao[1]

[1]Georgia Institute of Technology, [2]Georgia Tech Research Institute

stefanabikaram@gatech.edu, callie.hao@ece.gatech.edu

Sharc Lab @ Georgia Tech

sharclab.ece.gatech.edu



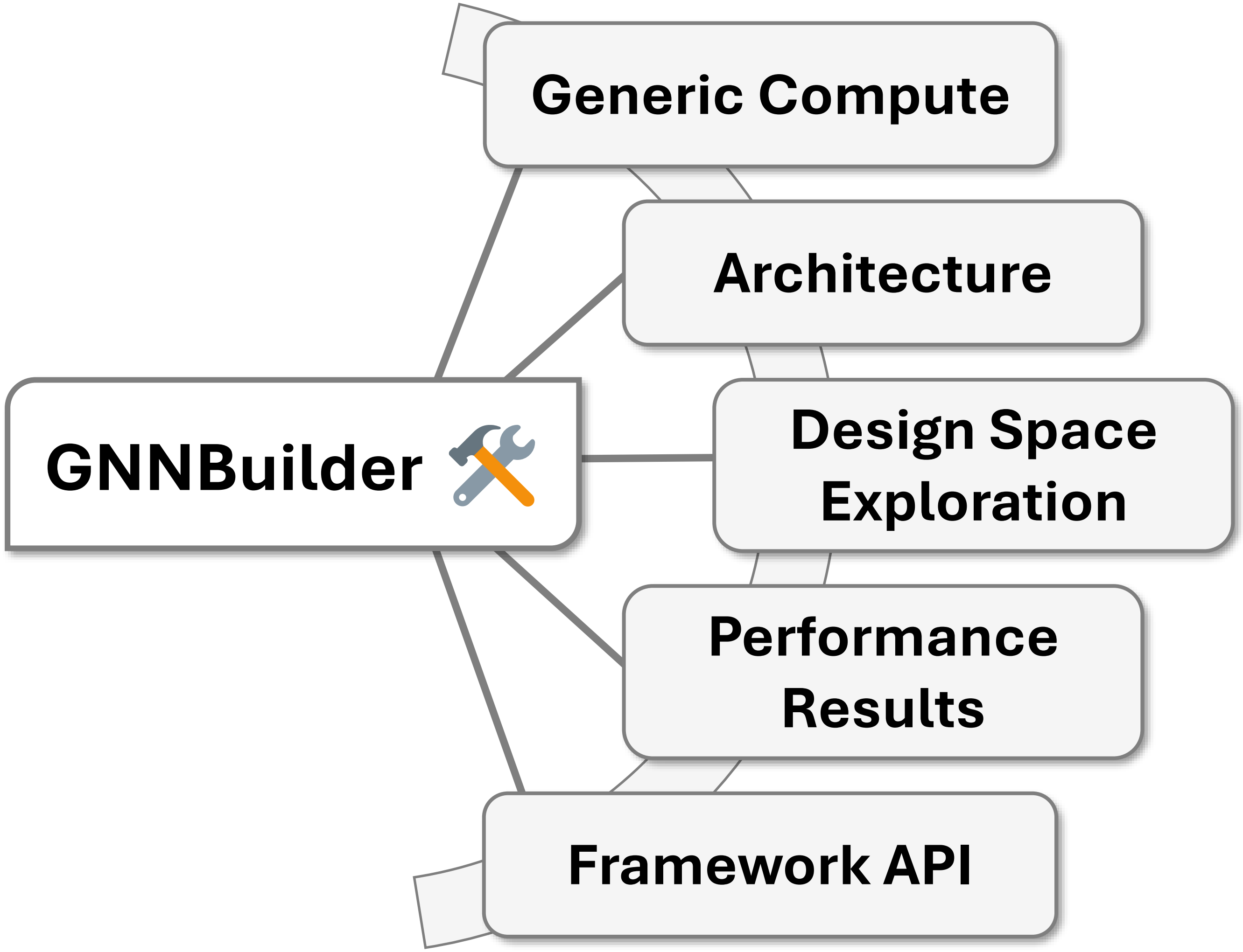
Background
- DL for Graphs
- GNNs
- Accelerators

GNNBuilder
- Generic Compute
- Architecture
- Design Space Exploration
- Performance Results
- Framework API

Limitations

Ongoing Work

Documentation and Open Source



Background

DL for Graphs

GNNs

Accelerators

GNNBuilder

Generic Compute

Architecture

Design Space Exploration

Performance Results

Framework API

Limitations

Ongoing Work

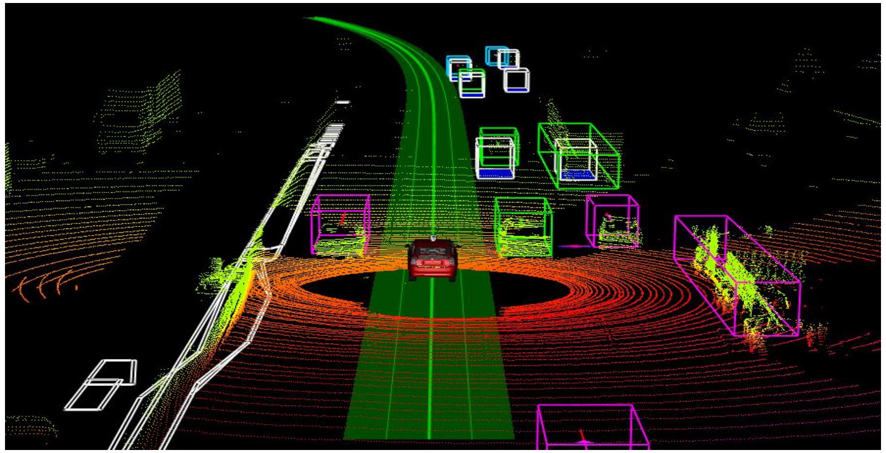
Documentation and Open Source

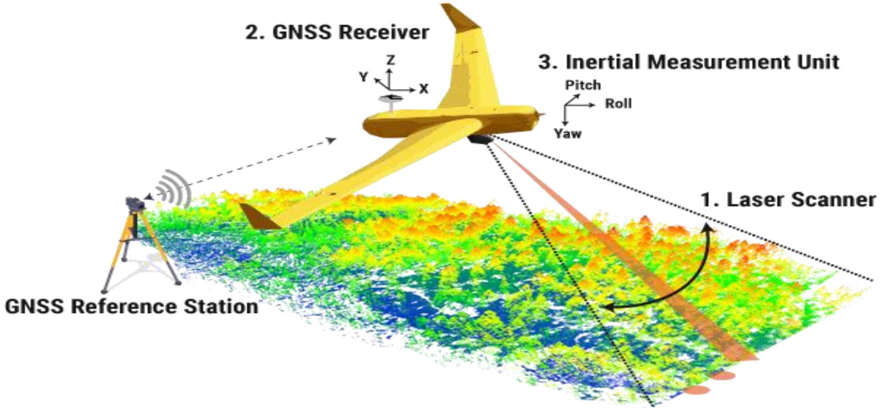
Deep Learning for Graphs

What "Deep Learning for Graphs" applications **benefit most from hardware acceleration?**

Deep Learning for Graphs

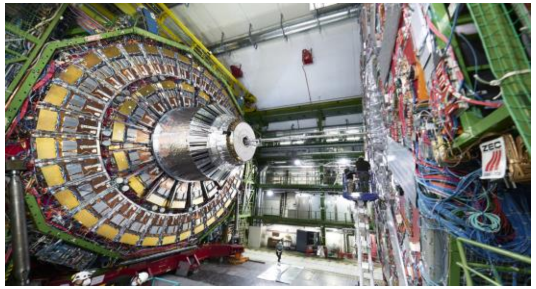
Point Cloud Processing



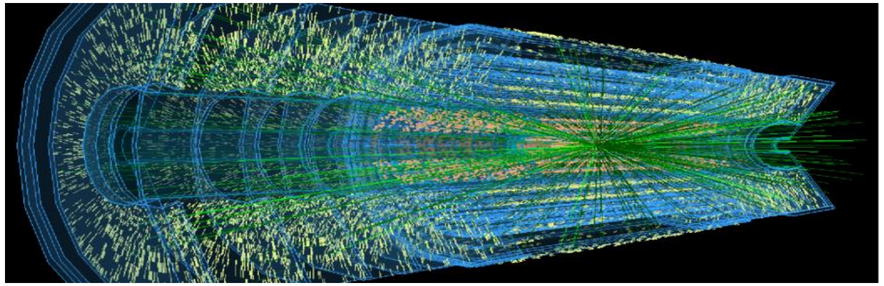


High Energy Physics!

CERN

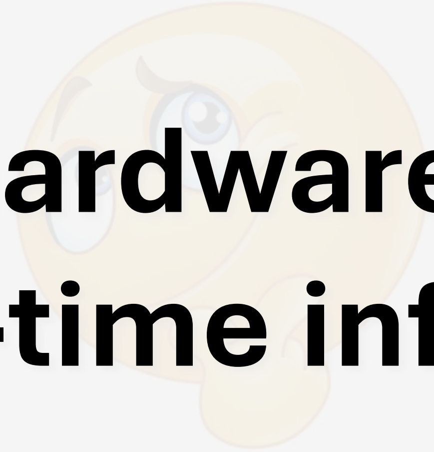


Fermilab



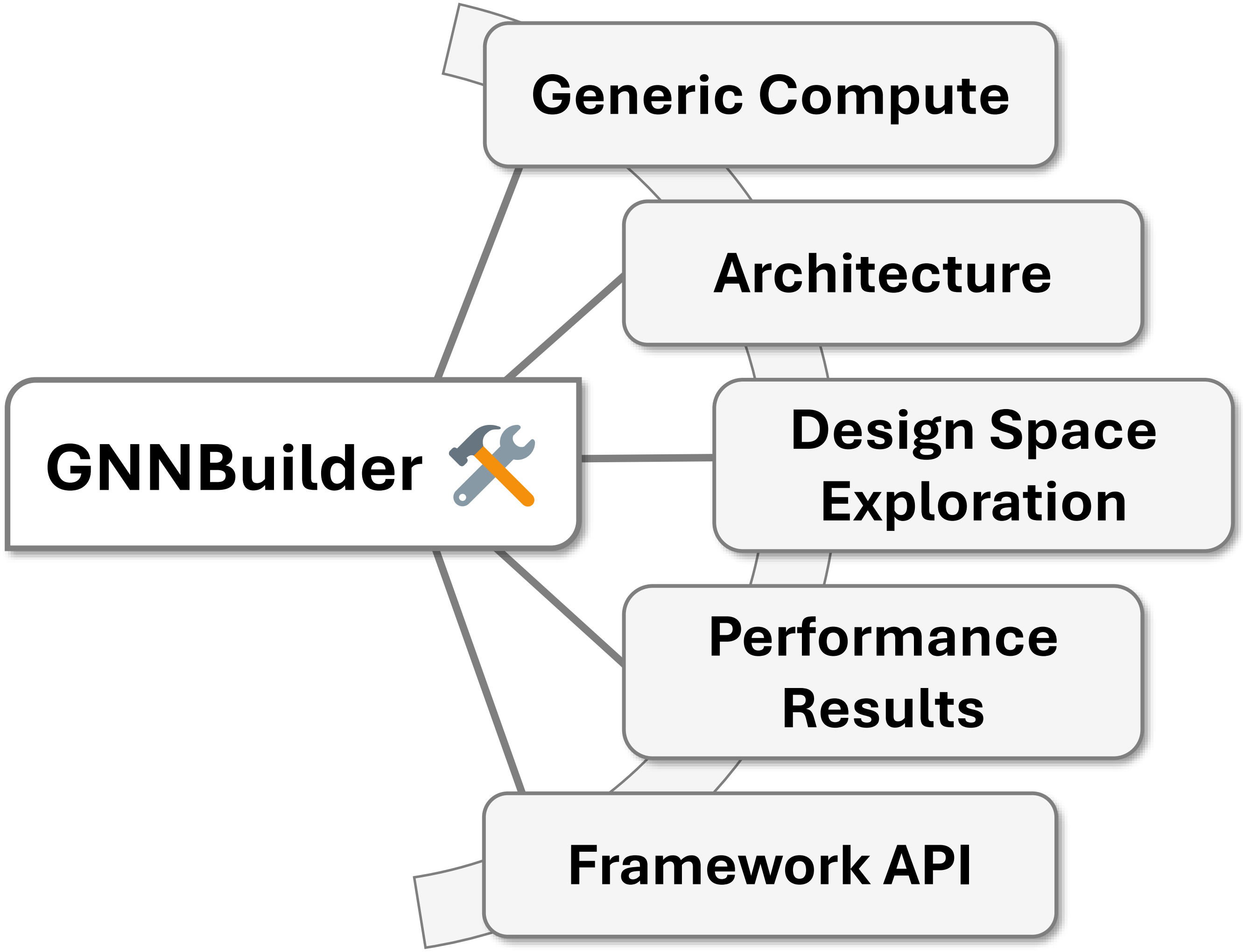
Deep Learning for Graphs

Point Cloud Processing



High Energy Physics!

Benefit from **hardware acceleration**
for **real-time inference**!



Background

DL for Graphs

GNNs

Accelerators

GNNBuilder

Generic Compute

Architecture

Design Space Exploration

Performance Results

Framework API

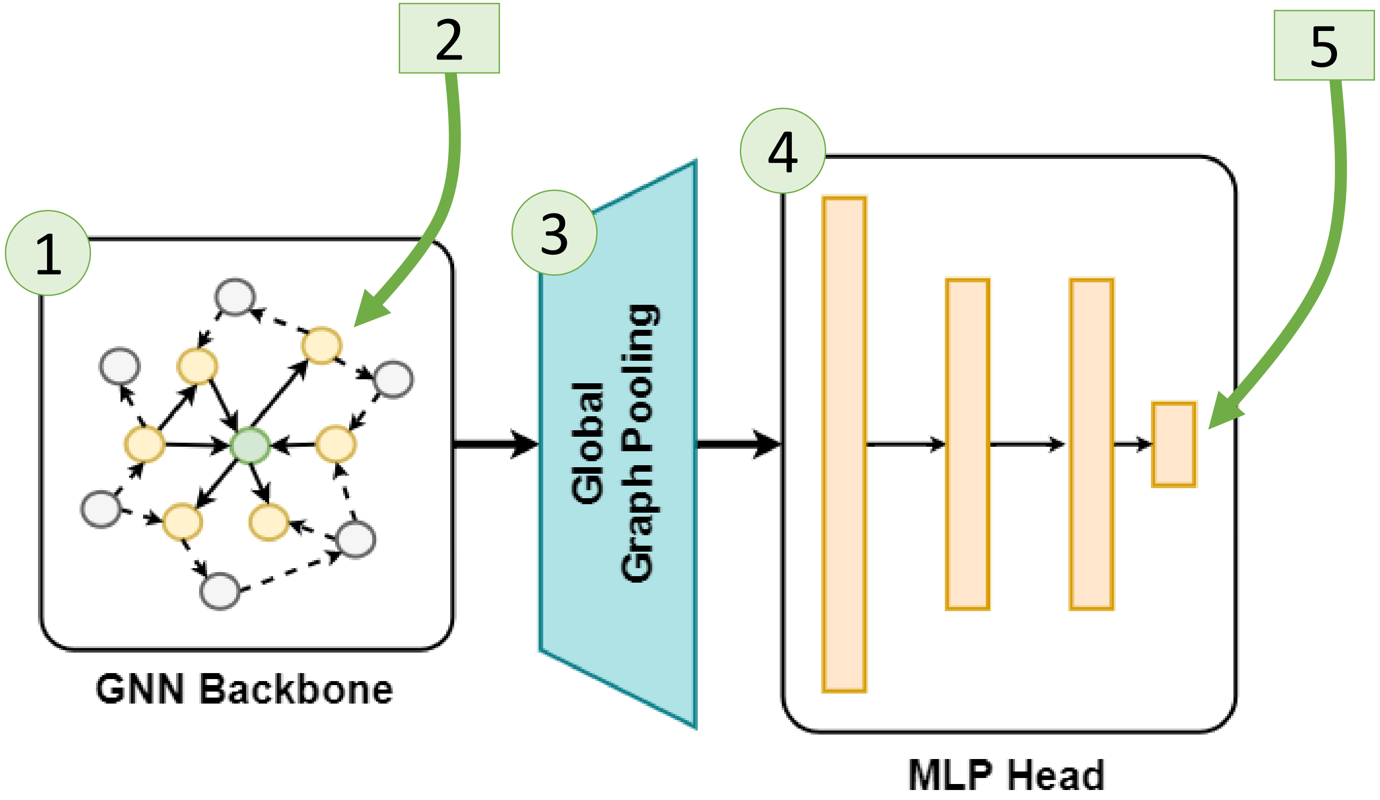
Limitations

Ongoing Work

Documentation and Open Source

Graph Neural Networks (GNNs)

1. Graph Convolutions
2. Node-Level Embedding *
3. Pool Nodes **
4. Graph Prediction Head
5. Graph-Level Embedding

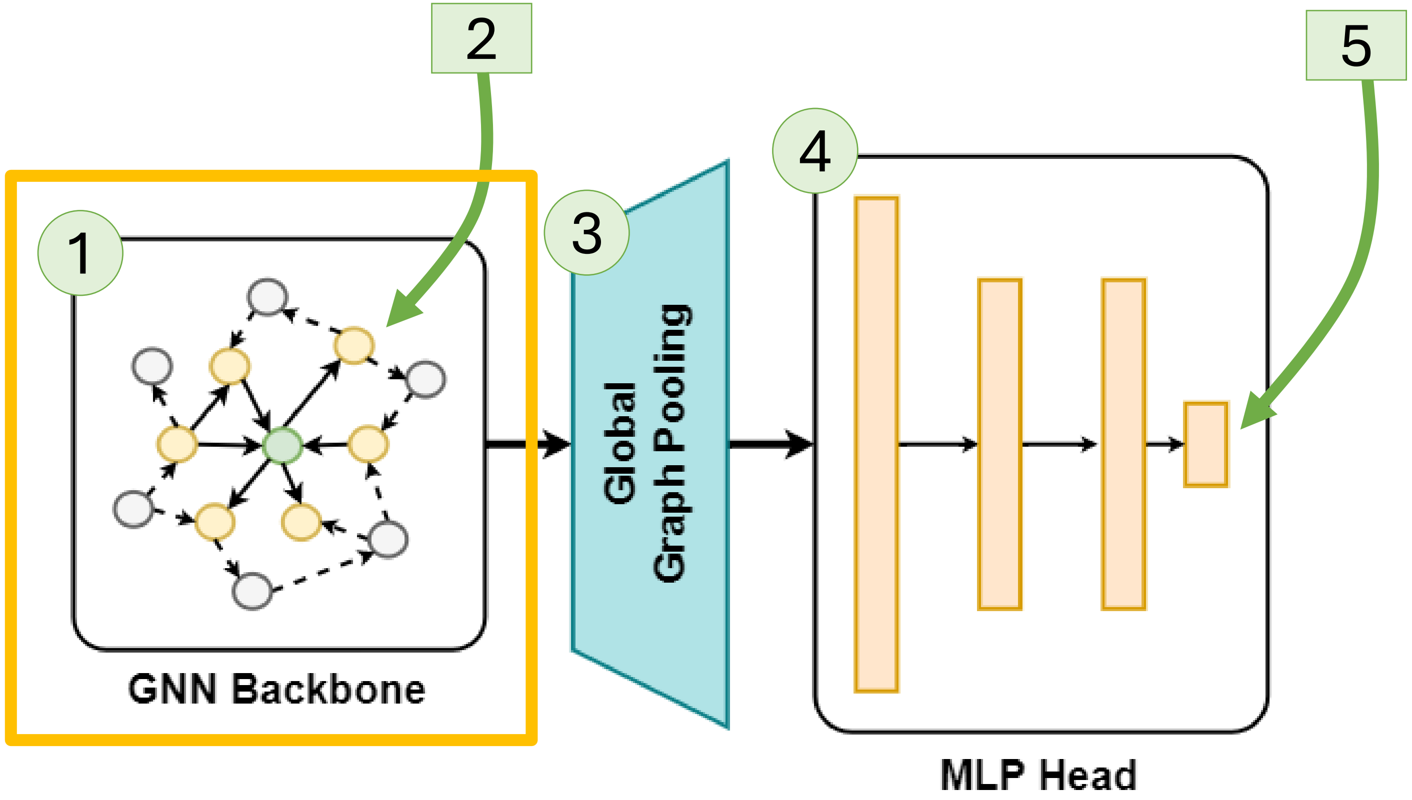


* Or compute edge embeddings

** Stop here for node-level and edge-level tasks

Graph Neural Networks (GNNs)

