Benchmark Studies of Machine Learning Inference using SOFIE

Lorenzo Moneta, Ioanna Panagou, Sanjiban Sengupta, Neel Shah, Paul Paul Wollenhaupt

CERN

ROOT
Data Analysis Framework
https://root.cern

October 19 - 25, 2024
CHEP 2024

Machine Learning Inference

- Evaluation (Inference) of Machine Learning models is becoming more and more relevant
  - Efficient inference of ML models is critical for production workflows
  - Seamless incorporation of inference into existing systems (e.g., reconstruction, simulation, or analysis software) is needed
  - Requires support for model evaluation directly within C++ code, not only Python
  - Thread management is crucial for utilising models in multi-threaded environments
  - Often models need to be evaluated at the event level (single-batch processing)
  - Important optimising both speed and memory efficiency

Tensorflow and PyTorch

- **Tensorflow** and **PyTorch** provide inference capabilities
  - limited to their model formats
  - using in a C++ environment can be challenging
    - not trivial to use Tensorflow C++ API
    - require some heavy dependence
  - can be difficult to control threads (Tensorflow)
  - often not optimised to desired use cases (e.g. single event evaluatioı
- Torch C++ library (**LibTorch**) is more convenient
  - easier to install
  - support might not be there for all existing extensions (e.g. pytorch geometric or pytorch cluster, which are used for GNN models)
    - some issues encountered in converting models from ONNX to Torch format

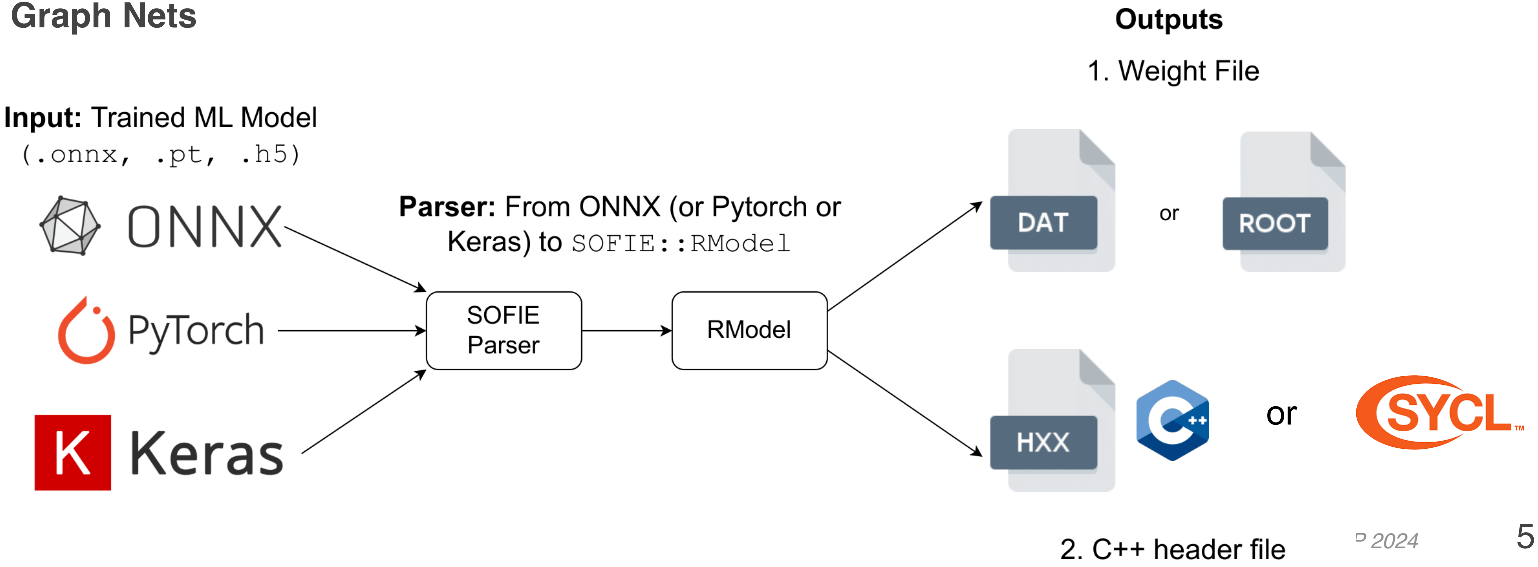
ONNX and ONNXRuntime

- A standard for describing and sharing deep learning models exists:
  - **ONNX** (“Open Neural Network Exchange”)
  - cannot describe all possible deep learning models (e.g. GNN) fully
- **ONNXRuntime**: an efficient inference engine based on ONNX
  - Open source, developed by Microsoft
  - **Can work in both C++ and Python**
  - Supporting both CPU and GPU
    - NVidia GPUs via TensorRT and AMD using ROCm
  - **Has been successfully integrated in the HEP software:**
    - ATLAS and CMS integrated it into their software frameworks:
  - Convenient C++ API
  - Easy control of threads
  - It is based on ONNX input format for trained models
    - conversions exist from Tensorflow and PyTorch
    - not all models can be converted to ONNX format

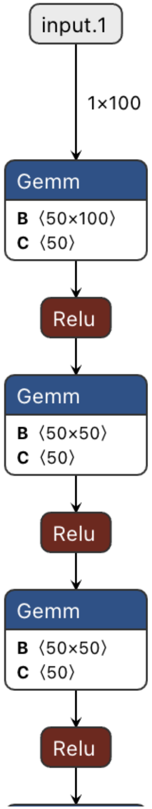
SOFIE

SOFIE : System for Optimised Fast Inference code Emit

- **Input**: trained ML model file
  - **ONNX**: Common standard for ML models
  - **Tensorflow/Keras** and **PyTorch** models (with reduced support than ONNX)
  - Support message passing **GNNs** from DeepMind's **Graph Nets**
- **Output**: **generated C++ code**
  - Easily **invokable directly** from C++ (plug-and-use)
  - **Minimal dependency** (on BLAS only)
  - Can be **compiled at run time** using ROOT Cling JIT and can be **used in Python.**