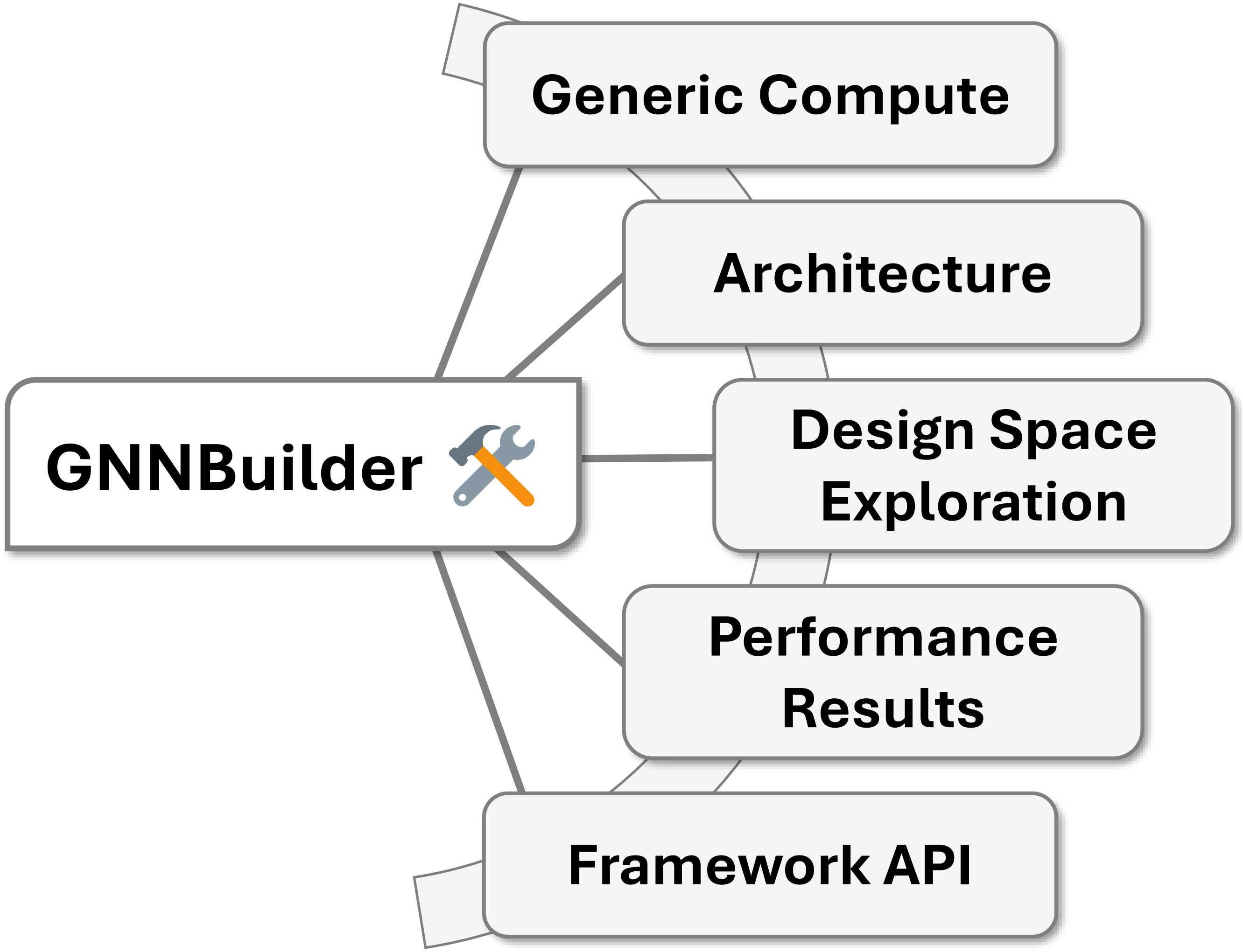
1. Graph Convolutions
2. Node-Level Embedding *
3. Pool Nodes **
4. Graph Prediction Head
5. Graph-Level Embedding



* Or compute edge embeddings

** Stop here for node-level and edge-level tasks

This part is the challenge!



Background
DL for Graphs
GNNs
Accelerators

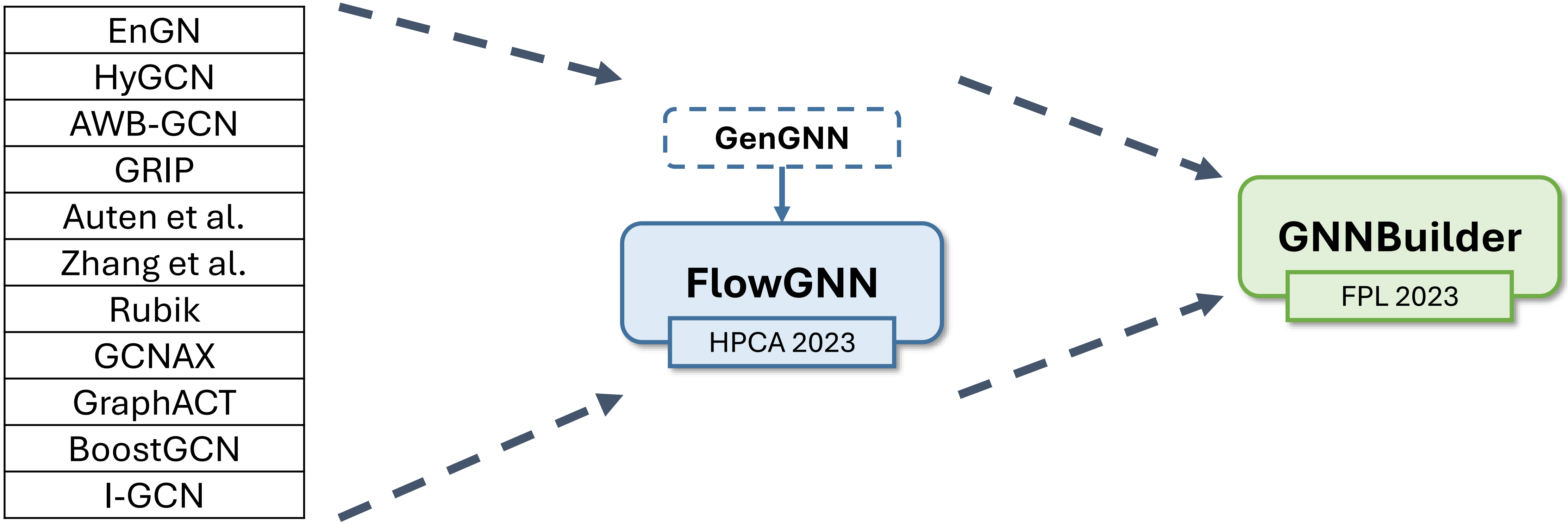
GNNBuilder
Generic Compute
Architecture
Design Space Exploration
Performance Results
Framework API

Limitations

Ongoing Work

Documentation and Open Source

GNN Hardware Acceleration Landscape

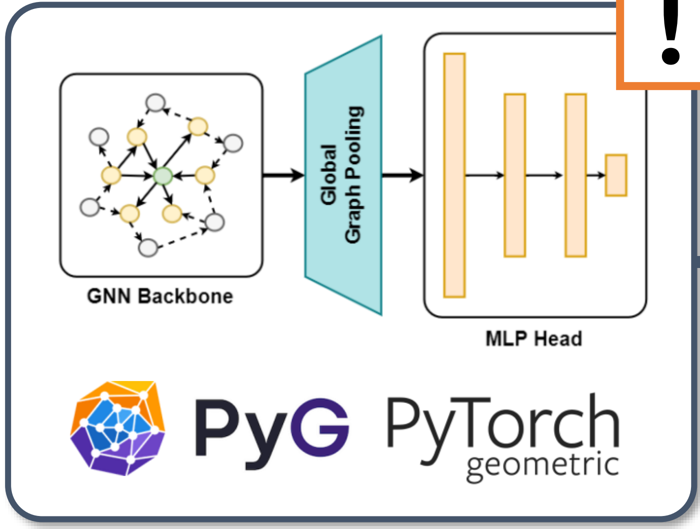


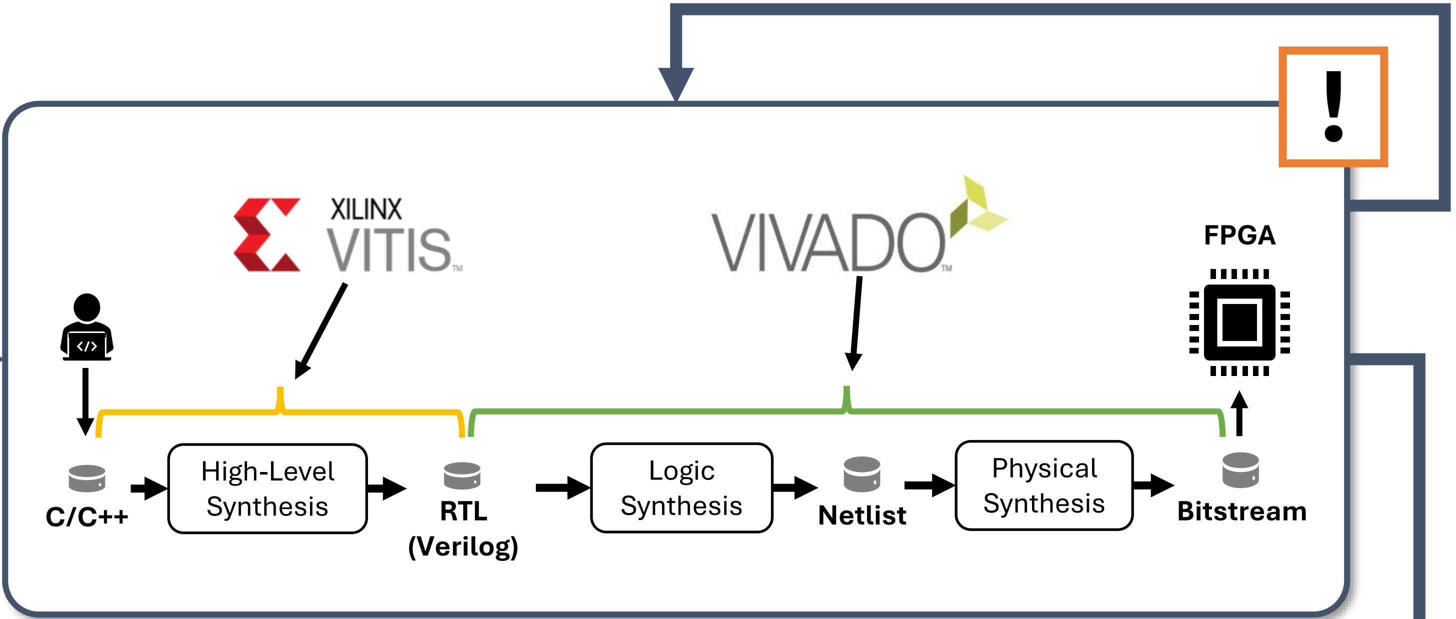
- ☑ ***Hardware Acceleration***
- ✗ General Computation
- ✗ End-To-End Automation

- ☑ Hardware Acceleration
- ☑ ***General Computation***
- ✗ End-To-End Automation

- ☑ Hardware Acceleration
- ☑ General GNN Computation
- ☑ ***End-To-End Automation***

Design Workflow





Software Models are **Always Changing**

Hardware Design is **Complex, Slow**, and **Requires Expert Knowledge**

Platforms are **Varied** and **Heterogeneous**

GNNBuilder Overview

Generic Compute: wide range of GNN model support

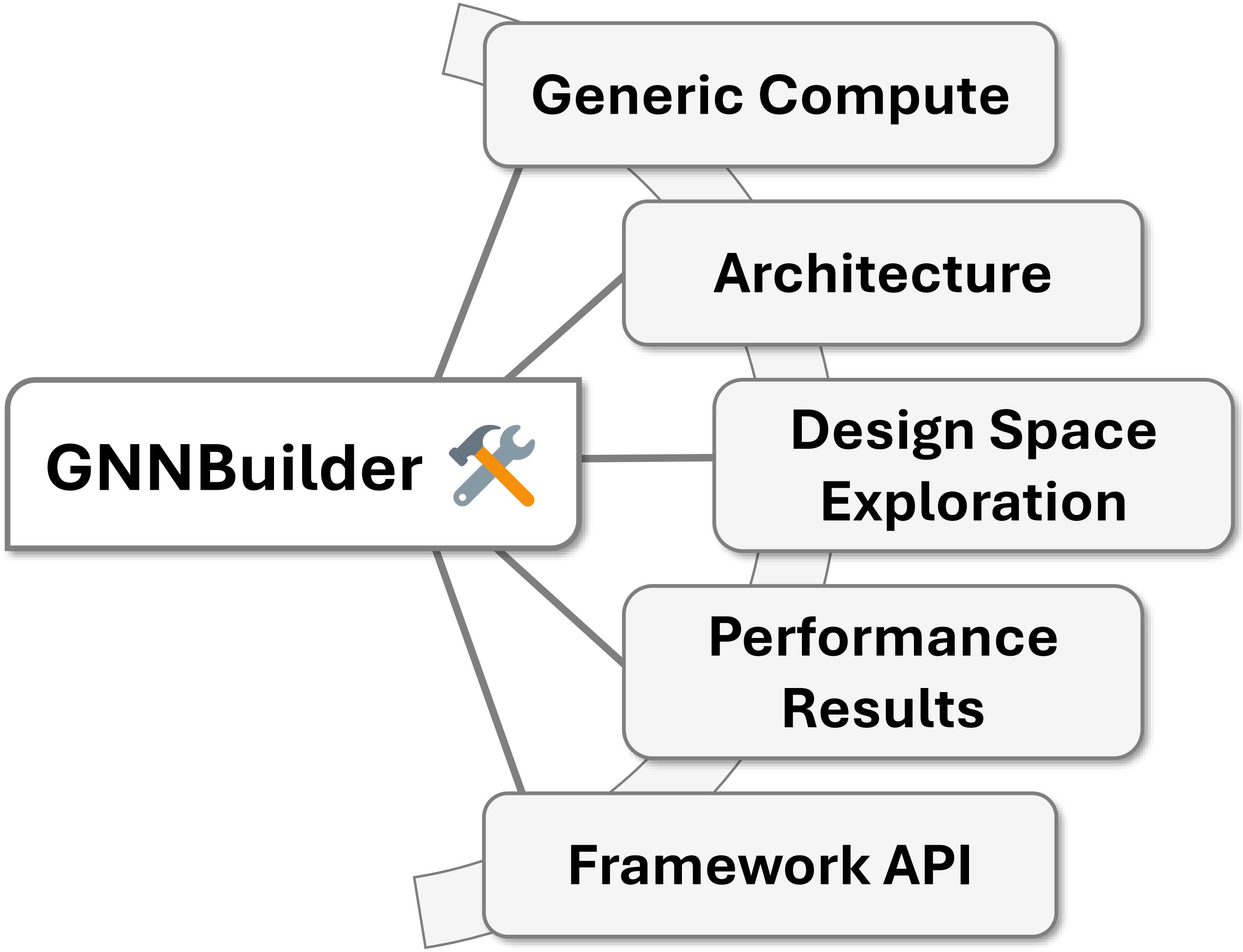
Architecture: Parameterizable SW and HW model architecture

Design Space Exploration: ML-based DSE models

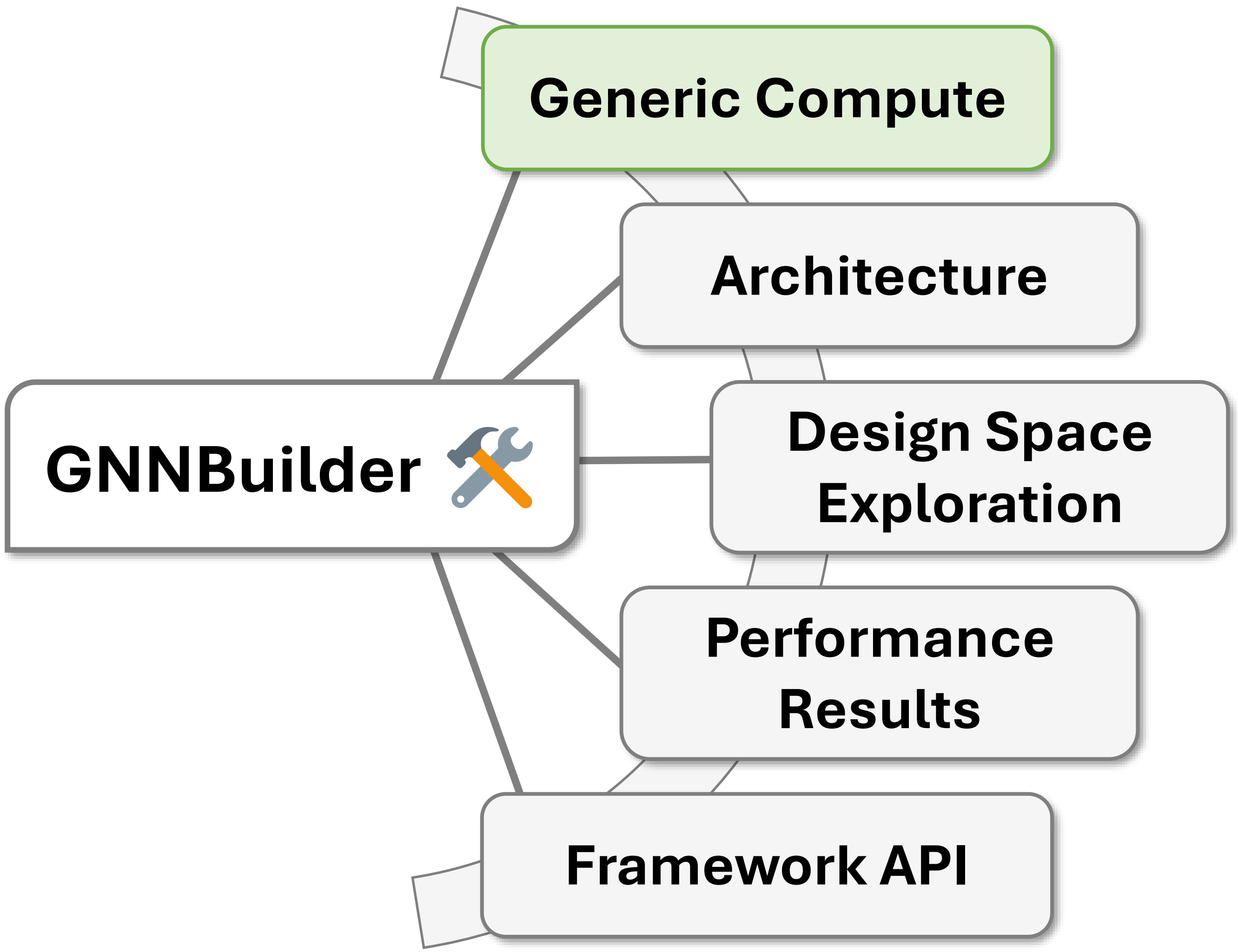
Performance Results: Better performance against CPU and GPU

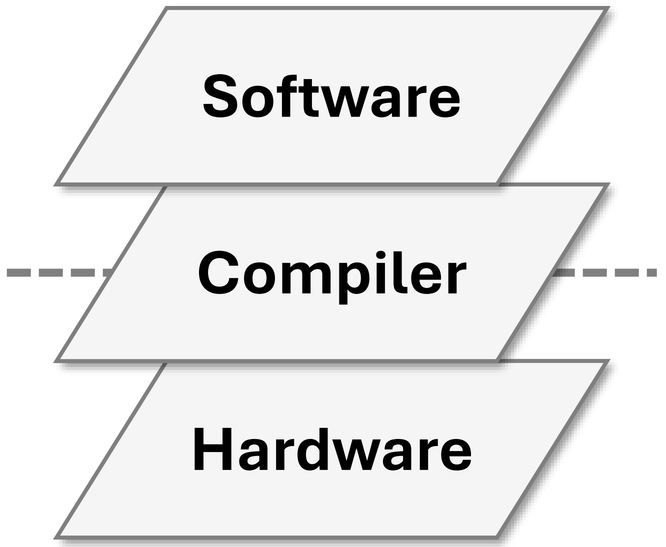
Framework API: Open-source Python API with end-to-end workflow

Extensibility: Interoperability with PyTorch and extendable by anyone



- Background
  - DL for Graphs
  - GNNs
  - Accelerators
- **GNNBuilder**
  - Generic Compute
  - Architecture
  - Design Space Exploration
  - Performance Results
  - Framework API
- Limitations
- Ongoing Work
- Documentation and Open Source


GNNBuilder
Generic Compute
Architecture
Design Space Exploration
Performance Results
Framework API


Software
Compiler
Hardware

"SpMM is All You Need"

Sum-Multiply Reductions

GCN + GIN+ GraphSAGE

"SpMM is **Not** All You Need"

Non-Sum Aggregation

PNA + GraphSAGE

Anisotropic Message Passing

Graph Attention Networks

Edge Embeddings

Almost all other cases...

"SpMM is All You Need"

EnGN
HyGCN
AWB-GCN
GRIP
Auten et al.
Zhang et al.
Rubik
GCNAX
GraphACT
BoostGCN
I-GCN

☑ ***Hardware Acceleration***
☒ General Computation

"SpMM is **Not** All You Need"

FlowGNN

GNNBuilder

☑ Hardware Acceleration
☑ ***General Computation***

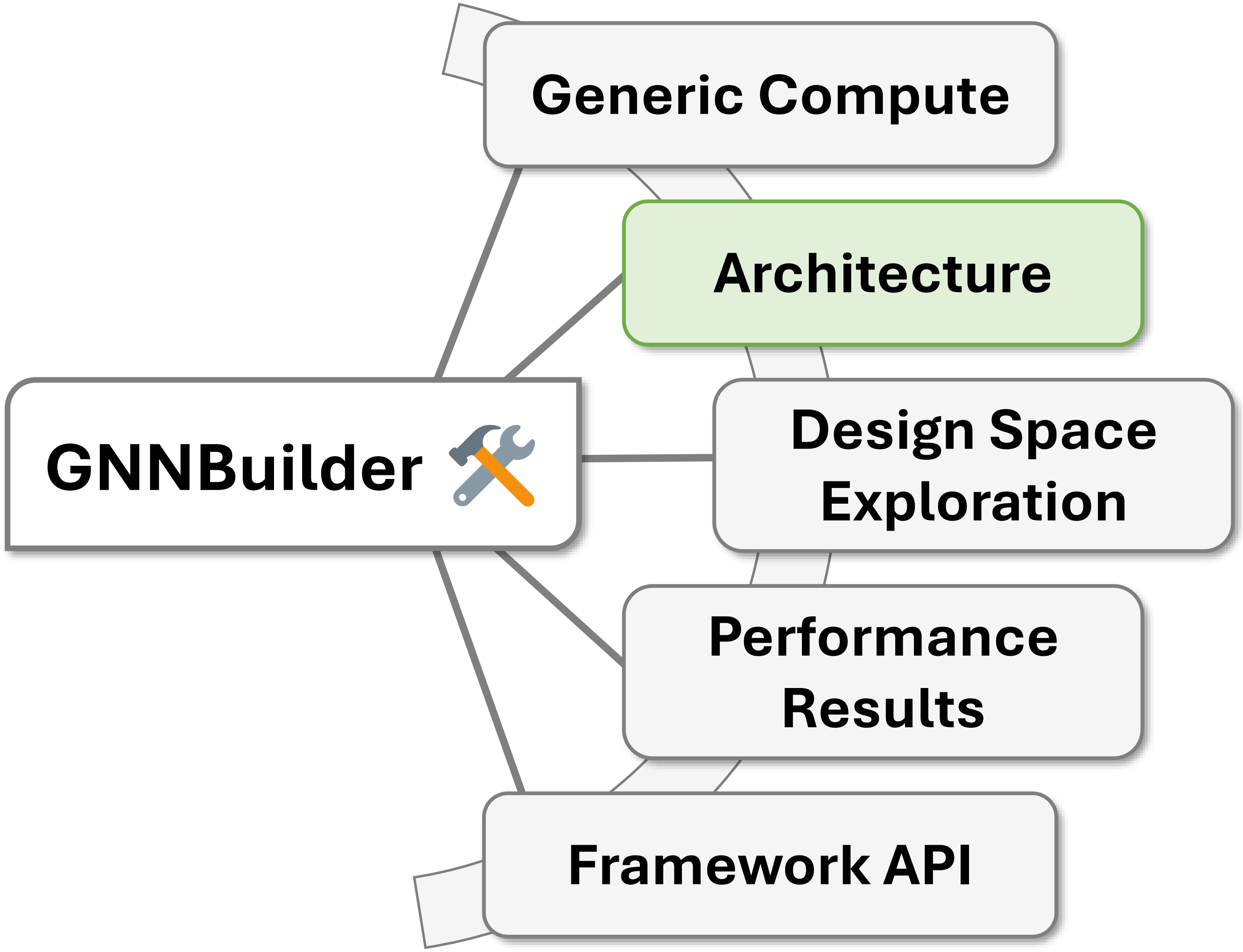
Generic Compute

Model	Representativeness
GCN [22]	GNN family that can be represented as sparse matrix-matrix multiplications (SpMM)
GraphSage [19]	GNN family with flexible / non-sum aggregation methods
GIN [42]	GNN family with *edge embeddings*, SpMM *does not* apply
PNA [9]	A popular Anisotropic GNN family arbitrarily using multiple aggregation methods and sophisticated message function, SpMM *does not* apply

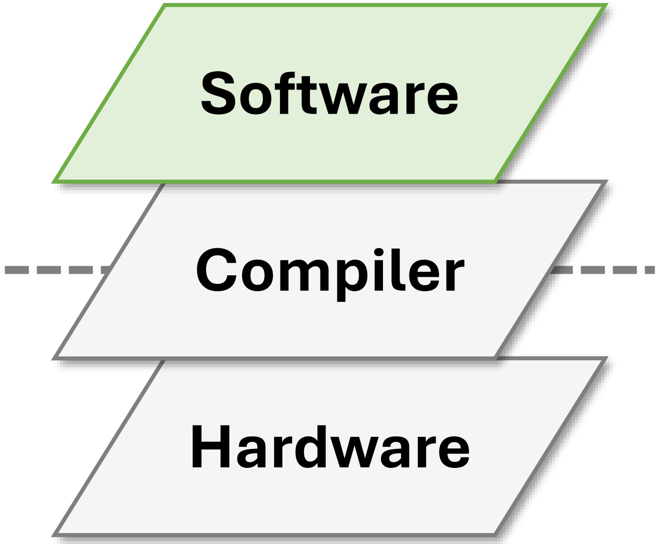
GCN: graph convolutional network; **GIN:** graph isomorphism network; **GraphSAGE:** graph sample and aggregate; **PNA:** principal neighbourhood aggregation.

Generic Compute

	HP-GNN [28]	DeepBurning-GL [25]	GNNBuilder
Acceleration Goal	Training	Inference	Inference
Programming Language	Self-defined	PyTorch and DGL	PyTorch
Anisotropic GNN Family	No	No	Yes
Extensibility	Low	Low	Very High
Arbitrary Quantization	No	No	Yes
Arbitrary Aggregation	No	No	Yes
Arbitrary Activation	Fixed	Fixed	Arbitrary
Skip Connections	No	No	Yes
Arbitrary Global Pooling	No	No	Yes
Arbitrary MLP Head	No	No	Yes
Fixed / Floaing Point Testbench	No	No	Yes
Open Source	No	No	Yes



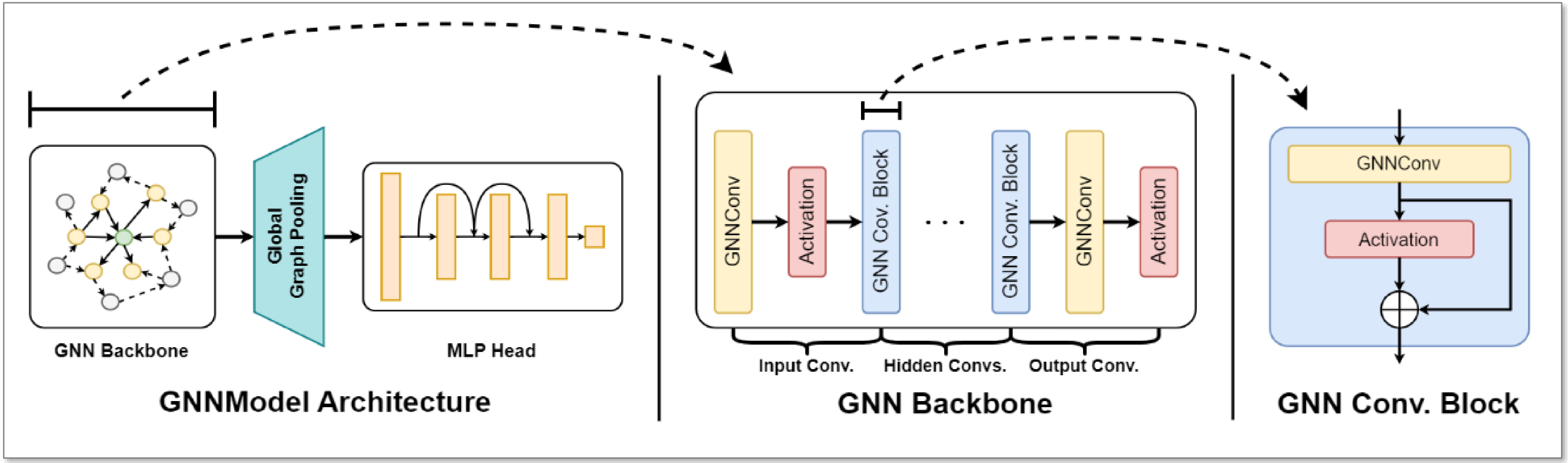
Generic Compute
Architecture
GNNBuilder
Design Space Exploration
Performance Results
Framework API



Software
Compiler
Hardware

Model Architecture

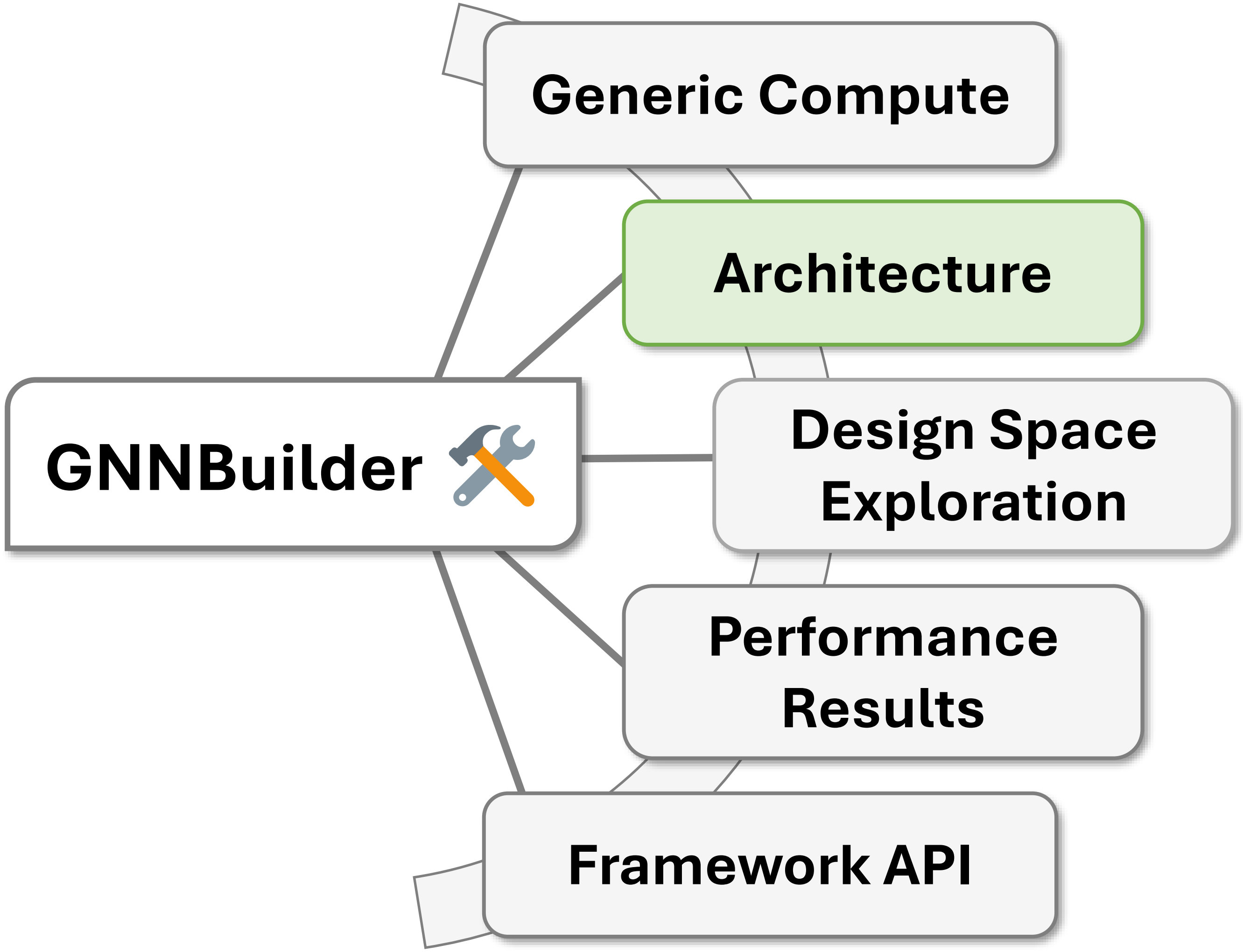
Support a parameterized model architecture for **node-level** and **graph-level tasks**.

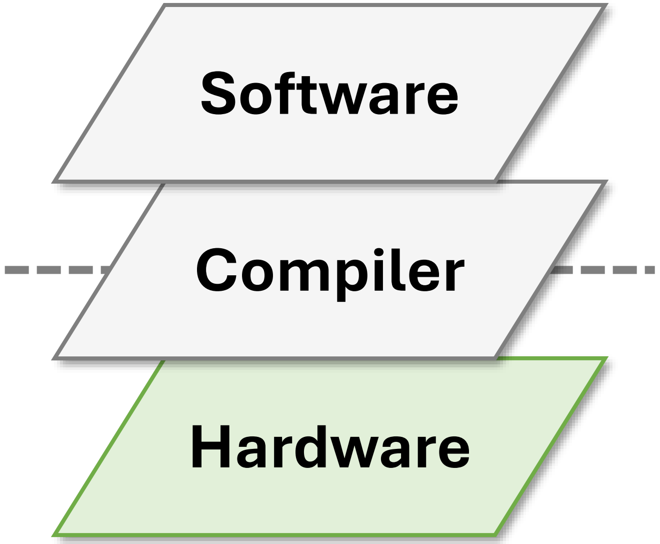


Model Architecture

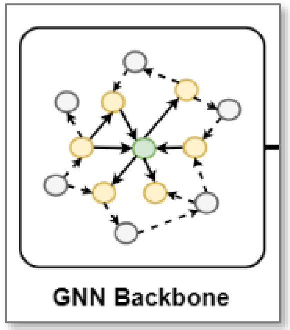
Fully parameterized GNN model using PyTorch Geometric.

```
model = gnnb.GNNModel(
    graph_input_feature_dim=dataset.num_features,
    graph_input_edge_dim=dataset.num_edge_features,
    gnn_hidden_dim=16,
    gnn_num_layers=2,
    gnn_output_dim=8,
    gnn_conv=gnnb.SAGEConv_GNNB,
    gnn_activation=nn.ReLU,
    gnn_skip_connection=True,
    global_pooling=gnnb.GlobalPooling(["add", "mean", "max"]),
    mlp_head=gnnb.MLP(
        in_dim=8 * 3,
        out_dim=dataset.num_classes,
        hidden_dim=8,
        hidden_layers=3,
        activation=nn.ReLU,
        p_in=8,
        p_hidden=4,
        p_out=1,
    ),
    output_activation=None,
    gnn_p_in=1,
    gnn_p_hidden=8,
    gnn_p_out=4,
)
```

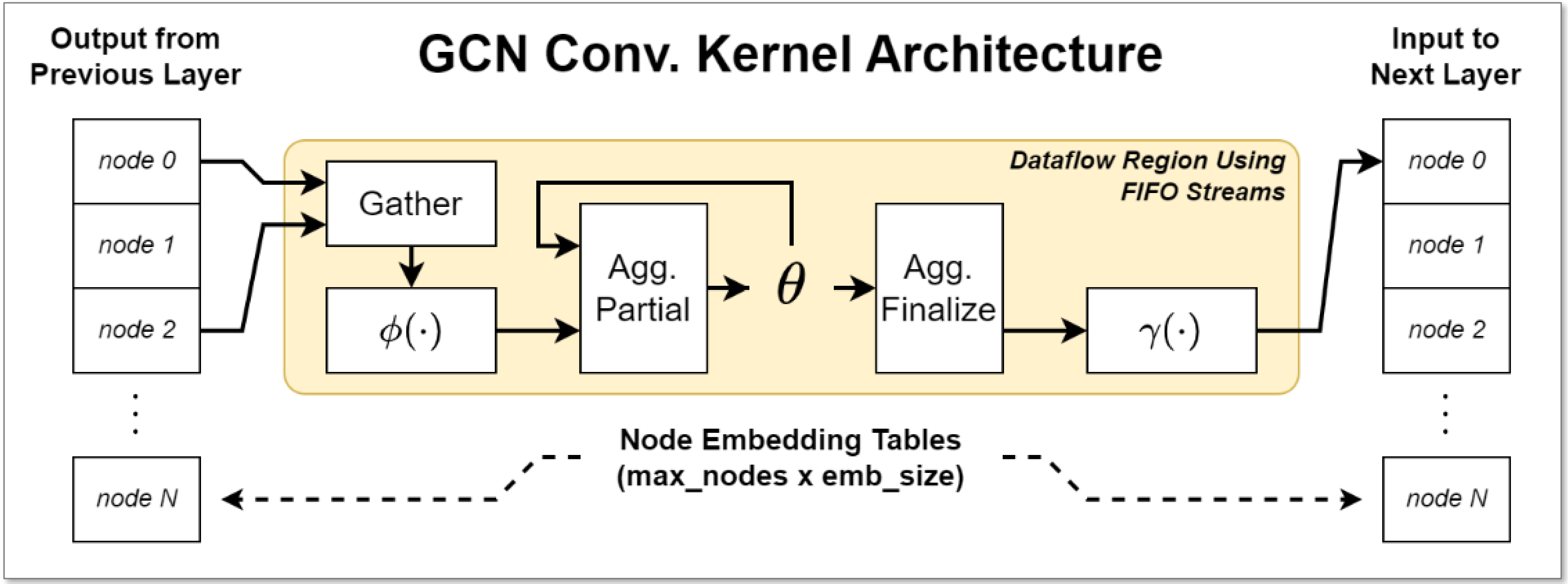

GNNBuilder
Generic Compute
Architecture
Design Space Exploration
Performance Results
Framework API