Code Generation




```
namespace TMVA_SOFIE_Linear_event{

struct Session {

Session(std::string filename ="") {
     // read weight data file
   ……………………..
}
std::vector<float> infer(float* tensor_input1){
……………………..
//——— Gemm
    BLAS::sgemm_(&op_0_transB, &op_0_transA, &op_0_n, &op_0_m, &op_0_k, &op_0_alpha,
tensor_0weight, &op_0_ldb, tensor_input1, &op_0_lda, &op_0_beta, tensor_21, &op_0_n);

//------ RELU
   for (int id = 0; id < 50 ; id++){
      tensor_22[id] = ((tensor_21[id] > 0 )? tensor_21[id] : 0);
   }
……………………..
//——— Gemm
   BLAS::sgemm_(&op_18_transB, &op_18_transA, &op_18_n, &op_18_m, &op_18_k,
&op_18_alpha, tensor_18weight, &op_18_ldb, tensor_38, &op_18_lda, &op_18_beta,
tensor_39, &op_18_n);

// return output
   std::vector<float> ret (tensor_39, tensor_39 + 10);
   return ret;
}
};
}
```

Using the Generated code: in C++

- SOFIE generated code can be easily used in C++

```
#include “Model.hxx”
// create session class
TMVA_SOFIE_Model::Session ses(“model_weights.dat”);
//—- event loop
for (ievt = 0; ievt < N; ievt++) {
   // evaluate model: input is a C float array
   float * input = event[ievt].GetData();
   auto result = ses.infer(input);
   …..
}
```

1. include generated Model header file
2. Create session class (read weight data file)
3. Evaluate the model calling `Session::infer` function

See full Example tutorial code

Using the Generated code: in Python

- Code can be compiled using ROOT Cling and used in Python

```
import ROOT
# compile generate SOFIE code using ROOT interpreter
ROOT.gInterpreter.Declare('#include "Model.hxx"')
# create session class
s = ROOT.TMVA_SOFIE_Model.Session('model_weights.dat')
#-- event loop
…….
# evaluate the model , input can be a numpy array
# of type float32
  result = s.infer(input)
```

Compile at run-time SOFIE generated code using Cling

See full Example tutorial code

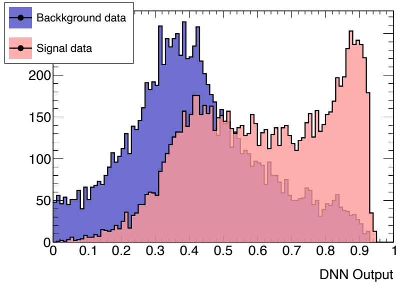
✦ With the RSofieReader class, it is possible to **generate/compile/evaluate** a model in **a single step** directly from the input ONNX file

✦ see RSofieReader tutorial

SOFIE Integration with RDataFrame

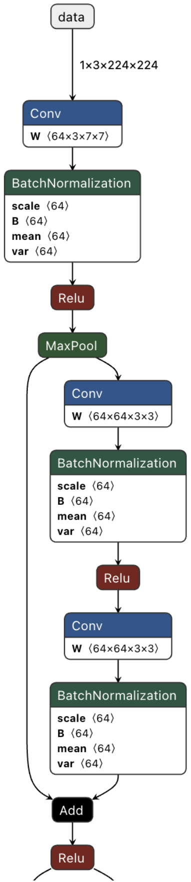
- Have a generic functor class adapting SOFIE model evaluation signature to **`RDF::Define`**: **`SofieFunctor<N,Session>`**
  - supporting multi-thread evaluation, using the RDF slots (using **`RDF::DefineSlot`**)

```
ROOT::RDataFrame df("tree", "inputDataFile.root");
auto h1 = df.DefineSlot("DNN_Value",
SofieFunctor<7,TMVA_SOFIE_higgs_model_dense::Session>(nslots),
{"m_jj", "m_jjj", "m_lv", "m_jlv","m_bb","m_wbb","m_wwbb"}).
Histo1D("DNN_Value");
h1->Draw();
```



See example tutorial code in C++ or Python

ONNX Supported Operators



Operators implemented in ROOT	**CPU**	**GPU**
Perceptron: Gemm	✓	✓
Activations: Relu, Selu, Sigmoid, Softmax, Tanh, LeakyRelu, Swish	✓	✓
Convolution and **Deconvolution** (1D, 2D and 3D)	✓	✓
Pooling: MaxPool, AveragePool, GlobalAverage	✓	✓
Recurrent: RNN, GRU, LSTM	✓	✓
Layer Unary operators: Neg, Exp, Sqrt, Reciprocal, Identity	✓	✓
Layer Binary operators: Add, Sum, Mul, Div	✓	✓
Other Layer operators: Reshape, Flatten, Transpose, Squeeze, Unsqueeze, Slice, Concat, Reduce,	✓	✓
BatchNormalization, LayerNormalization	✓	✓
Operators for GNN/Transformers: TopK, Gather, Range, Tile, If	✓	
Custom operator	✓	

- current CPU support available in **ROOT**
- GPU/SYCL is implemented in a separate dev branch (see ROOT PR)

New developments

GPU Extension of SOFIE

- Extended SOFIE functionality to produce **GPU** code using **SYCL**

```
// generate SYCL code internally
model.GenerateGPU();
// write output header and data weight file
model.OutputGeneratedGPU();
```

model.hxx

```
namespace TMVA_SOFIE_Linear_event{

struct Session {

Session(std::string filename ="") {
   if (filename.empty()) filename =
"Linear_event.dat";
   std::ifstream f;
   f.open(filename);
   // read weight data file
   ………………..
}
std::vector<float> infer(float*
tensor_input1){
```

with SYCL code

```
#include "Model.hxx"

// create session class
TMVA_SOFIE_Model::Session
ses("model_weights.dat");

//-- event loop

for (ievt = 0; ievt < N; ievt++) {

   // evaluate model: input is a C float array

   float * input = event[ievt].GetData();

   auto result = ses.infer(input);
```

Inference code needs to be linked against oneAPI MKL libraries and compiled using SYCL compiler

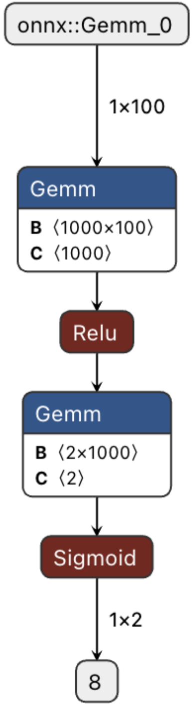
- **Minimise overhead of data transfers** between host and device
- **Manage buffers efficiently, declaring them at the beginning**
- Use libraries for **GPU Offloading**: GPU BLAS from Intel one API and PortBLAS for other GPUs
- **Fuse operators** when possible in a single kernel
- **Replace conditional check** with relational functions

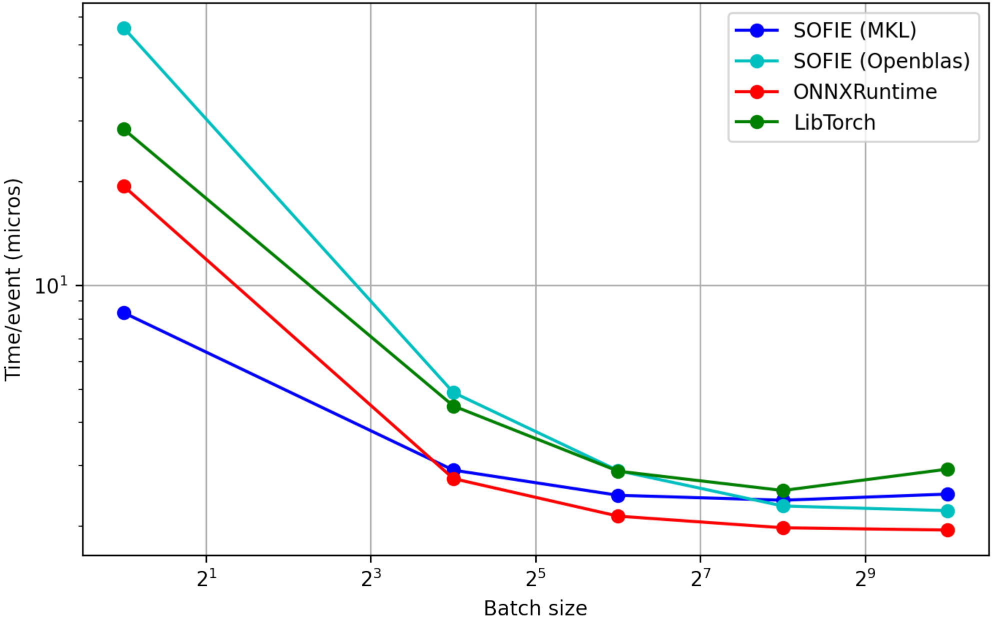
Benchmark Results

- Benchmark ML model evaluation and its memory consumption
  - using CPU from a standard Linux desktop
    - AMD Ryzen 24 threads 4.4 GHz
  - **all tests run in single-thread mode**
- **SOFIE** linked using 2 different BLAS implementations:
  - Openblas
  - Intel MKL (from Intel oneapi)
- **ONNXRuntime** used CPU version 1.19.2
- **LibTorch** used CPU version 2.3.1
- GPU benchmark using SOFIE dev branch with SYCL on NVidia Desktop GPU (RTX 4090)
  - see more in [IWOCL paper]

Linear Model Benchmark

- Test of a simple linear model: Gemm Layer
  - **scaling as a function of batch size**

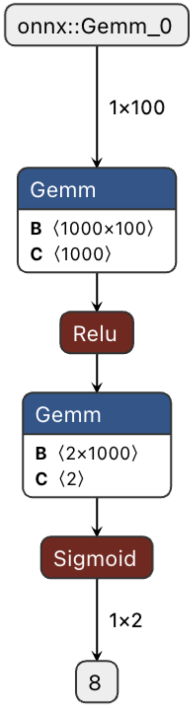




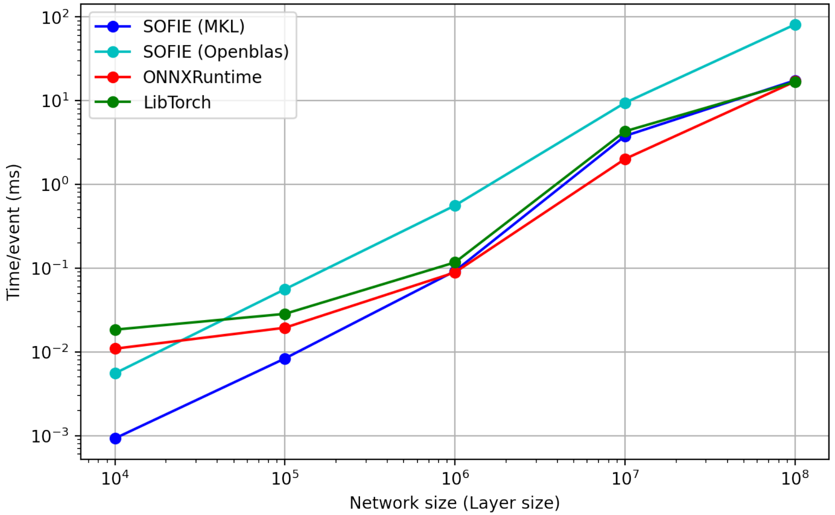
➡ **SOFIE** (with **MKL**) faster for single event evaluation

Linear Model Benchmark (2)