Software
Compiler
Hardware

Hardware Architecture

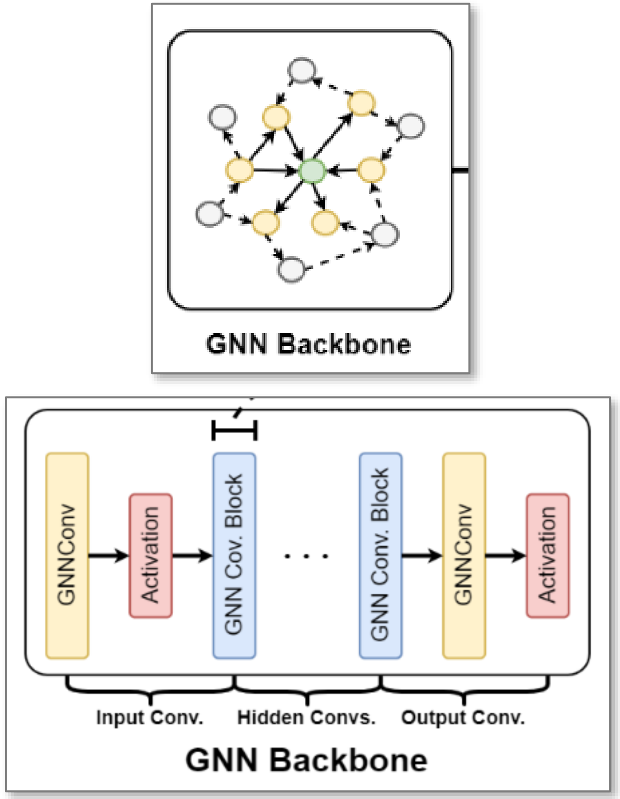


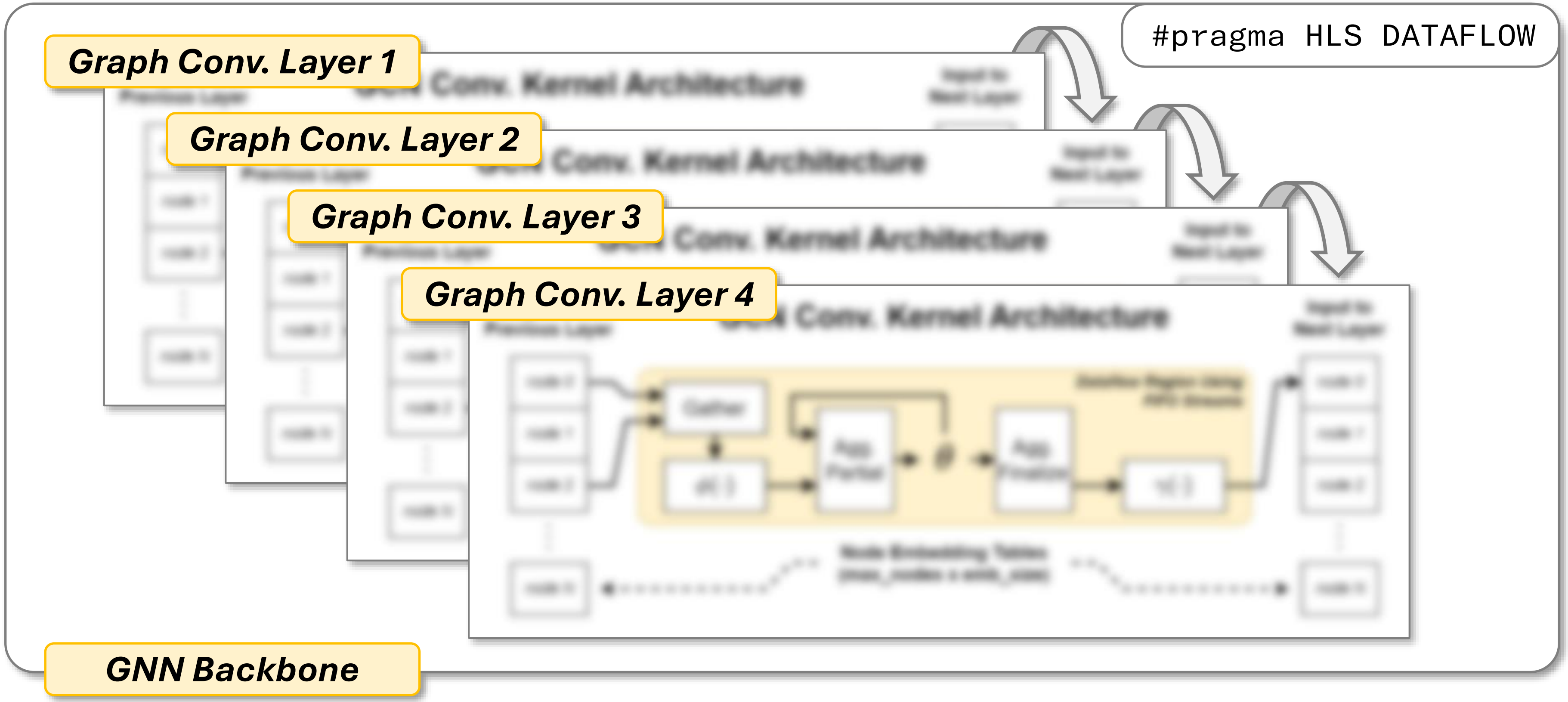
We use an **explicit message passing** hardware kernel architecture for each graph convolution.



Hardware Architecture

The entire GNN Backbone is an "HLS DATAFLOW" region allowing for higher data throughput between layers without stalling.



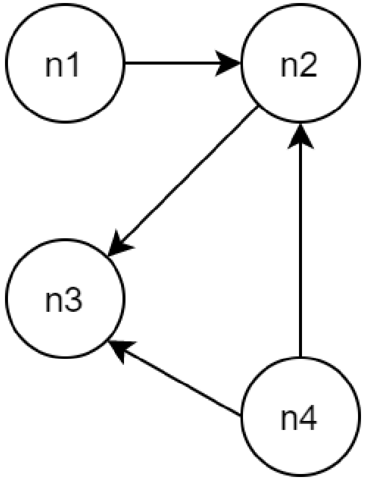


Graph Representation in Memory

We support COO (edge list) format for graph input.

We process the graph to create a degree table and neighbor table.

All this processing is done once on the device during inference.



COO Edge List

From:	1	2	4	4
To:	2	3	3	2

Compute

Degree Tables

	n1	n2	n3	n4
In-Degree:	0	2	2	0
Out-Degree:	1	1	0	2

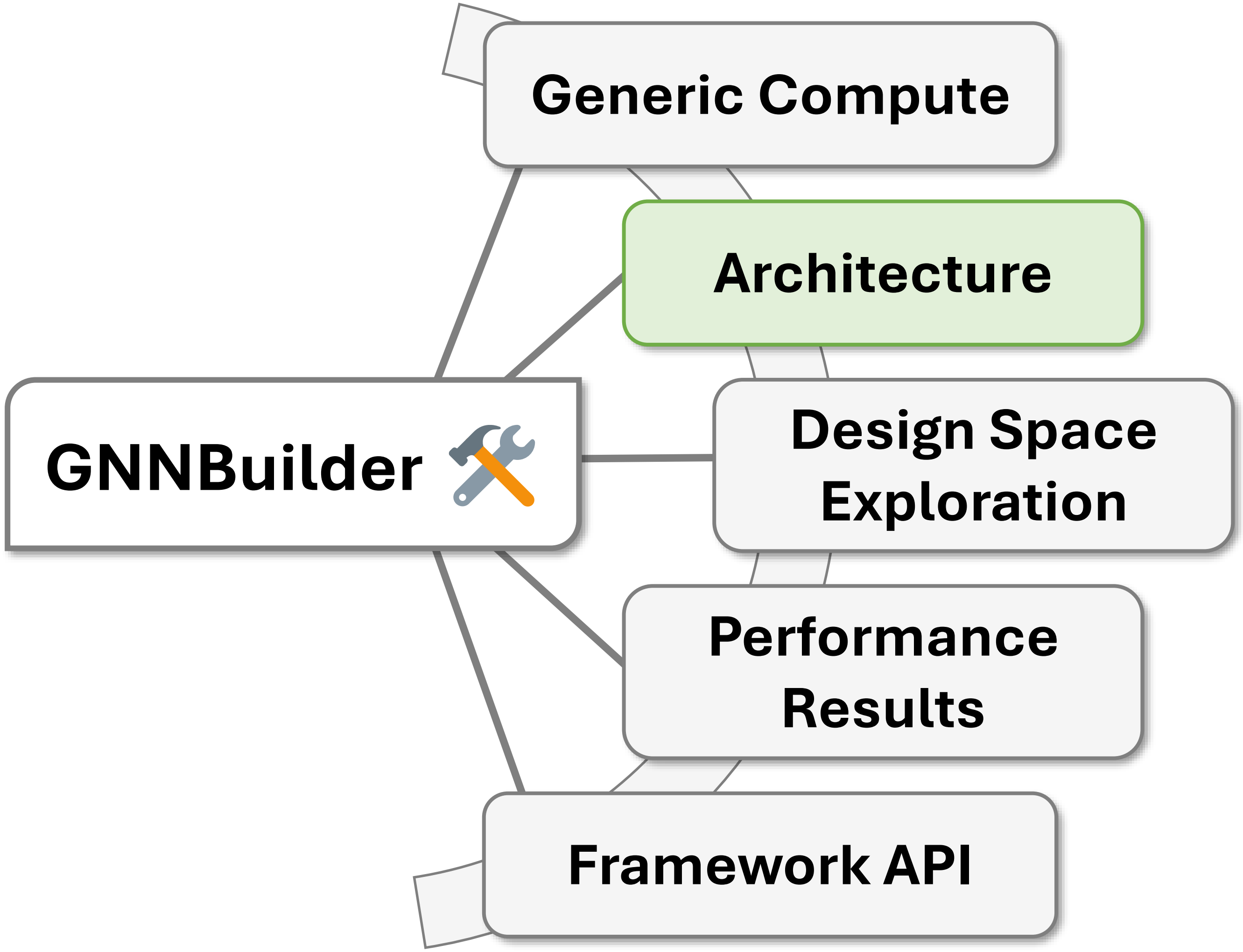
Compute

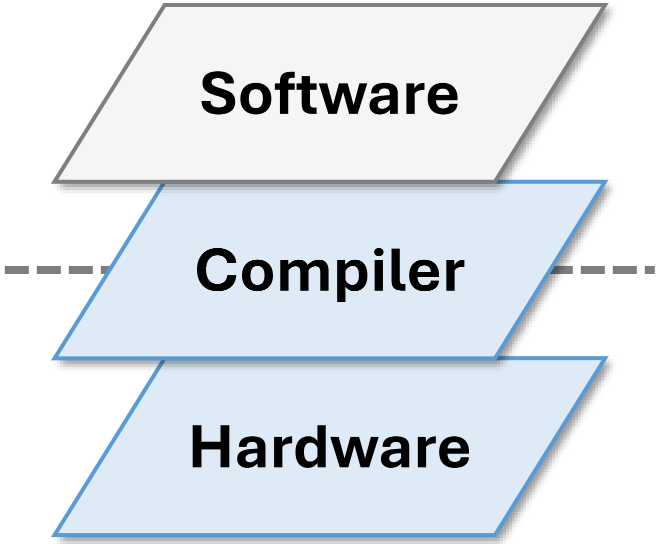
Neighbor Offset Table

0	0	2	4

1	4	2	4

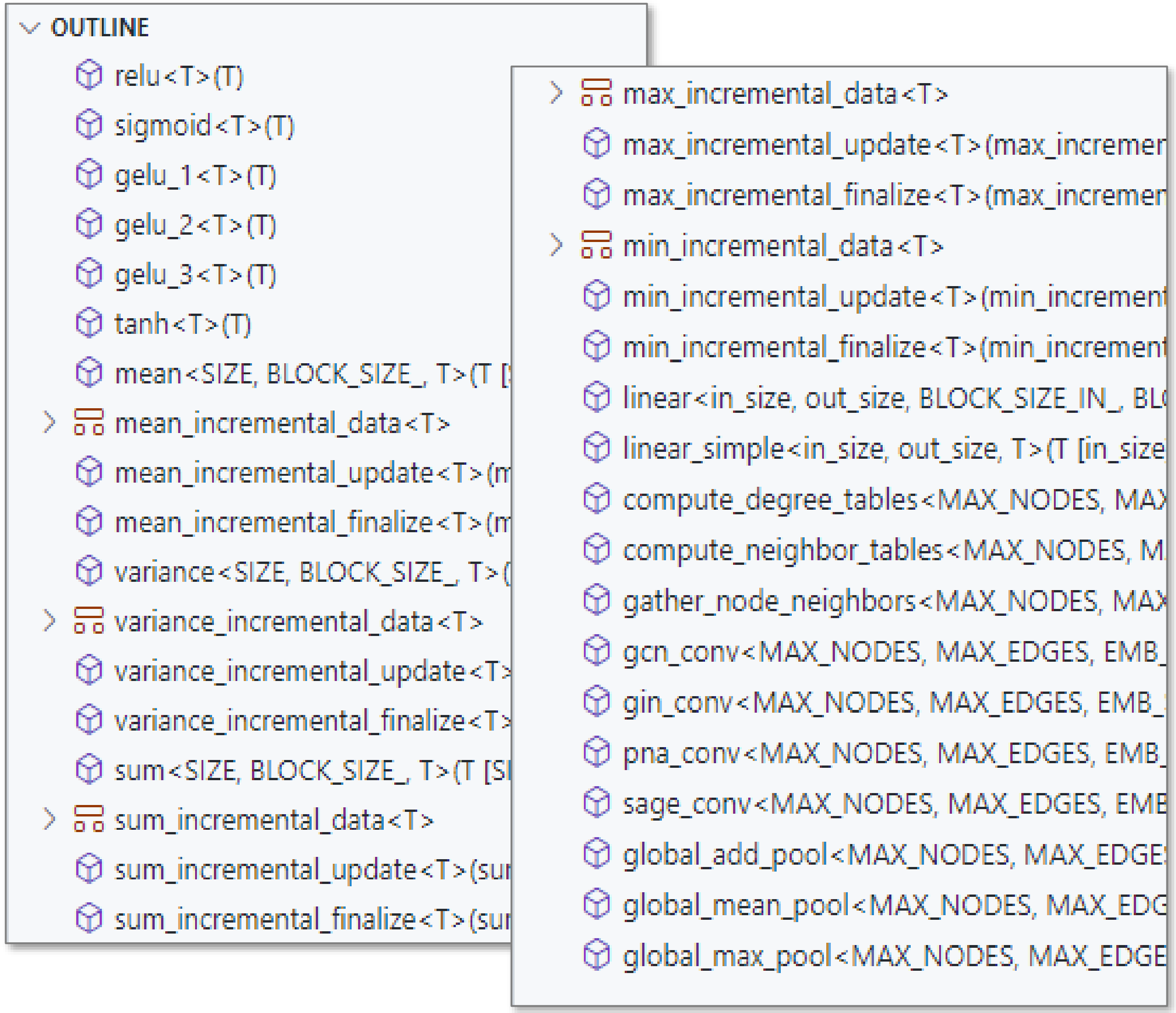
Neighbor Table


Generic Compute
Architecture
GNNBuilder
Design Space Exploration
Performance Results
Framework API


Software
Compiler
Hardware

HLS Implementation

GNNBuilder comes with a modular C++ HLS header library for GNN compute!



HLS Implementation

Break down key details of a GCN convolution layer.

Note how model parameters are passed down to the hardware.

```
template <int MAX_NODES,
          int MAX_EDGES,
          int EMB_SIZE_IN,
          int EMB_SIZE_OUT,
          typename T,
          int NUM_NODES_GUESS = MAX_NODES,
          int NUM_EDGES_GUESS = MAX_EDGES,
          int DEGREE_GUESS = MAX_NODES,
          int P_IN = 1,
          int P_OUT = 1>
void gcn_conv(
    int num_nodes,
    int num_edges,
    T node_embedding_table_in[MAX_NODES][EMB_SIZE_IN],
    T node_embedding_table_out[MAX_NODES][EMB_SIZE_OUT],
    int edge_list[MAX_EDGES][2],
    int neightbor_table_offsets[MAX_NODES],
    int neighbor_table[MAX_EDGES],
    int in_degree_table[MAX_NODES],
    int out_degree_table[MAX_NODES],
    T apply_lin_weight[EMB_SIZE_OUT][EMB_SIZE_IN],
    T apply_lin_bias[EMB_SIZE_OUT]) {
```

HLS Implementation

Used for static array / memory sizing.

Set by user at design time based on dataset / workload.

```
template <int MAX_NODES,
          int MAX_EDGES,
          int EMB_SIZE_IN,
          int EMB_SIZE_OUT,
          typename T,
          int NUM_NODES_GUESS = MAX_NODES,
          int NUM_EDGES_GUESS = MAX_EDGES,
          int DEGREE_GUESS = MAX_NODES,
          int P_IN = 1,
          int P_OUT = 1>
void gcn_conv(
    int num_nodes,
    int num_edges,
    T node_embedding_table_in[MAX_NODES][EMB_SIZE_IN],
    T node_embedding_table_out[MAX_NODES][EMB_SIZE_OUT],
    int edge_list[MAX_EDGES][2],
    int neighbor_table_offsets[MAX_NODES],
    int neighbor_table[MAX_EDGES],
    int in_degree_table[MAX_NODES],
    int out_degree_table[MAX_NODES],
    T apply_lin_weight[EMB_SIZE_OUT][EMB_SIZE_IN],
    T apply_lin_bias[EMB_SIZE_OUT]) {
```

HLS Implementation

Used for static array / memory sizing.

Used for passing in "`ap_fixed`" typedef as user-defined fixed-point datatype.

Set by user at design time based on task accuracy tradeoff.

```
template <int MAX_NODES,
          int MAX_EDGES,
          int EMB_SIZE_IN,
          int EMB_SIZE_OUT,
          typename T,
          int NUM_NODES_GUESS = MAX_NODES,
          int NUM_EDGES_GUESS = MAX_EDGES,
          int DEGREE_GUESS = MAX_NODES,
          int P_IN = 1,
          int P_OUT = 1>
void gcn_conv(
    int num_nodes,
    int num_edges,
    T node_embedding_table_in[MAX_NODES][EMB_SIZE_IN],
    T node_embedding_table_out[MAX_NODES][EMB_SIZE_OUT],
    int edge_list[MAX_EDGES][2],
    int neighbor_table_offsets[MAX_NODES],
    int neighbor_table[MAX_EDGES],
    int in_degree_table[MAX_NODES],
    int out_degree_table[MAX_NODES],
    T apply_lin_weight[EMB_SIZE_OUT][EMB_SIZE_IN],
    T apply_lin_bias[EMB_SIZE_OUT]) {
```

HLS Implementation

Used for static array / memory sizing.

Used for passing in "`ap_fixed`" typedef.

Used for "`#pragma HLS tripcount`" to improve latency estimation.

Estimated from dataset / workload at design time.

```cpp
template <int MAX_NODES,
          int MAX_EDGES,
          int EMB_SIZE_IN,
          int EMB_SIZE_OUT,
          typename T,
          int NUM_NODES_GUESS = MAX_NODES,
          int NUM_EDGES_GUESS = MAX_EDGES,
          int DEGREE_GUESS = MAX_NODES,
          int P_IN = 1,
          int P_OUT = 1>
void gcn_conv(
    int num_nodes,
    int num_edges,
    T node_embedding_table_in[MAX_NODES][EMB_SIZE_IN],
    T node_embedding_table_out[MAX_NODES][EMB_SIZE_OUT],
    int edge_list[MAX_EDGES][2],
    int neightbor_table_offsets[MAX_NODES],
    int neighbor_table[MAX_EDGES],
    int in_degree_table[MAX_NODES],
    int out_degree_table[MAX_NODES],
    T apply_lin_weight[EMB_SIZE_OUT][EMB_SIZE_IN],
    T apply_lin_bias[EMB_SIZE_OUT]) {
```

HLS Implementation

Used for static array / memory sizing.

Used for passing in “ap_fixed” typedef.

Used for “#pragma HLS tripcount” to improve latency estimation.

Used to parallelize intra-layer operations such as node transformations.

Set by the user at design time based on perf. requirements and DSE exploration.

```
template <int MAX_NODES,
          int MAX_EDGES,
          int EMB_SIZE_IN,
          int EMB_SIZE_OUT,
          typename T,
          int NUM_NODES_GUESS = MAX_NODES,
          int NUM_EDGES_GUESS = MAX_EDGES,
          int DEGREE_GUESS = MAX_NODES,
          int P_IN = 1,
          int P_OUT = 1>
void gcn_conv(
    int num_nodes,
    int num_edges,
    T node_embedding_table_in[MAX_NODES][EMB_SIZE_IN],
    T node_embedding_table_out[MAX_NODES][EMB_SIZE_OUT],
    int edge_list[MAX_EDGES][2],
    int neightbor_table_offsets[MAX_NODES],
    int neighbor_table[MAX_EDGES],
    int in_degree_table[MAX_NODES],
    int out_degree_table[MAX_NODES],
    T apply_lin_weight[EMB_SIZE_OUT][EMB_SIZE_IN],
    T apply_lin_bias[EMB_SIZE_OUT]) {
```

HLS Implementation

Used for static array / memory sizing.