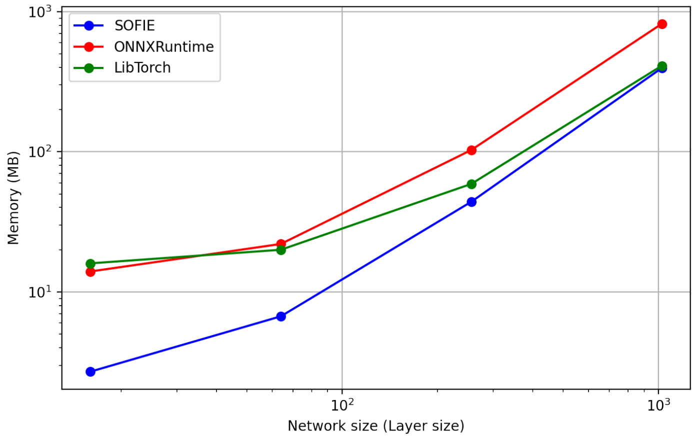
- Test of a simple Linear model
  - **scaling as a function of size (number of input x output of Gemm)**



CPU Time/event



Memory usage

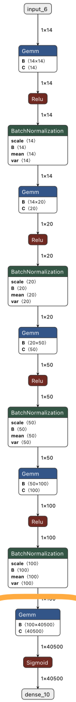


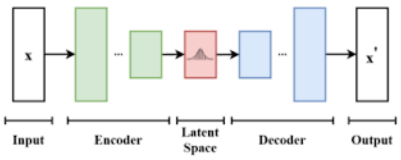
- **SOFIE** performs better at small sizes
  - **Better performances with MKL** implementation of BLAS

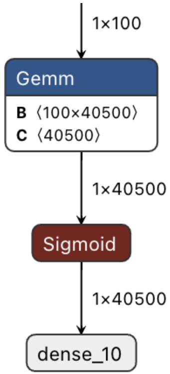
- Memory usage as foreseen for **SOFIE** and **LibTorch**
  - **Less memory in SOFIE for small models**
  - ONNXRuntime uses 2x more.

Benchmark of FastSim Model

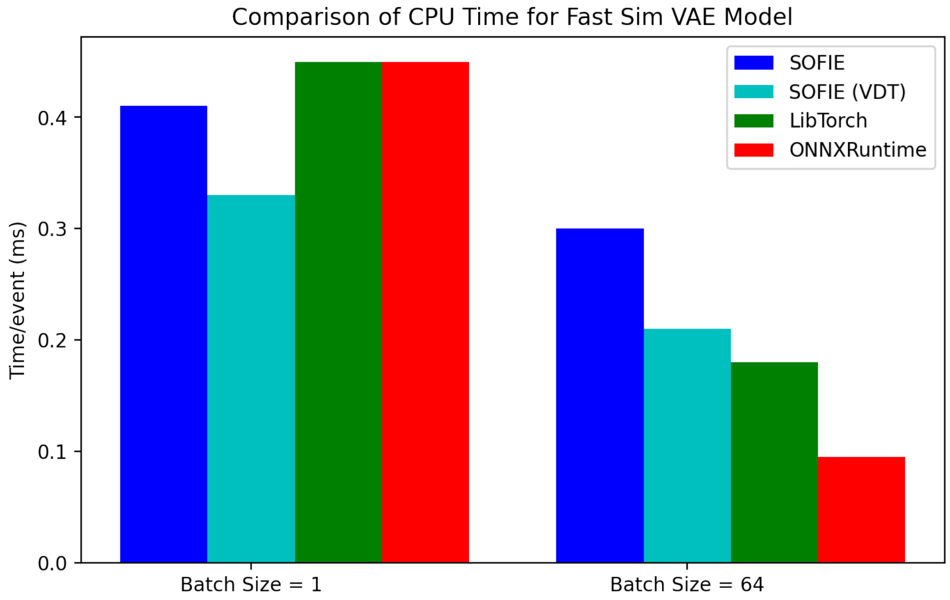
- The decoder of a VAE: model used in Par04 example in Geant4







➡ Benchmark **CPU time**

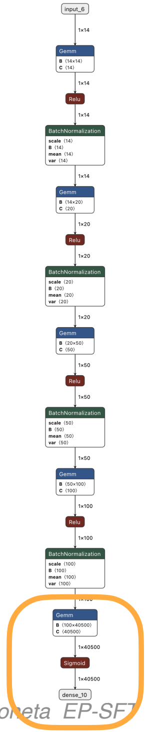


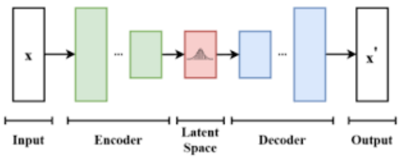
➡ **SOFIE faster for single event**

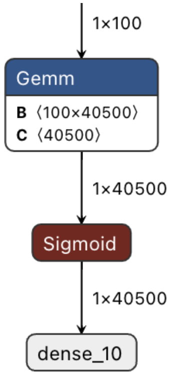
➡ optimised by using **vdt::exp** when evaluating Sigmoid function

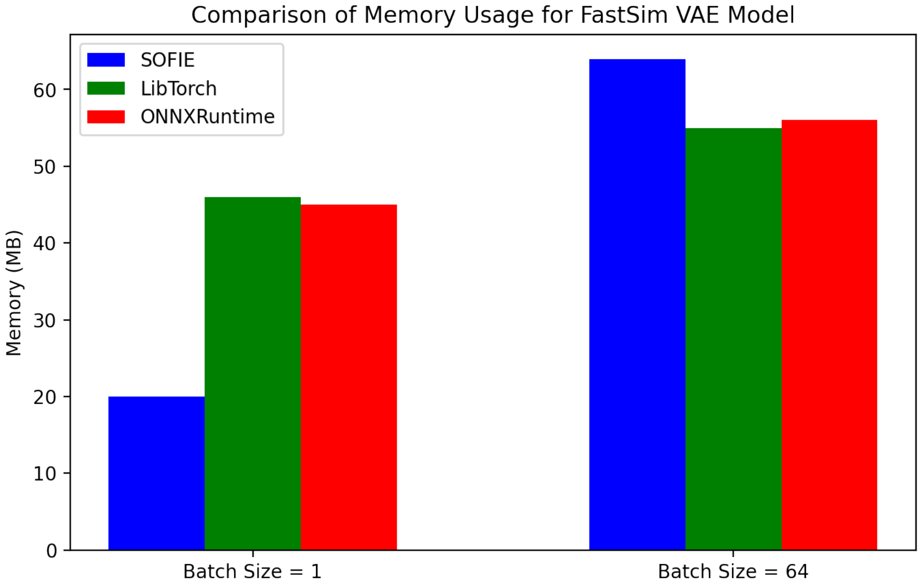
- VAE model used in Par04 example in Geant4