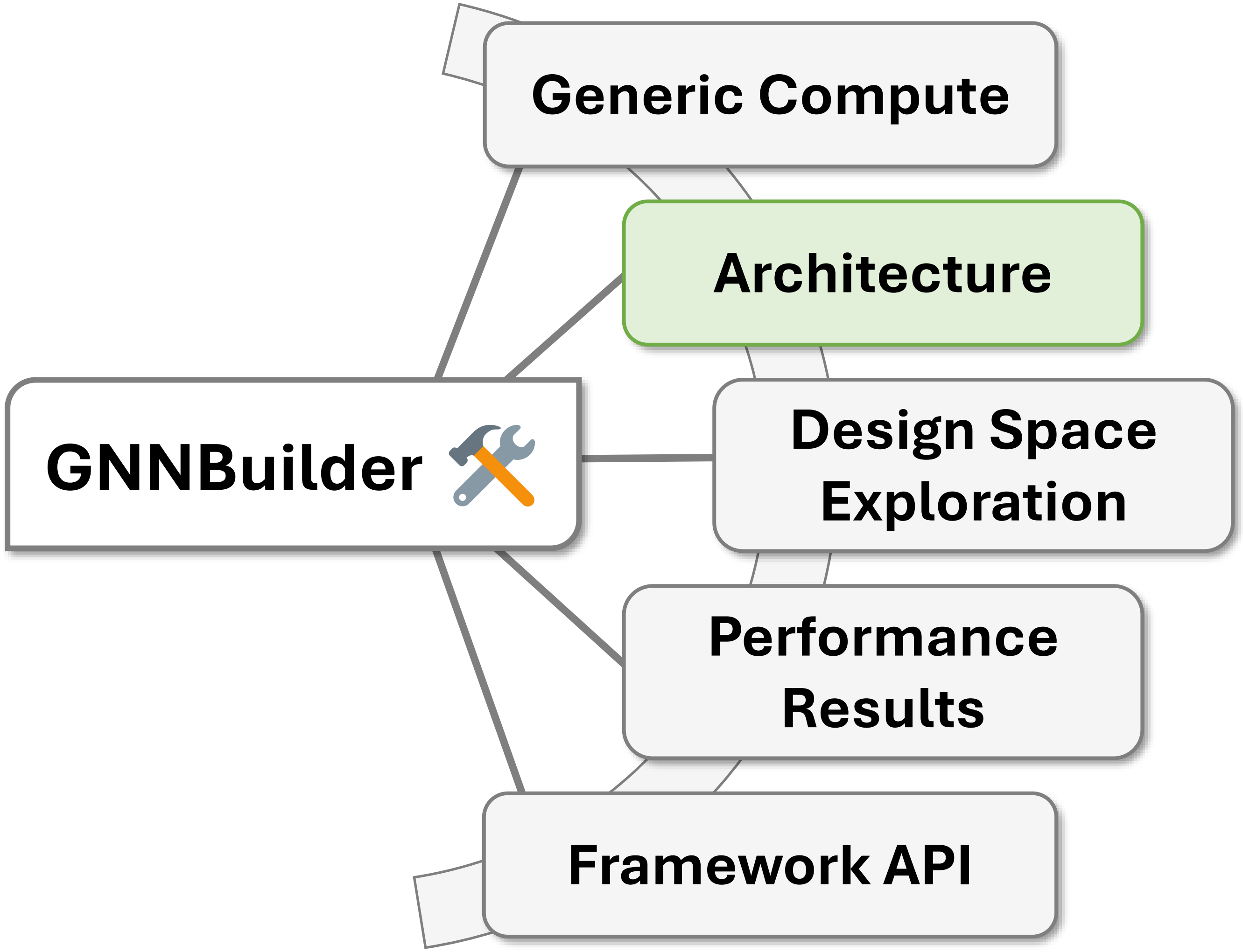
Used for passing in "`ap_fixed`" typedef.

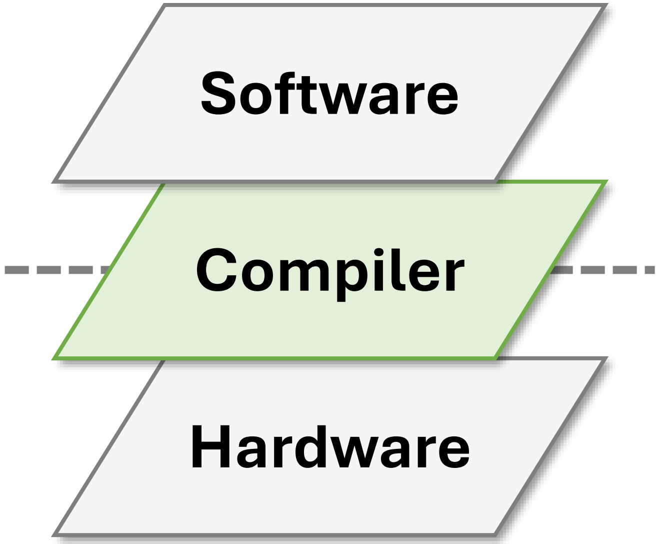
Used for "`#pragma HLS tripcount`" to improve latency estimation.

Used to parallelize intra-layer operations such as node transformations.

More details in the code repo. and documentation.

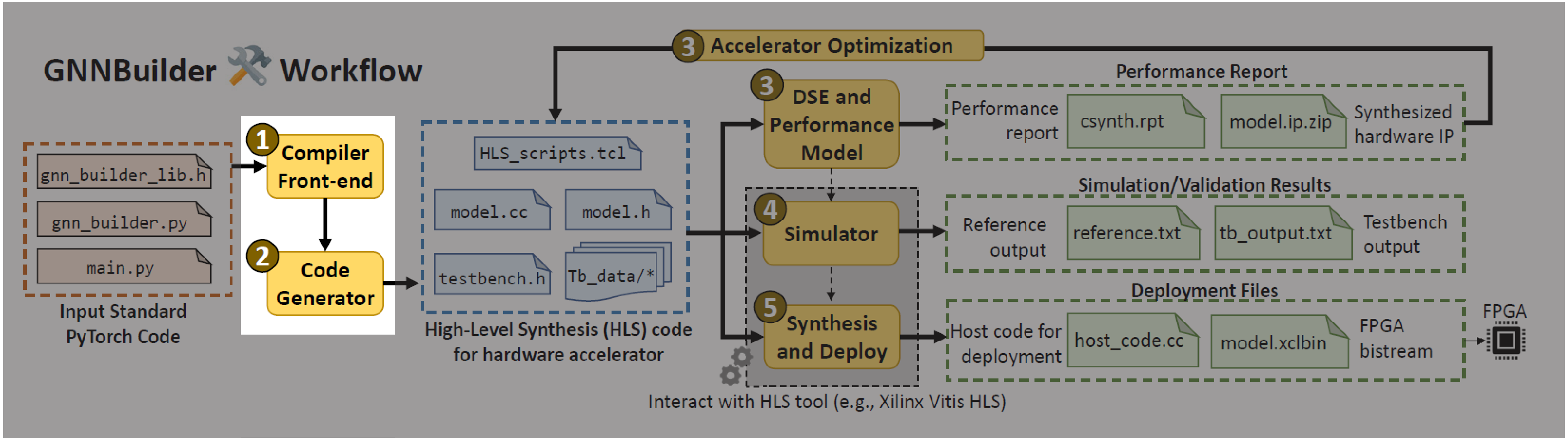
```
template <int MAX_NODES,
          int MAX_EDGES,
          int EMB_SIZE_IN,
          int EMB_SIZE_OUT,
          typename T,
          int NUM_NODES_GUESS = MAX_NODES,
          int NUM_EDGES_GUESS = MAX_EDGES,
          int DEGREE_GUESS = MAX_NODES,
          int P_IN = 1,
          int P_OUT = 1>
void gcn_conv(
    int num_nodes,
    int num_edges,
    T node_embedding_table_in[MAX_NODES][EMB_SIZE_IN],
    T node_embedding_table_out[MAX_NODES][EMB_SIZE_OUT],
    int edge_list[MAX_EDGES][2],
    int neightbor_table_offsets[MAX_NODES],
    int neighbor_table[MAX_EDGES],
    int in_degree_table[MAX_NODES],
    int out_degree_table[MAX_NODES],
    T apply_lin_weight[EMB_SIZE_OUT][EMB_SIZE_IN],
    T apply_lin_bias[EMB_SIZE_OUT]) {
```


GNNBuilder
Generic Compute
Architecture
Design Space Exploration
Performance Results
Framework API


Software
Compiler
Hardware

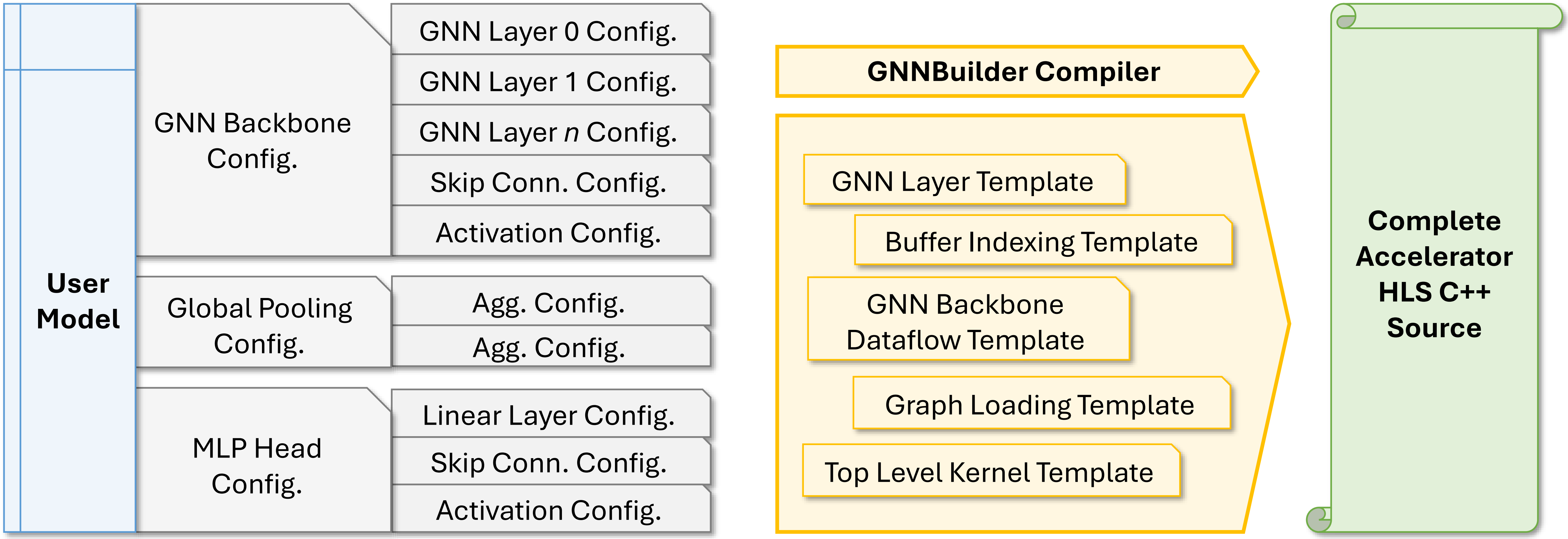
Compiler and Code Generation

GNNBuilder implements a **template-based compiler** that lowers PyTorch models, down to a structured model IR, and then down to HLS C++ source code.



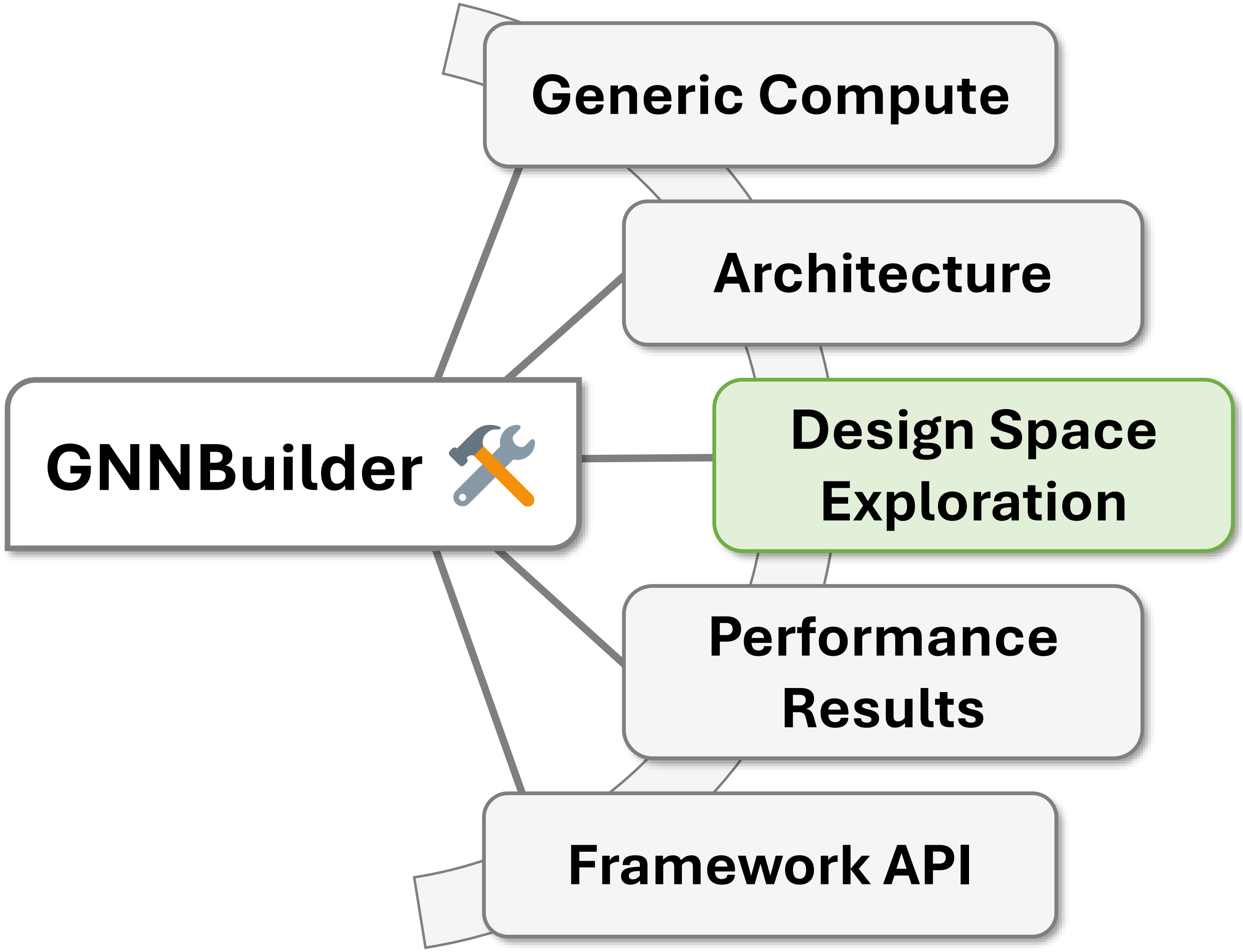
Compiler and Code Generation

We extract model and hardware architecture logic into individual compiler template components.



Extra Features

- User specified fixed-point data type for features and weights
  - Uses ap_fixed format Xilinx HLS library
  - Also uses hls::math library
- Auto build and run of C++ testbench using Clang
  - Can export PyTorch Geometric dataset to testbench files
  - Can use testbench as a correct model to explore quantization loss
- Auto synthesis using Vitis HLS
  - Runs synthesis flow for a user-specified device and clock speed
  - Extracts synthesis results: latency + resource usage
- Auto deploy to FPGA with host code (work in progress)
  - Run Vitis + Vivado build flow to generate .xclbin
  - Generate OpenCL host code to evaluate dataset (like testbench)
  - Looking into using XRT support in the PYNQ library to keep everything in Python

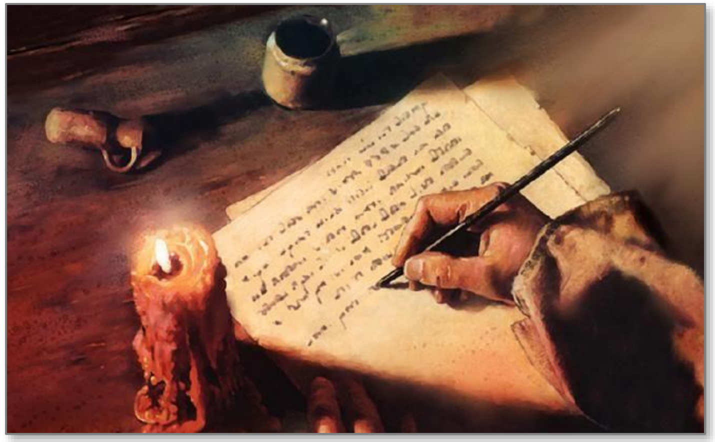


- Background
  - DL for Graphs
  - GNNs
  - Accelerators
- **GNNBuilder**
  - Generic Compute
  - Architecture
  - **Design Space Exploration**
  - Performance Results
  - Framework API
- Limitations
- Ongoing Work
- Documentation and Open Source

Design Space Exploration

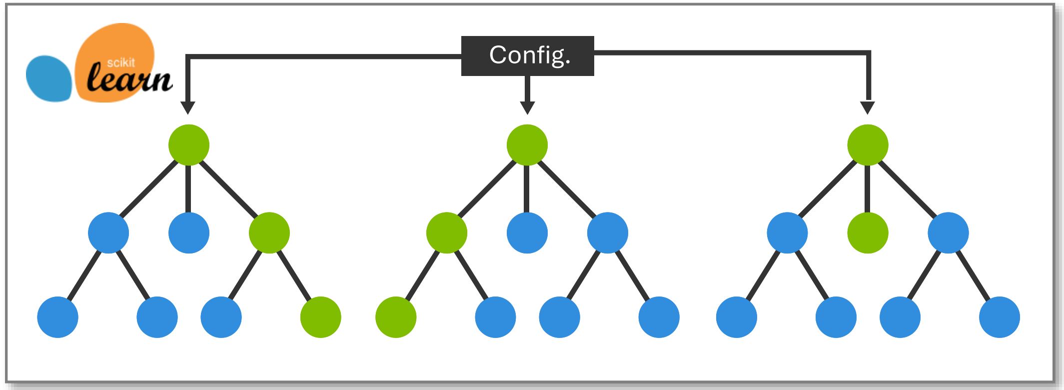
Predicting **Latency** and **Resource Usage**