➡ Benchmark **memory usage**









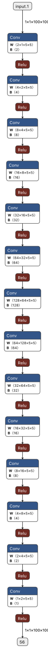
➡ **SOFIE has less memory overhead for small models**

➡ increase memory usage for larger complexity

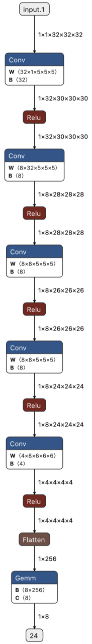
Benchmark: Convolutional models

- **CPU** performance for convolution models
  - 2D and 3D model and a Resnet model

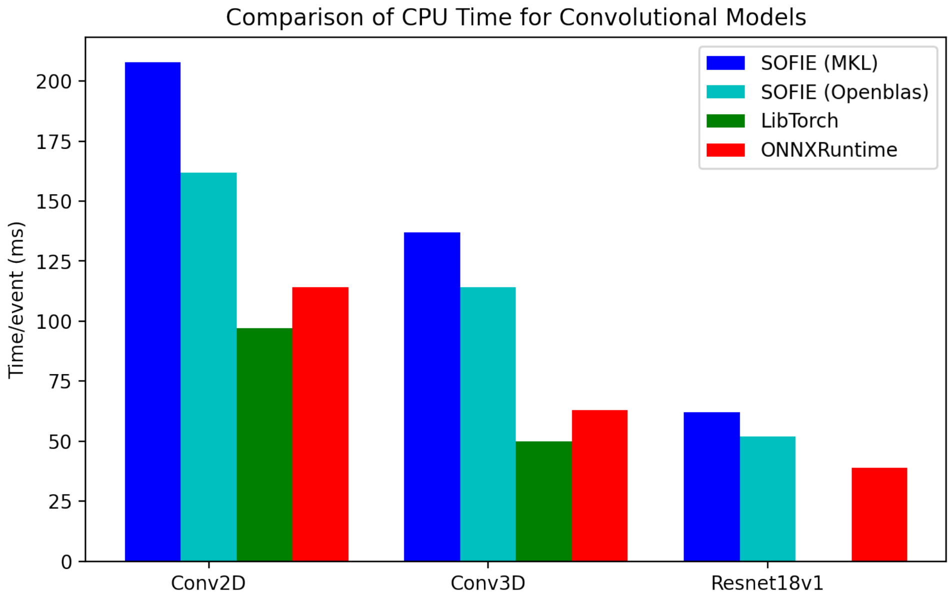
14 layers (100x100)



5 layers (32x32x32)



10 layers (224x224)



➡ **Better performances in LibTorch and ONNXRuntime**

➡ using probably more optimised convolutional kernels

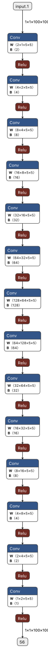
Benchmark: Convolutional models (2)

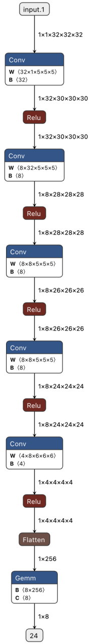
- Memory usage in convolution models
  - 2D and 3D model and a Resnet model

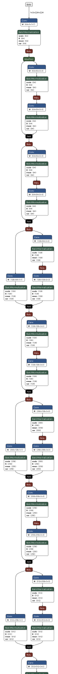
14 layers (100x100)

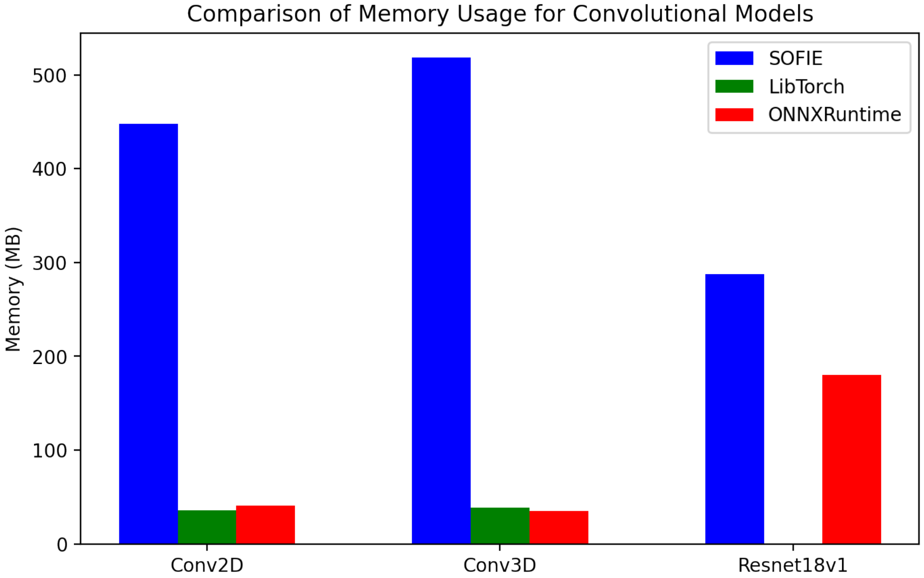
5 layers (32x32x32)

20 layers (224x224)