Analytical Model



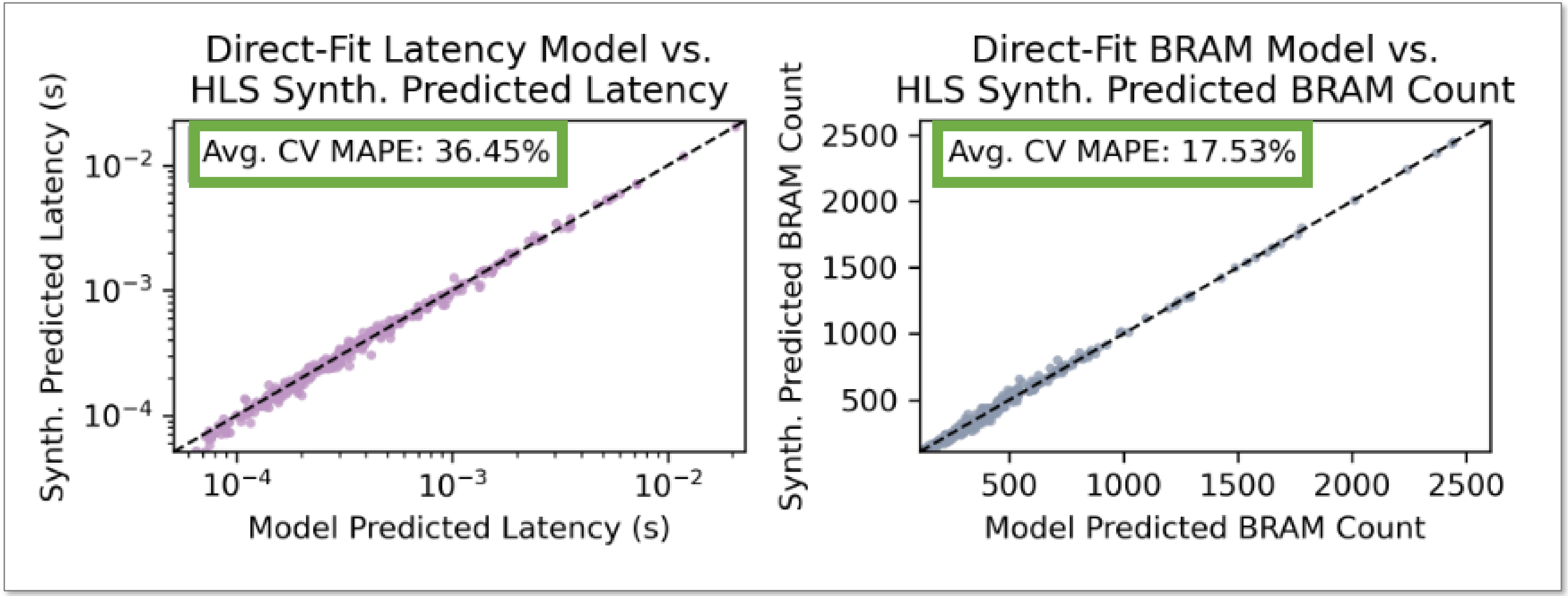
Complete Failure!
Too Much Complexity

ML-Based Model



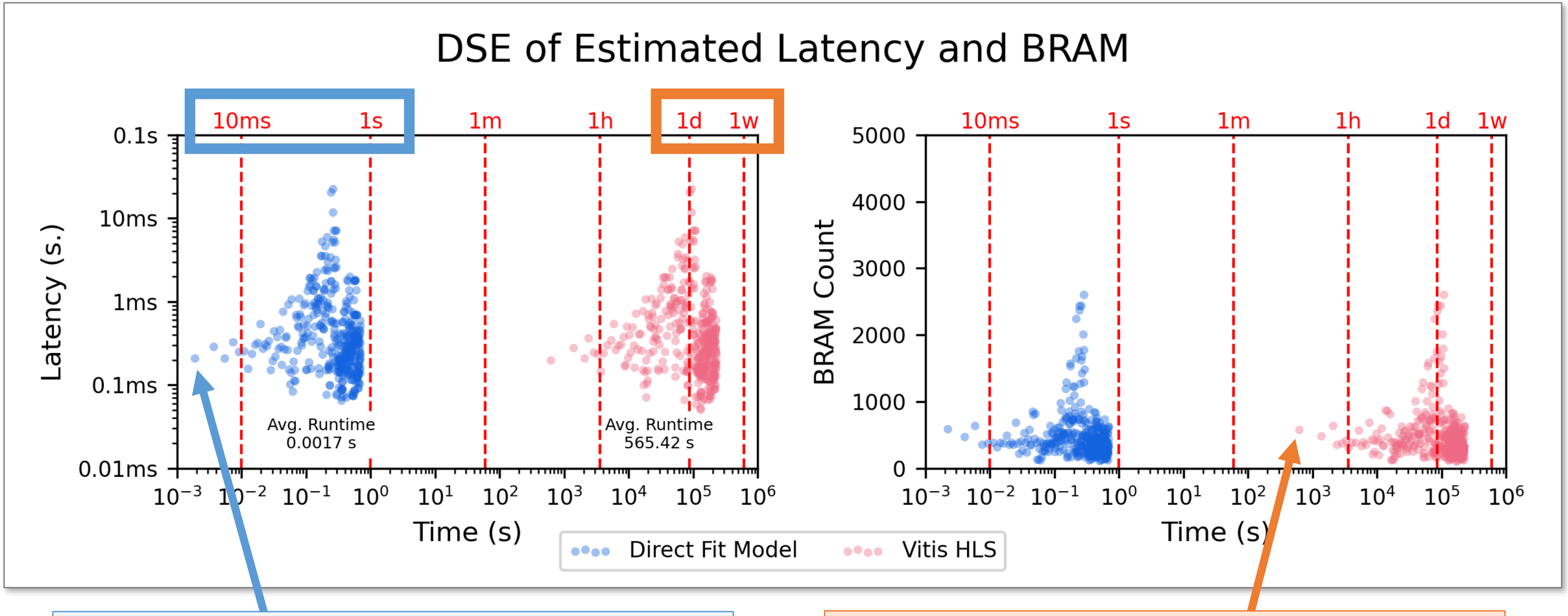
Success!
Generalizes Across Accelerator Configs.

Design Space Exploration



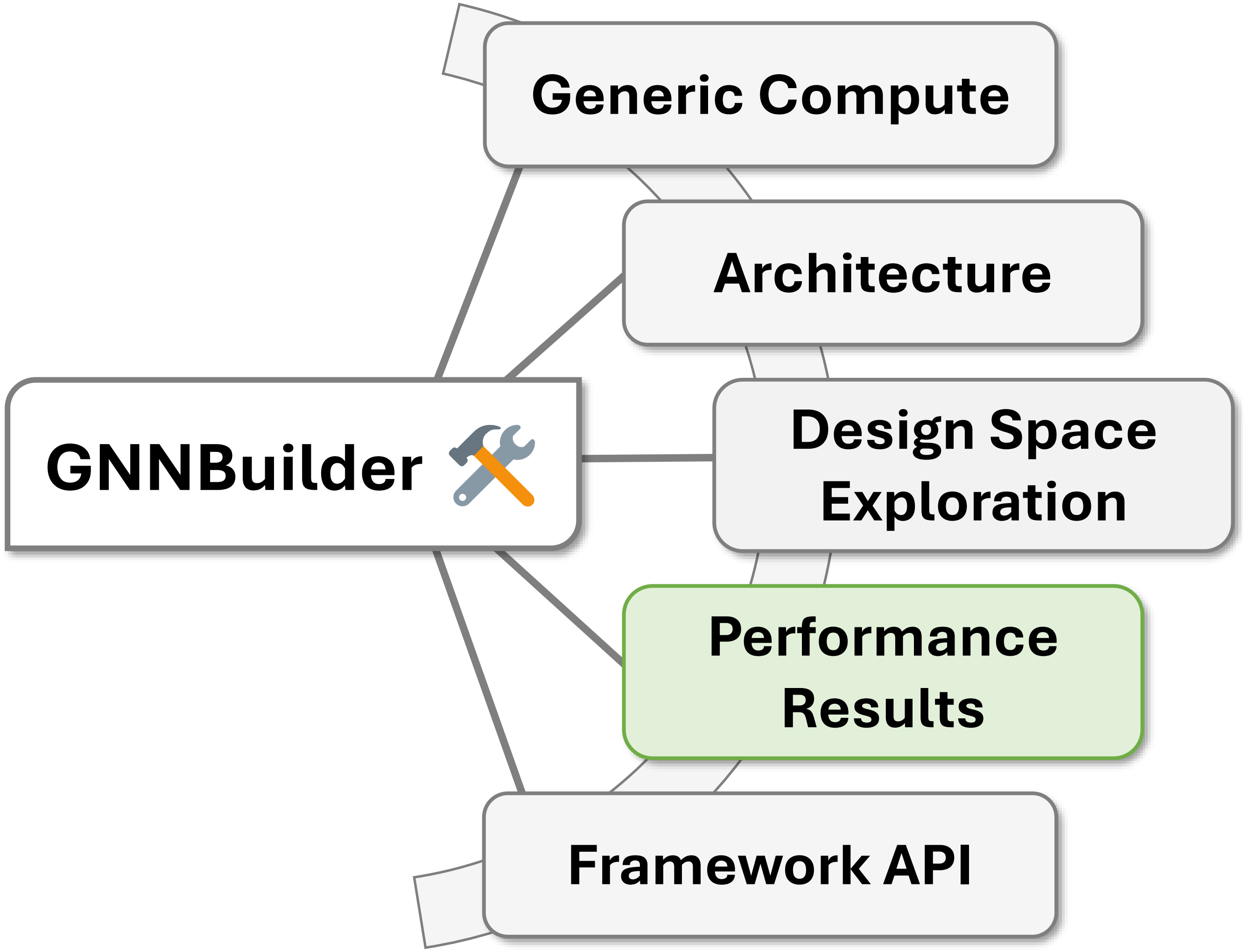
MAPE: Mean Absolute Percent Error

Design Space Exploration



Single DSE Model - 1.7 ms.

Single HLS Synthesis - 9.5 min.



Background
- DL for Graphs
- GNNs
- Accelerators

GNNBuilder
- Generic Compute
- Architecture
- Design Space Exploration
- **Performance Results**
- Framework API

Limitations

Ongoing Work

Documentation and Open Source

Accelerator Performance

- **PyG-CPU**: A PyTorch Geometric CPU model
- **PyG-GPU**: A PyTorch Geometric GPU model
- **CPP-CPU**: A C++ floating point CPU model
- **FPGA-Base**: Proposed HW model with no parallelism
- **FPGA-Parallel**: Proposed HW model with parallelism

CPU: Intel Xeon Gold 6226R
GPU: Nvidia RTX A6000
FPGA: Xilinx Alveo U280 @ 300 MHz.

We use a **batch_size=1** across all evaluations for **real-time inference applications**.

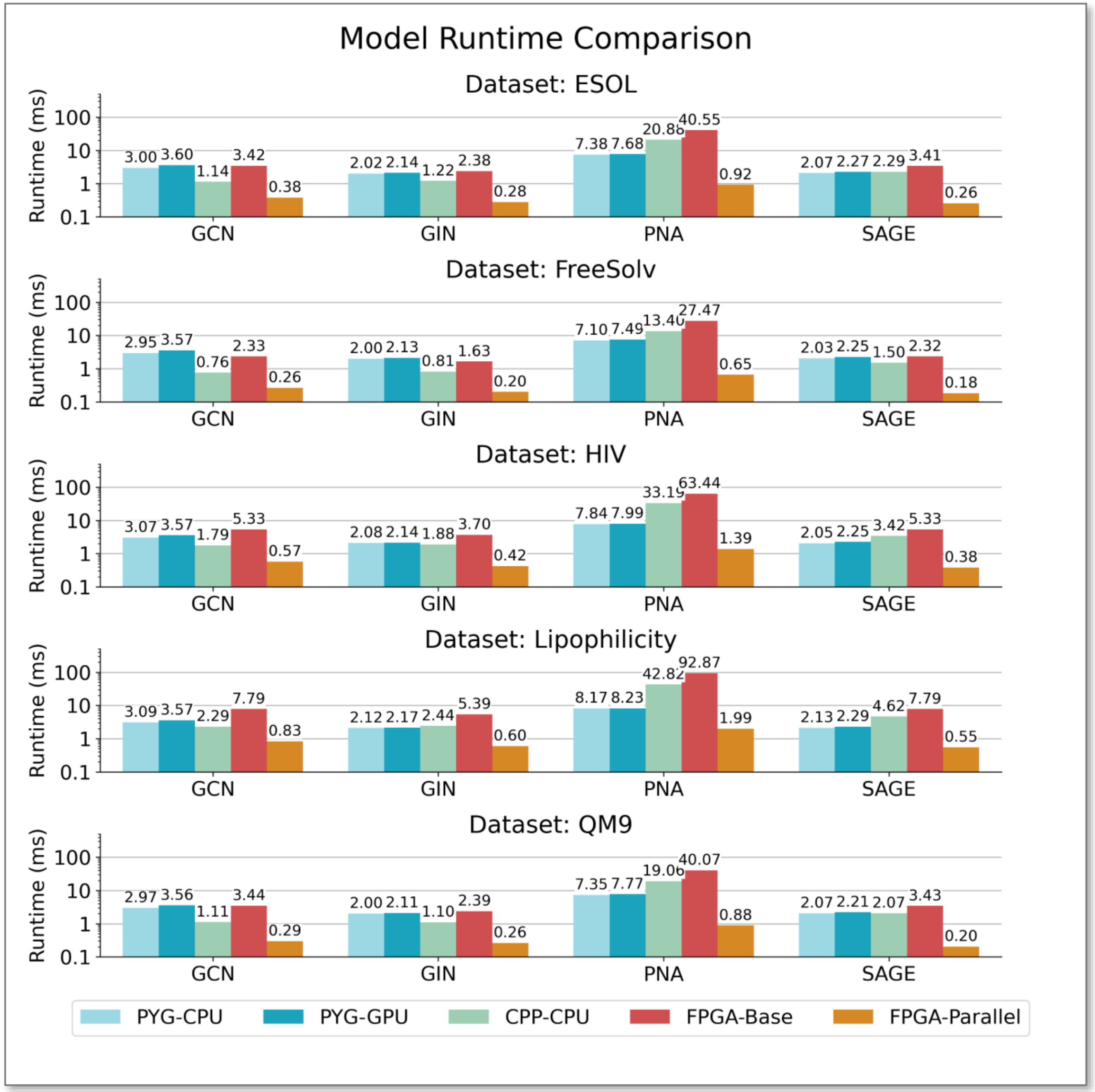
Accelerator Performance

- **PyG-CPU**: A PyTorch Geometric CPU model
- **PyG-GPU**: A PyTorch Geometric GPU model
- **CPP-CPU**: A C++ floating point CPU model
- **FPGA-Base**: Proposed HW model with no parallelism
- **FPGA-Parallel**: Proposed HW model with parallelism

	PyG-CPU	PyG-GPU	CPP-CPU
GCN	6.46x	7.66x	3.04x
GIN	5.81x	6.08x	4.24x
PNA	6.48x	6.70x	22.14x
SAGE	6.58x	7.16x	8.84x
Geo. Mean	6.33x	6.87x	7.08x

Geo. Mean	**6.33x**	**6.87x**	**7.08x**

We achieve consistent speedups overs CPU and GPU implementations!

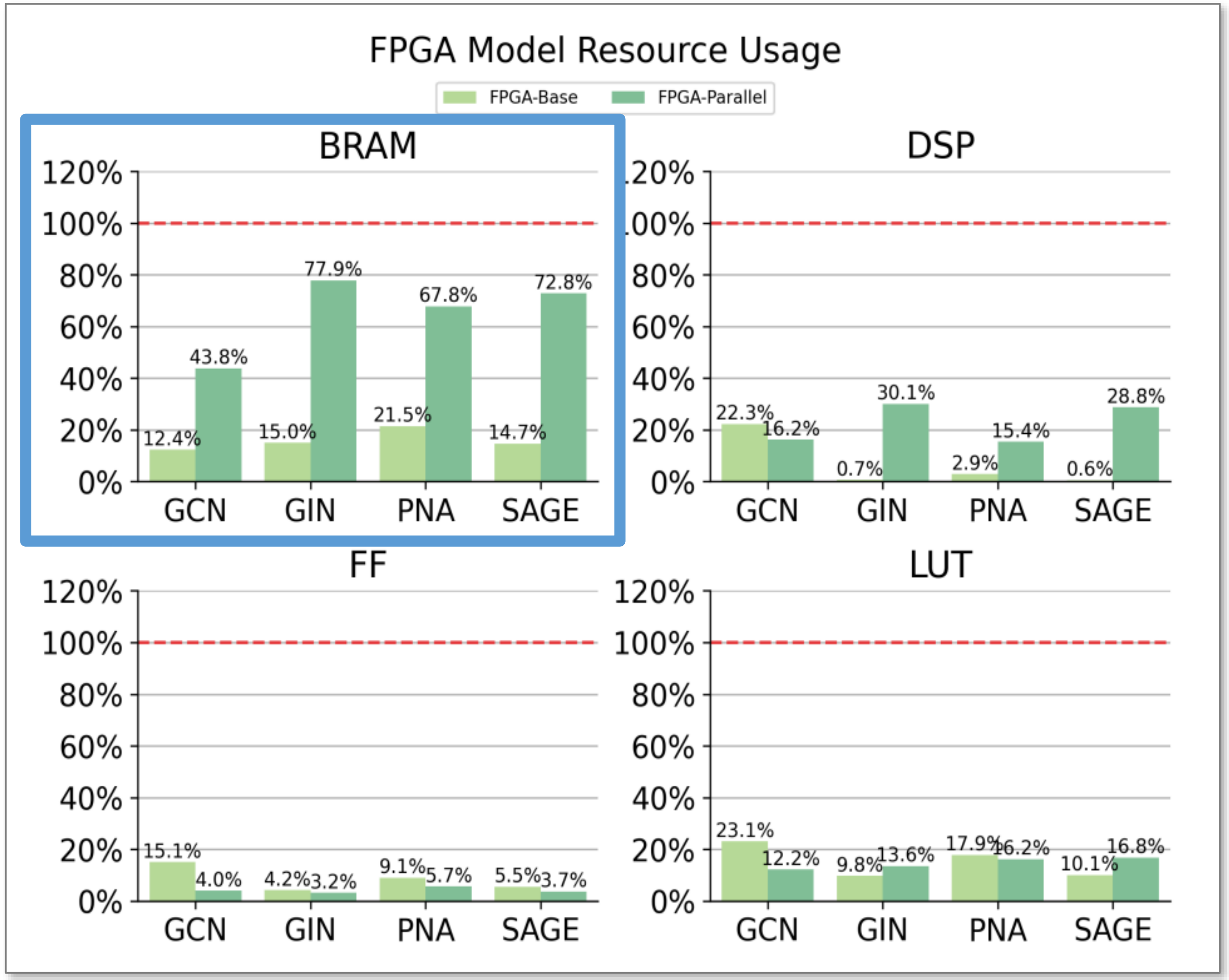


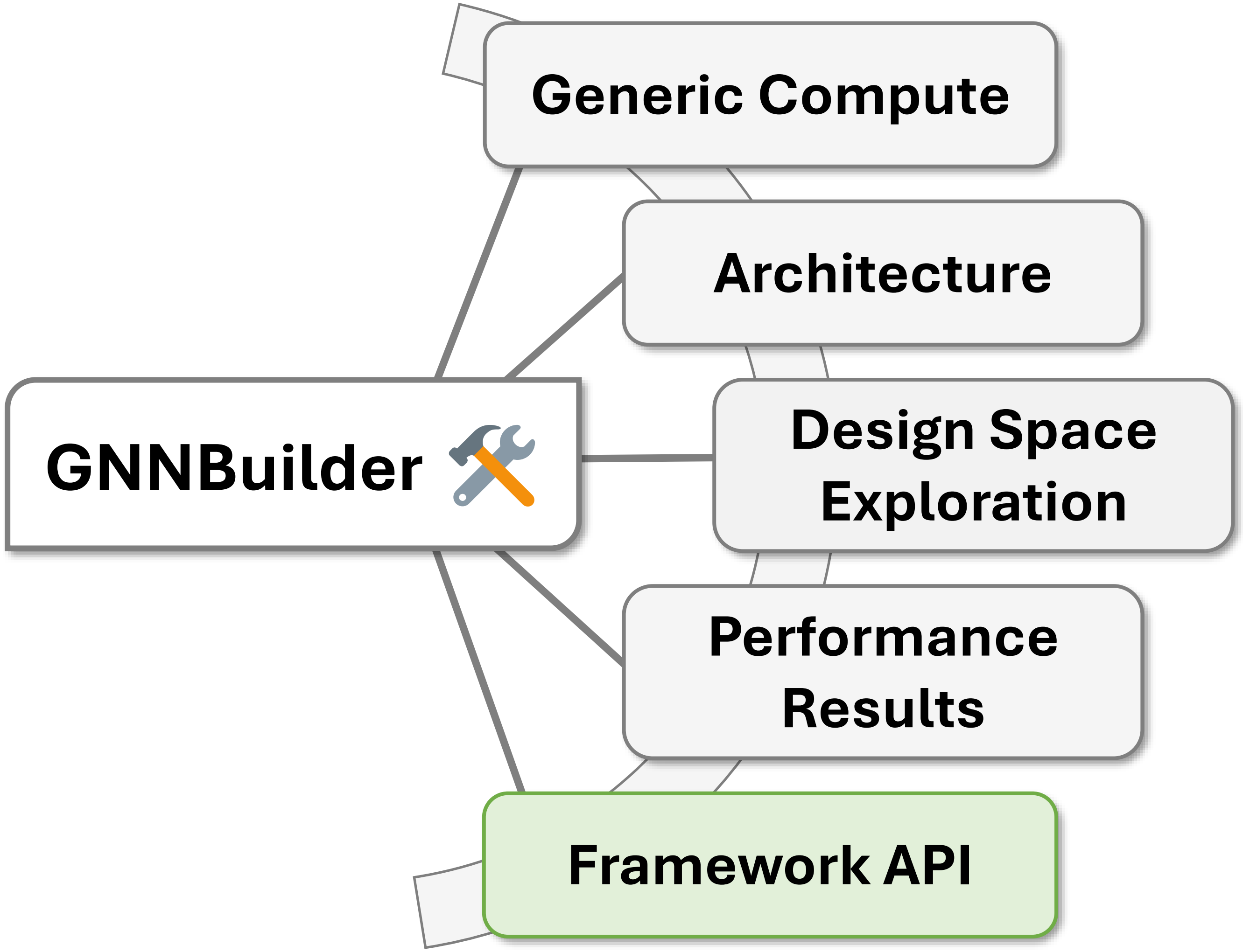
Accelerator Performance

- **PyG-CPU**: A PyTorch Geometric CPU model
- **PyG-GPU**: A PyTorch Geometric GPU model
- **CPP-CPU**: A C++ floating point CPU model
- **FPGA-Base**: Proposed HW model with no parallelism
- **FPGA-Parallel**: Proposed HW model with parallelism

	PyG-CPU	PyG-GPU	CPP-CPU
GCN	6.46x	7.66x	3.04x
GIN	5.81x	6.08x	4.24x
PNA	6.48x	6.70x	22.14x
SAGE	6.58x	7.16x	8.84x
Geo. Mean	6.33x	6.87x	7.08x
Geo. Mean	**6.33x**	**6.87x**	**7.08x**

Resource usage is dominated by on-chip memory (BRAM / URAM)





Background
- DL for Graphs
- GNNs
- Accelerators

GNNBuilder
- Generic Compute
- Architecture
- Design Space Exploration
- Performance Results
- **Framework API**

Limitations

Ongoing Work

Documentation and Open Source

Framework API

USER APIs FOR THE GNN_BUILDER PYTHON LIBRARY.

API Functions	Description
model.GNNModel(nn.Module)	PyTorch Model for GNNBuilder Arch.
model.GCNConv_GNNB(nn.Module)	GCN Conv. Layer
model.GINConv_GNNB(nn.Module)	GIN Conv. Layer
model.PNAConv_GNNB(nn.Module)	PNA Conv. Layer
model.SAGEConv_GNNB(nn.Module)	GraphSAGE Conv. Layer
model.GlobalPooling(nn.Module)	Global Graph Pooling Layer
model.MLP(nn.Module)	MLP Prediction Head
code_gen.Project()	GNNBuilder Project Class
Project.gen_hw_model()	Code Gen. For HW Kernel
Project.gen_testbench()	Code Gen. For Testbench
Project.gen_makefile()	Code Gen. For Testbench Makefile
Project.gen_vitis_hls_tcl_script()	Code Gen. For Vitis HLS Synth. Script
Project.build_and_run_testbench()	Build and Run Testbench
Project.run_vitis_hls_synthesis()	Launch Vitis HLS Synthesis Run

```python
import torch.nn as nn
from torch_geometric.datasets import
    MoleculeNet

import gnnbuilder as gnnb

dataset = MoleculeNet(root="./tmp/MoleculeNet
    ", name="hiv")

model = gnnb.GNNModel(
    graph_input_feature_dim=dataset.
    num_features,
    graph_input_edge_dim=dataset.
    num_edge_features,
    gnn_hidden_dim=16,
    gnn_num_layers=2,
    gnn_output_dim=8,
    gnn_conv=gnnb.SAGEConv_GNNB,
    gnn_activation=nn.ReLU,
    gnn_skip_connection=True,
    global_pooling=gnnb.GlobalPooling(["add",
     "mean", "max"]),
    mlp_head=gnnb.MLP(in_dim=8 * 3, out_dim=
    dataset.num_classes, hidden_dim=8,
    hidden_layers=3, activation=nn.ReLU, p_in
    =8, p_hidden=4, p_out=1),
    output_activation=None,
    gnn_p_in=1,
    gnn_p_hidden=8,
    gnn_p_out=4
)
```

```python
MAX_NODES = 600
MAX_EDGES = 600
num_nodes_avg, num_edges_avg = gnnb.
    compute_average_nodes_and_edges(dataset)
degree_avg =  gnnb.utils.
    compute_average_degree(dataset)
```

```python
proj = gnnb.Project(
     "gnn_model",
     model,
     "classification_integer",
     VITIS_HLS_PATH,
     BUILD_DIR,
     dataset=dataset,
     max_nodes=MAX_NODES,
     max_edges=MAX_EDGES,
     num_nodes_guess=num_nodes_avg,
     num_edges_guess=num_edges_avg,
     degree_guess=degree_avg,
     float_or_fixed="fixed",
     fpx=FPX(32, 16),
     fpga_part="xcu280-fsvh2892-2L-e",
     n_jobs=32,
)
```

```python
proj.gen_hw_model()
proj.gen_testbench()
proj.gen_makefile()
proj.gen_vitis_hls_tcl_script()
proj.gen_makefile_vitis()

tb_data = proj.build_and_run_testbench()
print(tb_data)
synth_data = proj.run_vitis_hls_synthesis()
print(synth_data)
```

Minimum working example to generate, testbench, and synth a GNN accelerator.

Import GNNBuilder

Specify Dataset / Workload

Specific GNN PyTorch Model
(can be used for training)
(or load a pre-trained model)

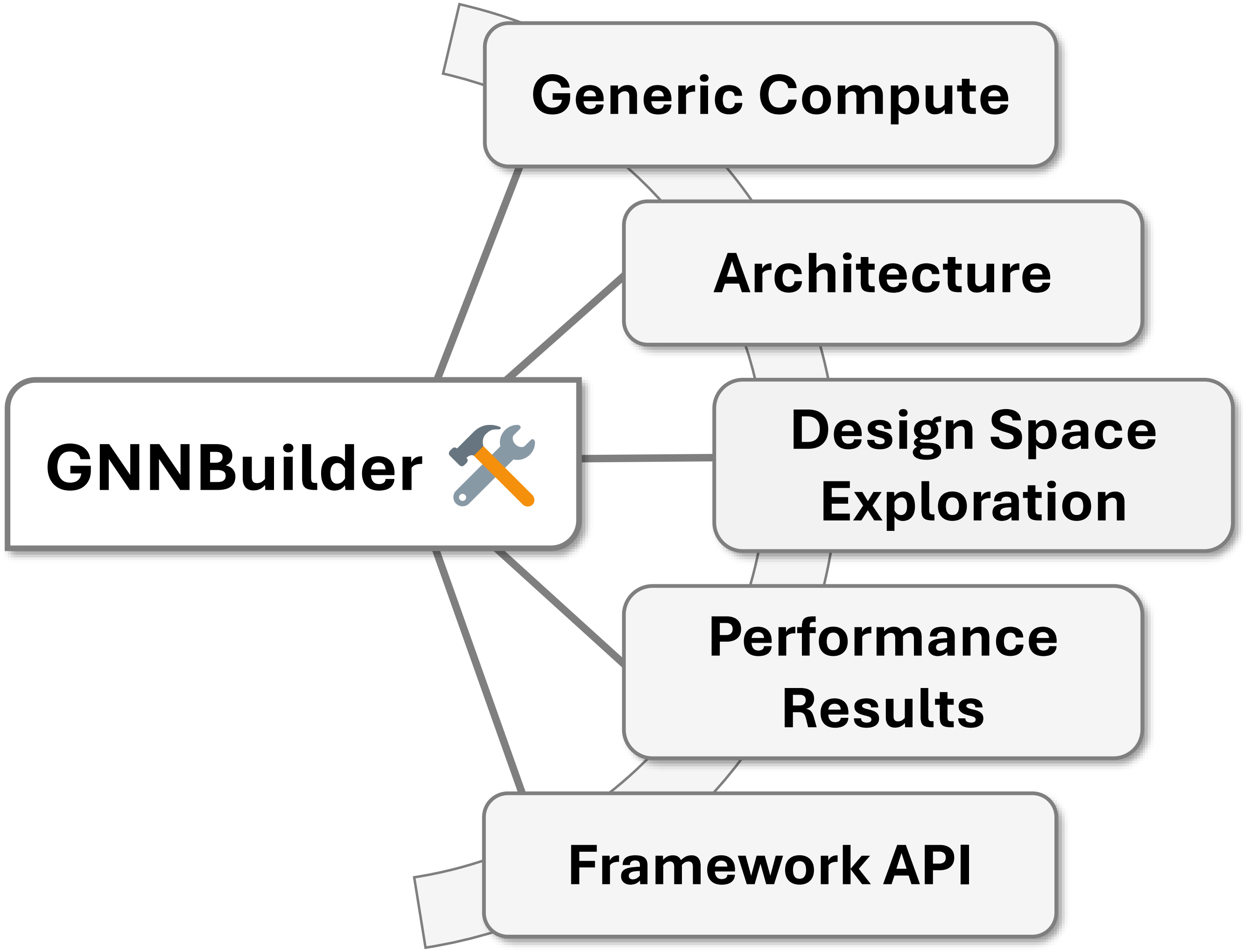
Characterize Dataset / Workload Graph Properties

Specify GNNBuilder Accelerator Project
(Define build path, device, tools, and fixed-point spec.)

Compile Down to HLS Design, Testbench, and Build Files

Call Tools to Run Hardware Testbench, Synthesis, and Implementation

Minimum working example to generate, testbench, and synth a GNN accelerator.



Background

DL for Graphs
GNNs
Accelerators

GNNBuilder

Generic Compute
Architecture
Design Space Exploration
Performance Results
Framework API

Limitations

Ongoing Work

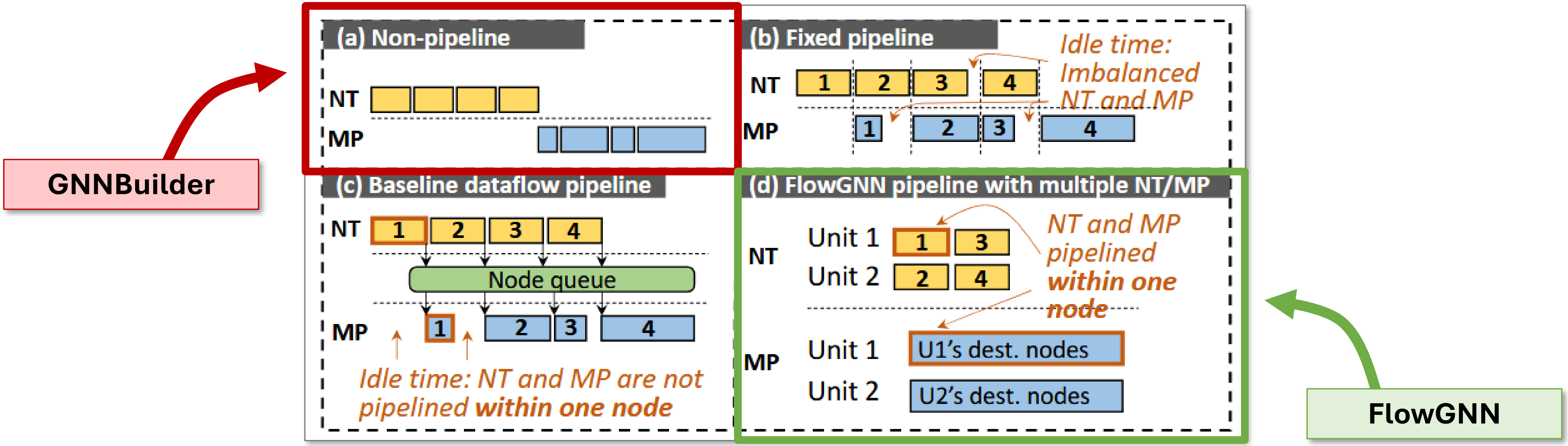
Documentation and Open Source

Limitations

GNNBuilder ⇒ 7x speedup vs. **FlowGNN** ⇒ up to 400x speedup

No Node-Level Parallelism in GNNBuilder

Each node is evaluated sequentially, nodes are not processed in parallel.



Limitations

GNNBuilder ⇒ 7x speedup vs. **FlowGNN** ⇒ up to 400x speedup

No Node-Level Parallelism in GNNBuilder

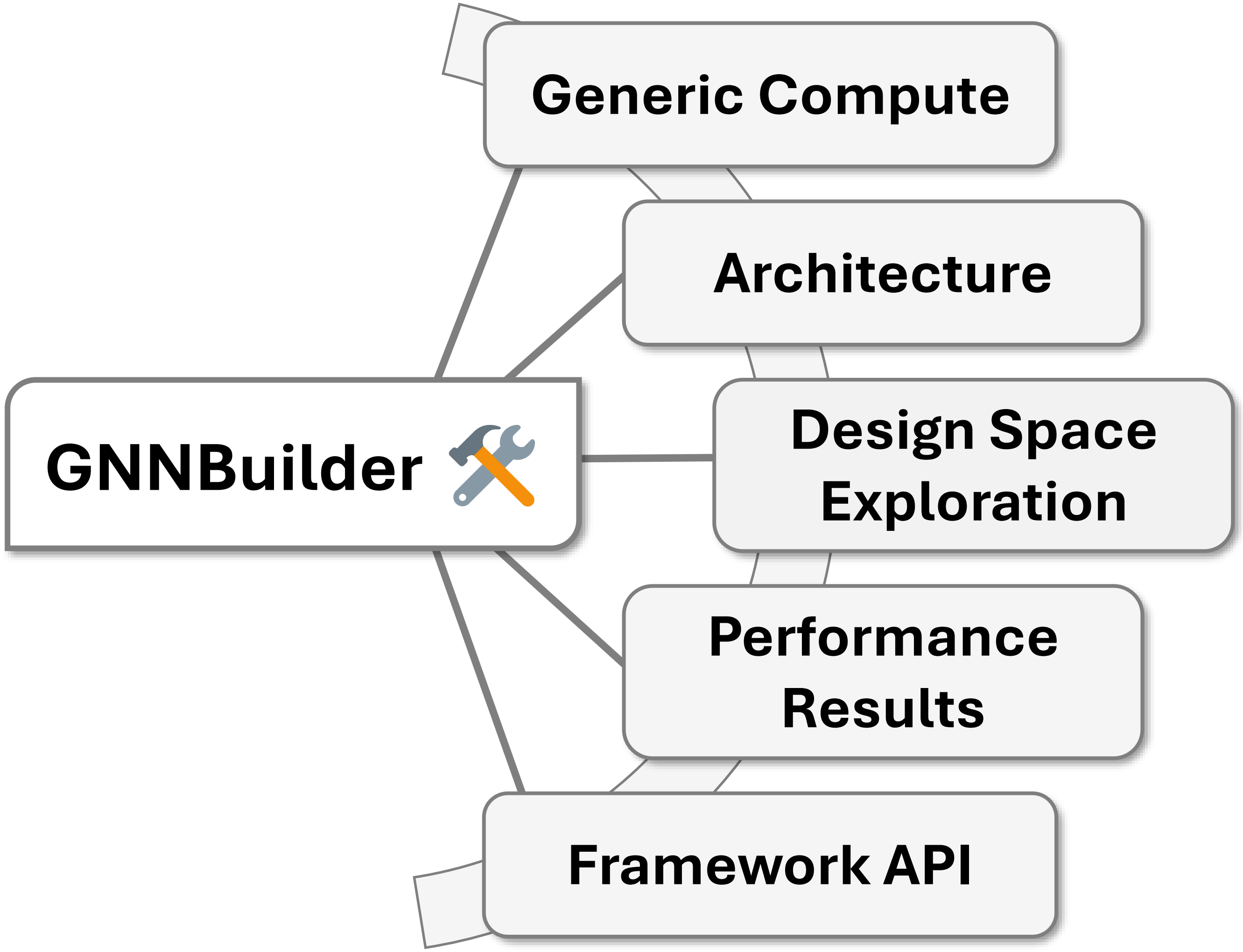
Each node is evaluated sequentially, nodes are not processed in parallel.

GNNBuilder

Adapt dataflow pipelining and parallelism from previous GNN accelerator publications.

GenGNN and ***FlowGNN***

FlowGNN



Background
DL for Graphs
GNNs
Accelerators
GNNBuilder
Generic Compute
Architecture
Design Space Exploration
Performance Results
Framework API
Limitations
Ongoing Work
Documentation and Open Source

Ongoing Work

Large Graph Support via Efficient Off-Chip HBM Access

Irregular unknown memory access patterns at runtime

Automated XRT and PYNQ On-Device Deployment and Evaluation

Expand Support for Novel GNN and NN Features

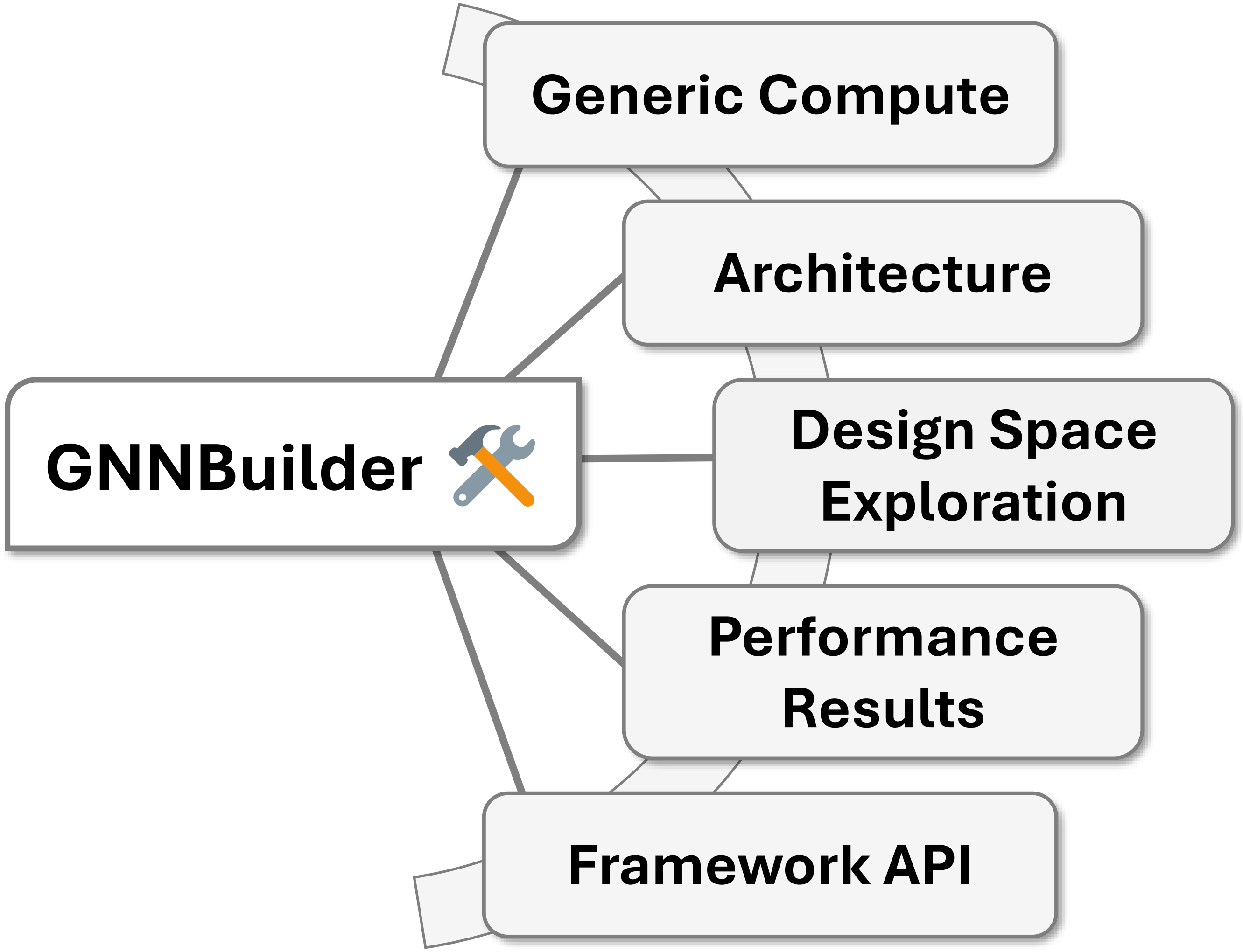
GAT, SimpleConv, Graph Sampling, Embedding Lookup Layers, Complex Edge Convs, ...

Multi-Tiered Accelerator Evaluation API

Fitted Models → LightningSim Co-Simulation → Vitis HW Emulation → On-Device Evaluation

Application Demos and Deployment

High Energy Physics Triggers and Reconstruction, Real Time Point Cloud Processing, ...