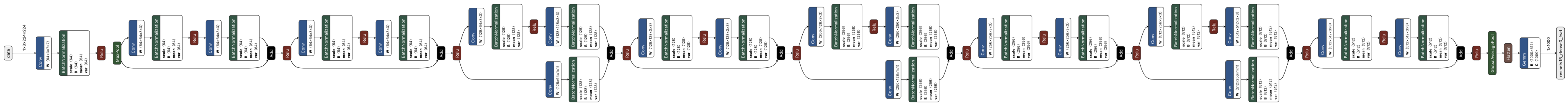


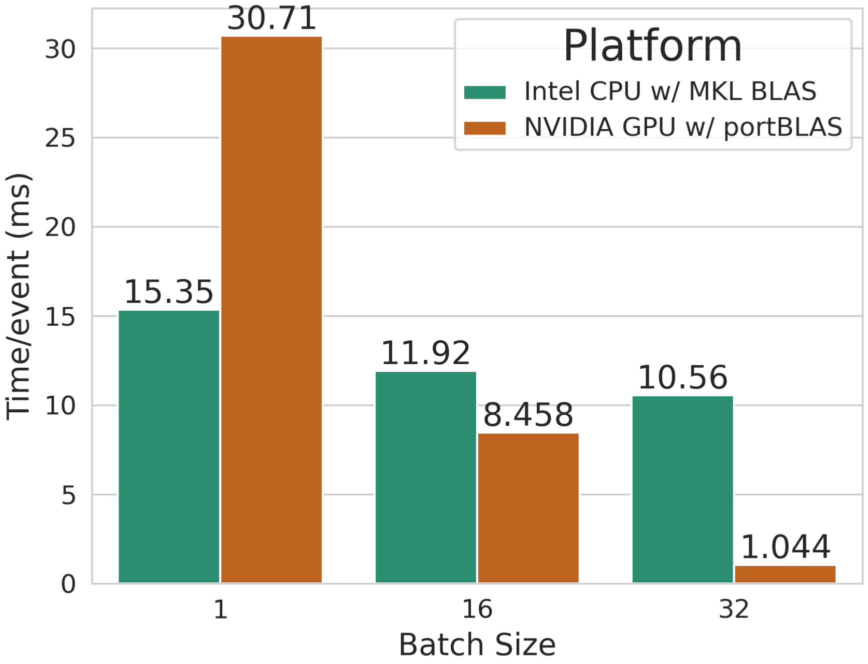


➡ Extensive usage of memory in SOFIE for ConvNets:

➡ no intra-layer optimisations

Benchmark on GPU vs CPU (ResNet)

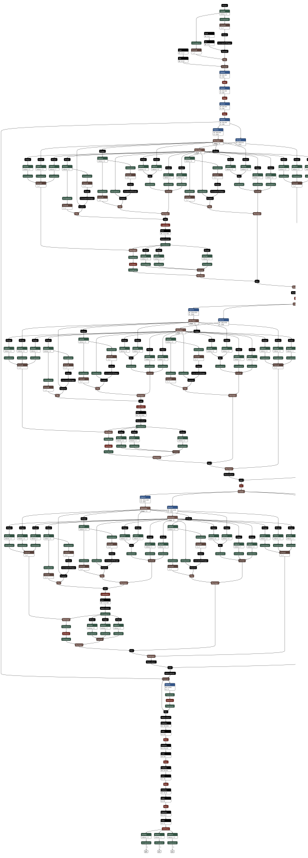




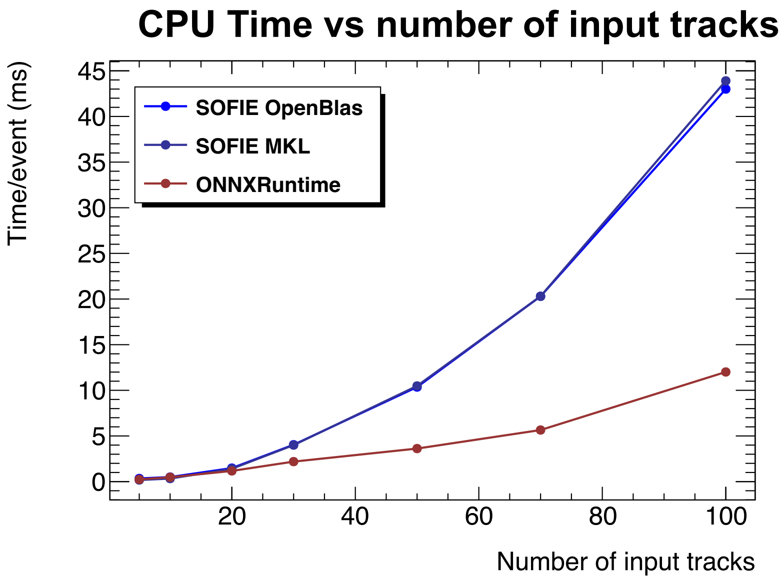
- ✦ Using SYCL GPU implementation for Reset
- ✦ 20 conv. layers with input images of sizes 224x224)
- ✦ Varying Batch size

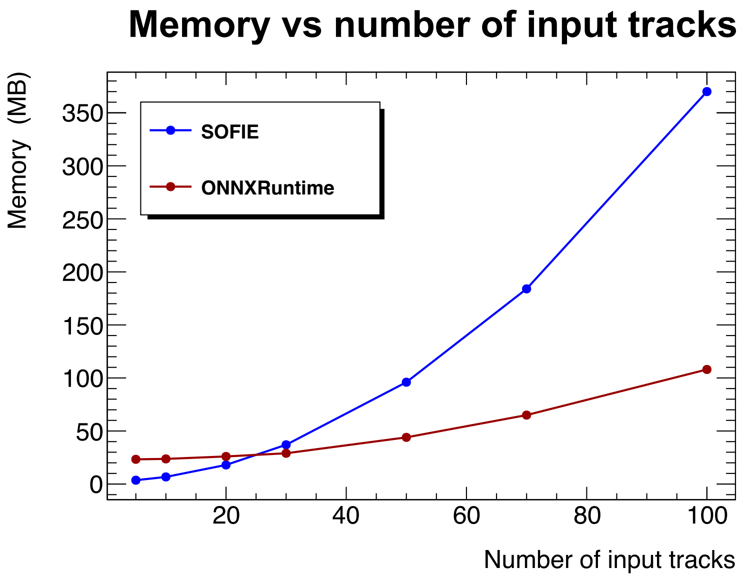
see [IWOCL paper]