Background

- DL for Graphs
- GNNs
- Accelerators

GNNBuilder

- Generic Compute
- Architecture
- Design Space Exploration
- Performance Results
- Framework API

Limitations

Ongoing Work

Documentation and Open Source

Source Code + Documentation + Setup

```
pip install git+https://github.com/sharc-lab/gnn-builder.git
```

```
conda install --channel https://sharc-lab.github.io/gnn-builder/repo gnn-builder
```

Source Code

github.com/sharc-lab/gnn-builder

Documentation + Tutorials

sharc-lab.github.io/gnn-builder/

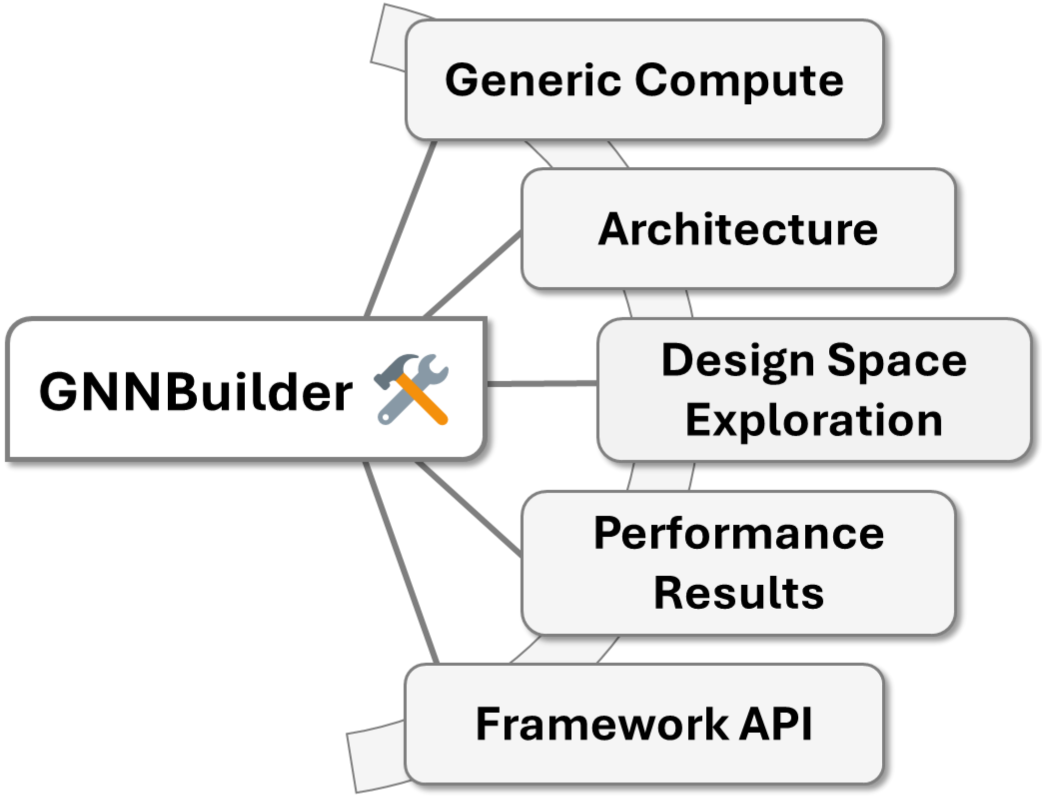
How to Add a New Graph Convolution Layer

1. Clone the repo...
2. Add HW kernel to header library...
3. Add associated layer wrapper to Python library...
4. Add the relevant compiler template unit...
5. Run testbench and synth...
6. Submit pull request...

See documentation for detailed walkthroughs and tutorials.

GNNBuilder

An Automated Framework for Generic Graph Neural Network Accelerator Generation, Simulation, and Optimization



Stefan Abi-Karam
stefanabikaram@gatech.edu

SHARC Lab @ Georgia Tech
sharclab.ece.gatech.edu

TMPO / CIPHER
Georgia Tech Research Institute

Interested in perusing academia…

Stefan Abi-Karam

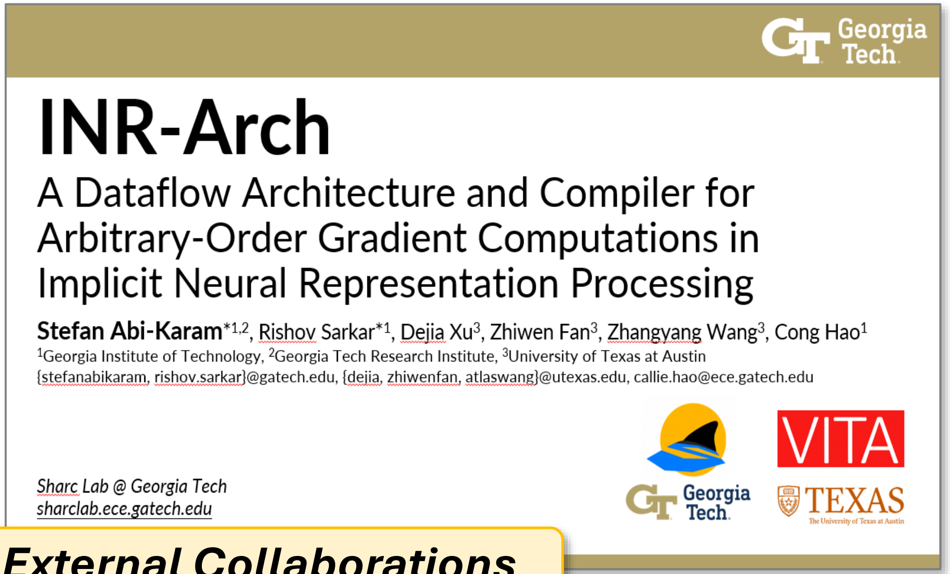
Research Faculty

Ph.D. Student

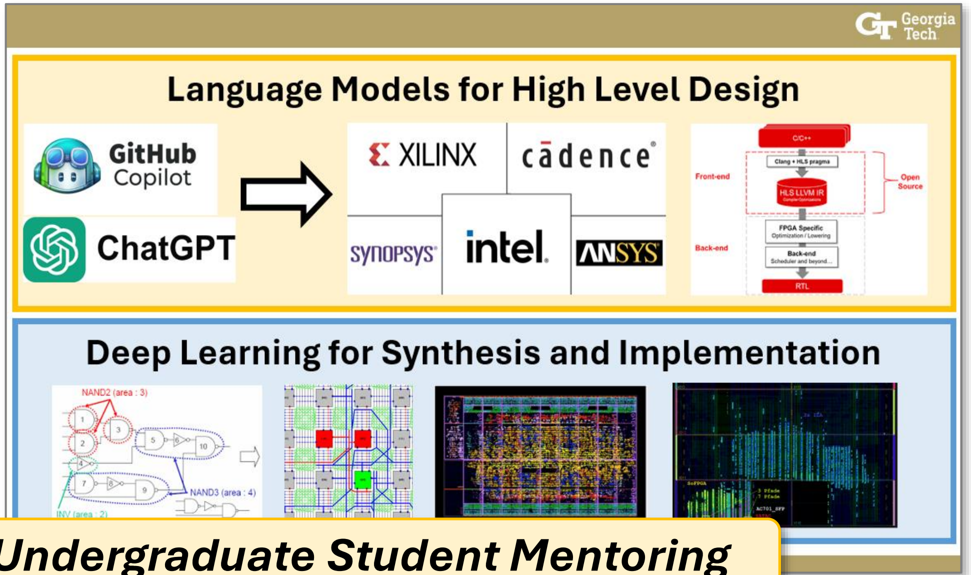
stefanabikaram@gatech.edu

stefanabikaram.com

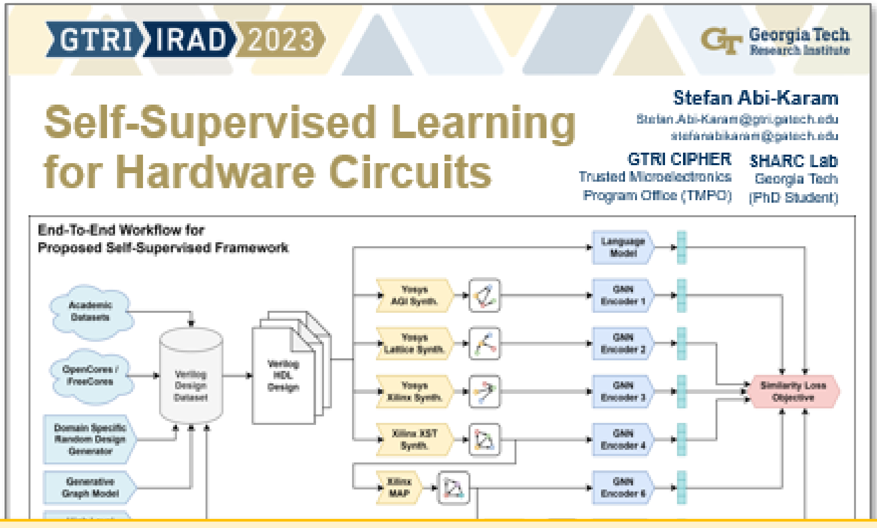
New Research Direction



External Collaborations



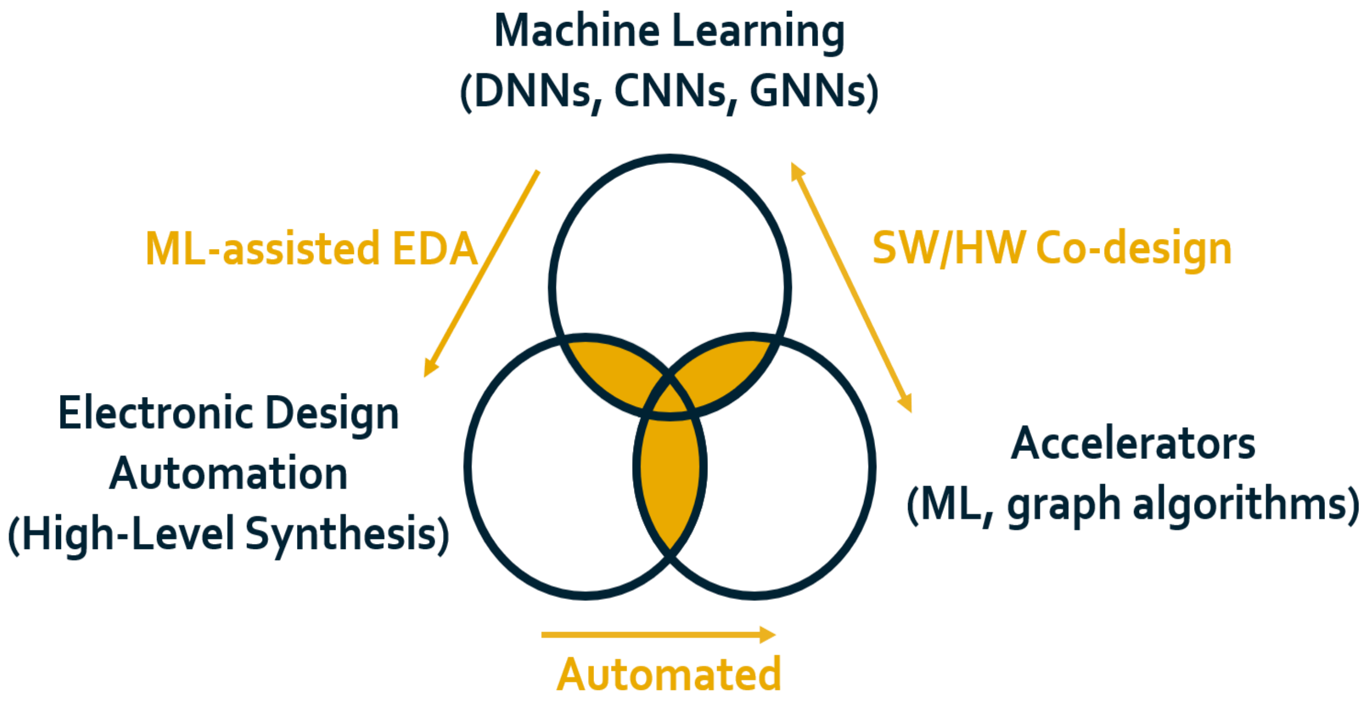
Undergraduate Student Mentoring



Raised $60K Internal Research Funding

Sharc Lab @ Georgia Tech

Prof. Callie Hao

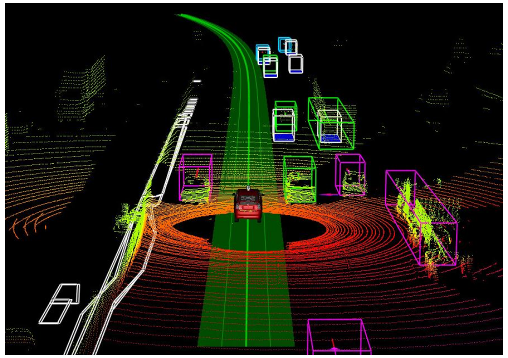


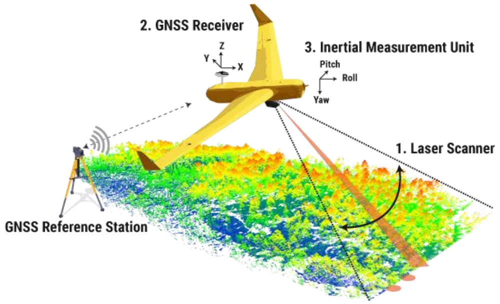
Deep Learning on Graphs

- Physics Modeling
- Scene Graph Understanding
- Source Code Analysis
- NLP
- HW/SW Co-Design
  Smart EDA Tools
  Hardware Security

- Weather Forecasting
- Social Network Analysis
- Recommender Systems
- Drug Discovery
- Fraud Detection
- Many more…

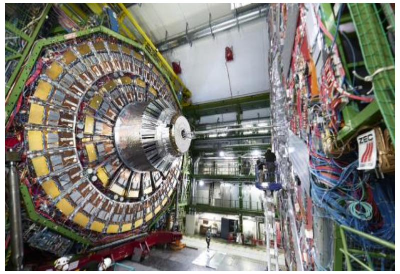
Point Cloud Processing



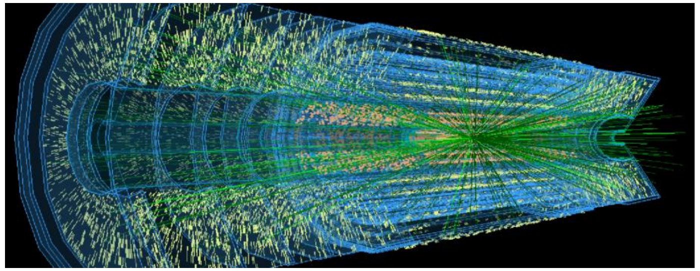


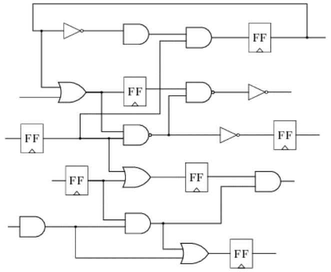
High Energy Physics!

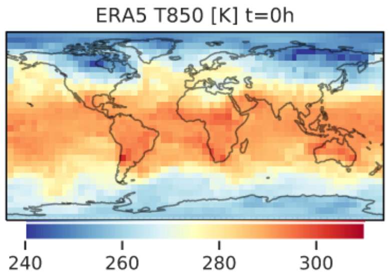
CERN

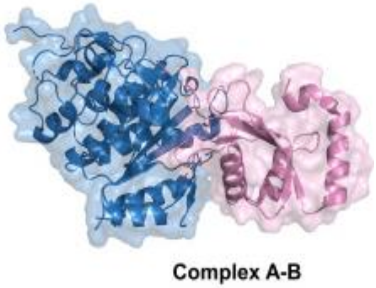


Fermilab









Doesn't Need Inference Acceleration → **Benefits From Inference Acceleration**

Accelerator Performance

- **PyG-CPU**: A PyTorch Geometric CPU model
- **PyG-GPU**: A PyTorch Geometric GPU model
- **CPP-CPU**: A C++ floating point CPU model
- **FPGA-Base**: Proposed HW model with no parallelism
- **FPGA-Parallel**: Proposed HW model with parallelism

We use a batch size of 1 across all evaluations.

For real time inference applications such as high energy physics triggers or autonomous vehicles, this is a reasonable assumption to make. We are unable to batch data in these cases.

If batching is needed, multiple graphs can be treated as one disconnected graph.

All benchmarks share the following model parameters:

GNN Hidden Dim	128
GNN Hidden Layers	6
GNN Output Dim	64
GNN Activation	ReLU
GNN Skip Connections	True
Global Pooling	[add, mean, max]
MLP Head Input Dim	64 x 3
MLP Hidden Dim	64
MLP Hidden Layers	4
MLP Activation	ReLU

Accelerator Performance

- **PyG-CPU**: A PyTorch Geometric CPU model
- **PyG-GPU**: A PyTorch Geometric GPU model
- **CPP-CPU**: A C++ floating point CPU model
- **FPGA-Base**: Proposed HW model with no parallelism
- **FPGA-Parallel**: Proposed HW model with parallelism

CPU: Intel Xeon Gold 6226R
GPU: Nvidia RTX A6000
FPGA: Xilinx Alveo U280 @ 300 MHz.

Evaluated Fixed Point Types

Model Type	Fixed-Point Type
FPGA-Parallel: GCN, SAGE, GIN	ap_fixed<16, 10>
FPGA-Parallel: PNA	ap_fixed<16, 10>
FPGA-Base	ap_fixed<16, 32>

Evaluated Parallelism Factors

Model Type	Input Parallelism	Hidden Parallelism	Output Parallelism
FPGA-Parallel: GCN, SAGE, GIN	gnn_p_in=1, p_in=8	gnn_p_hidden=16, p_hidden=8	gnn_p_out=8, p_out=1
FPGA-Parallel: PNA	gnn_p_in=1, p_in=8	gnn_p_hidden=8	gnn_p_out=8, p_out=1
FPGA-Base	1	1	1

Accelerator Performance

- **PyG-CPU**: A PyTorch Geometric CPU model
- **PyG-GPU**: A PyTorch Geometric GPU model
- **CPP-CPU**: A C++ floating point CPU model
- **FPGA-Base**: Proposed HW model with no parallelism
- **FPGA-Parallel**: Proposed HW model with parallelism

	PyG-CPU	PyG-GPU	CPP-CPU
GCN	6.46x	7.66x	3.04x
GIN	5.81x	6.08x	4.24x
PNA	6.48x	6.70x	22.14x
SAGE	6.58x	7.16x	8.84x
Geo. Mean	6.33x	6.87x	7.08x

Geo. Mean	**6.33x**	**6.87x**	**7.08x**

We achieve consistent speedups overs CPU and GPU implementations!

Based on results from FlowGNN we can roughly **estimate power efficiency**.

Approximate FPGA power: **~30W**
Approximate GPU power: **~70W**

This results in a **~2.33x approximate power efficiency margin** for the same throughput between devices.

Memory efficiency has yet to be profiled.

This is under the current work to support large graphs, HBM support, and graph sampling.

Compiler and Code Generation

We extract model and hardware architecture logic into individual compiler template components.

```
{% if (loop.index - 1) == 0 and (loop.length == 1) %}
{% set emb_buf_in = 'node_emb_first' %}
{% set emb_buf_out = 'node_emb_last' %}
{% elif (loop.index - 1)  == 0 and (loop.length != 1) %}
{% set emb_buf_in = 'node_emb_first' %}
{% set emb_buf_out = 'node_emb_buffer_hidden_0' %}
{% elif (loop.index - 1) == (loop.length-1) %}
{% set emb_buf_in = ( 'node_emb_buffer_hidden_' + ((loop.index%2)|string) ) %}
{% set emb_buf_out = 'node_emb_last' %}
{% else %}
{% set emb_buf_in = ( 'node_emb_buffer_hidden_' + ((loop.index%2)|string) ) %}
{% set emb_buf_out = ( 'node_emb_buffer_hidden_' + (((loop.index+1)%2)|string) ) %}
{% endif %}
```

```
{% if conv.__class__.__name__ == "GCNConv_AGNN" %}
gcn_conv<
    {{max_nodes}},
    {{max_edges}},
    {{sw_model.gnn_layer_sizes()[loop.index - 1][0]}},
    {{sw_model.gnn_layer_sizes()[loop.index - 1][1]}},
    F_TYPE,
    {{num_nodes_guess}},
    {{num_edges_guess}},
    {{degree_guess}},
    {{conv.p_in}},
    {{conv.p_out}}
>(
```

```
>(
    n_nodes,
    n_edges,
    {{emb_buf_in}},
    {{emb_buf_out}},
    edge_list,
    neightbor_table_offsets,
    neighbor_table,
    in_degree_table,
    out_degree_table,
    gnn_convs_{{(loop.index - 1)}}_conv_lin_weight_fixed,
    gnn_convs_{{(loop.index - 1)}}_conv_bias_fixed
);
```

Figure out which double buffer or input / output buffer to use for each layer.

Mapping PyTorch layers and parameters to hardware kernels.