Performances for an ATLAS GNN



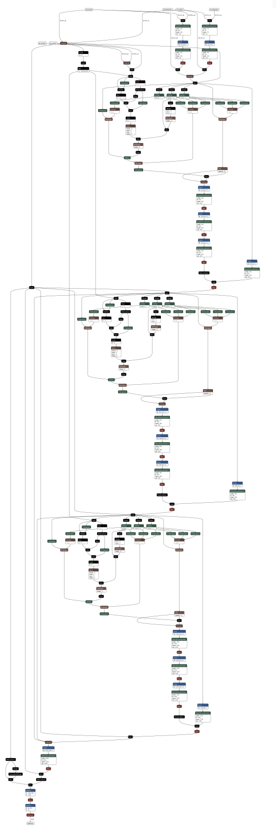
- One of the GNN1 model used for jet tagging
- Measure time and memory vs the number of input tracks
  - input 14 features/track



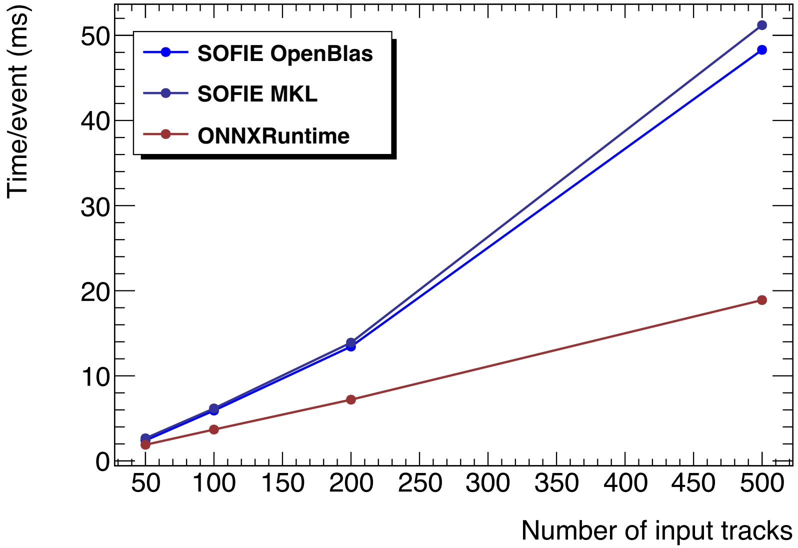


➡ Similar performances for smaller inputs
➡ better scaling in ONNXRuntime

Performances for CMS GNN

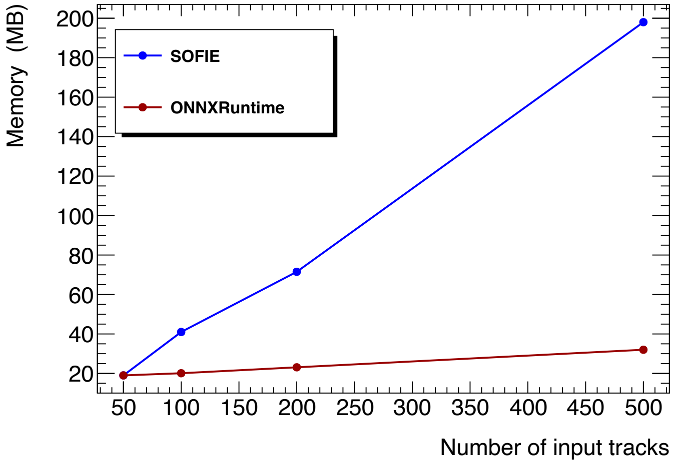


- Particle Net GNN used in CMS for jet flavour tagging
- measure time and memory usage vs the number of input tracks
  - input: 20 features/track



CPU Time vs number of input tracks

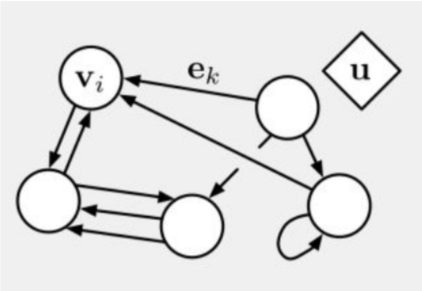
Memory vs number of input tracks



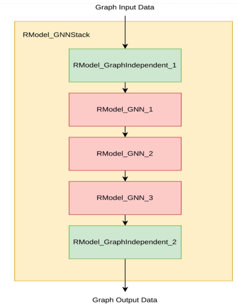
➡ Similar trend as in ATLAS GNN
- ➡ room for memory and CPU optimisation in SOFIE

GNN from GraphNets in SOFIE

- Added SOFIE support for direct parsing of GNN models
    - no need to use the ONNX format, direct parsing from the saved Python model
- Initiated with a network developed by LHCb
  (model for full event interpretation (arXiv:2304.08610)
- Message Passing GNN built and trained using the DeepMind's **Graph Nets** library
- Important to have an efficient implementation with minimal dependencies
- **Support for a dynamic number of nodes/edges**
- User can customise architecture to combine different GNN blocks

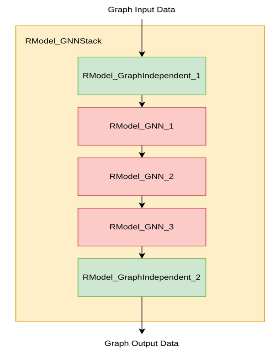


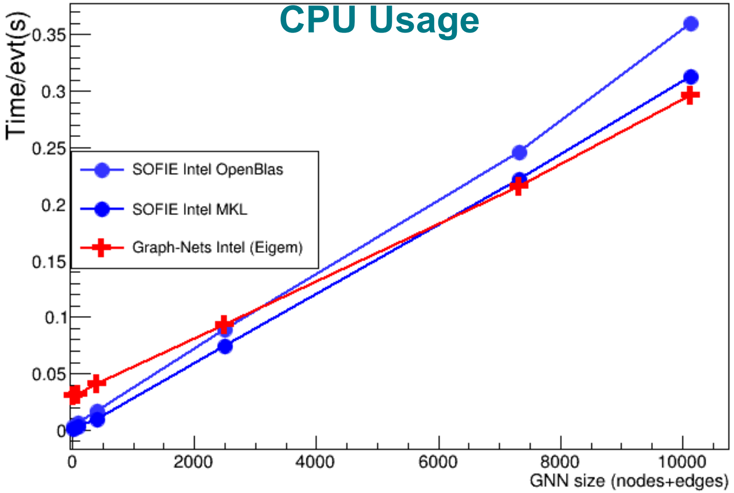
RModel_GNN

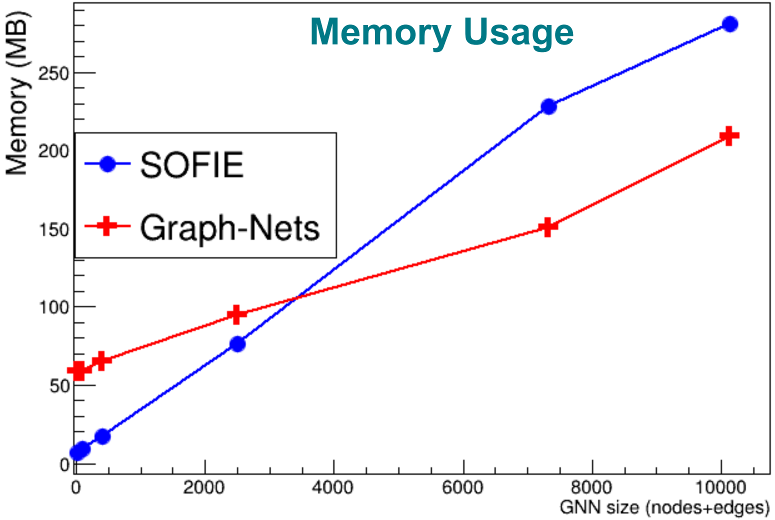


Benchmark of GraphNets GNN

- Test inference performance of a toy architecture from LHCb
  - scaling number of nodes and edges







- 10 faster for small GNN size
- comparable for large GNN
  - similar performances since dominated by matrix operations

- No optimisation for memory has been done so far in SOFIE.

Summary of Benchmarks

- **SOFIE** benchmark results demonstrate:
  - Faster inference for event-level evaluations
  - Lower memory consumption in smaller models
  - Larger model sizes lead to reduced performance
- Memory usage in SOFIE has not been optimised for multi-layer models
  - Fusions of the operator have not been implemented yet from ONNX model
  - Better scaling for models parsed from GraphNets
    (overhead in splitting model with large number of operators as done in ONNX
- **LibTorch** and **ONNXRuntime** show similar performances (CPU and memory, with smaller memory usage in LibTorch)
  - Less flexibility in converting models from ONNX format to Torch

Conclusions

- **SOFIE**, an **easy-to-use** inference engine for Deep Learning models, is available
  - Supporting several ONNX operators, including some production models from experiments (e.g complex GNNs)
  - Supporting also GNNs based on GraphNets (cannot be easily converted to ONNX)
  - Integrated with other ROOT tools (RDataFrame) for ML inference in end-user analysis
  - **Simple to use in C++ and Python**
  - Give fully control to users of the generated code and no additional dependency needed
  - SOFIE can also be used for storing models (and weights) in ROOT format
  - A prototype implementation for GPU using SYCL has been developed
- Future developments according to user needs
  - Plans to implement memory optimisations and fusion of ONNX operators
  - Extend GPU support (porting to CUDA and/or ALPAKA if interest by experiments)

SOFIE References

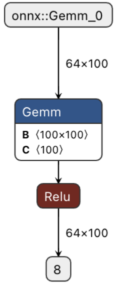
- SOFIE in ROOT GitHub:
  - https://github.com/root-project/root/tree/master/tmva/sofie)
- Example notebooks on using SOFIE:
  - https://github.com/lmoneta/tmva-tutorial/tree/master/sofie
- Tutorials are also available in the tutorial/tmva directory
- Link to SOFIE code in current ROOT master in GitHub
- Link to PR implementing SOFIE to SYCL code generation
- Link to benchmarks in *rootbench*

Backup Slides

Parsing input models

- Parser: from ONNX to `SOFIE::RModel` class
  - `RModel`: intermediate model representation in memory

```
using namespace TMVA::Experimental::SOFIE;
RModelParser_ONNX parser;
RModel model = parser.Parse("model.onnx");
```



- Parser exists also for (with more limited support)
  - Native PyTorch files (*model.pt* files)

```
SOFIE::RModel model = SOFIE::PyTorch::Parse("PyTorchModel.pt");
```

  - Native Keras files (*model.h5* files)

```
SOFIE::RModel model = SOFIE::PyKeras::Parse("KerasModel.h5");
```

Code Generation

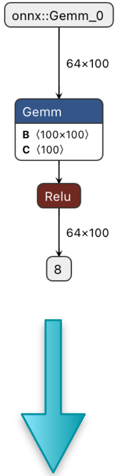
- **Parser**: from ONNX to `SOFIE::RModel` class
  - **`RModel`**: intermediate model representation in memory

```cpp
using namespace TMVA::Experimental::SOFIE;
RModelParser_ONNX parser;
RModel model = parser.Parse("Model.onnx");
```

- **Code Generation**: from **`RModel`** to a **C++ file** (`Model.hxx`) and a weight file (`Model.dat`)

```cpp
// generate text code internally
model.Generate();
// write output header file and data weight file
model.OutputGenerated();
```

- Generated code has minimal dependency
  - only linear algebra library (BLAS) and no ROOT dependency
  - can be easily integrated in your project

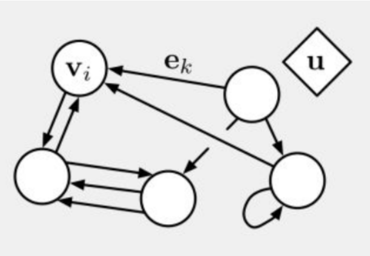


C++ code

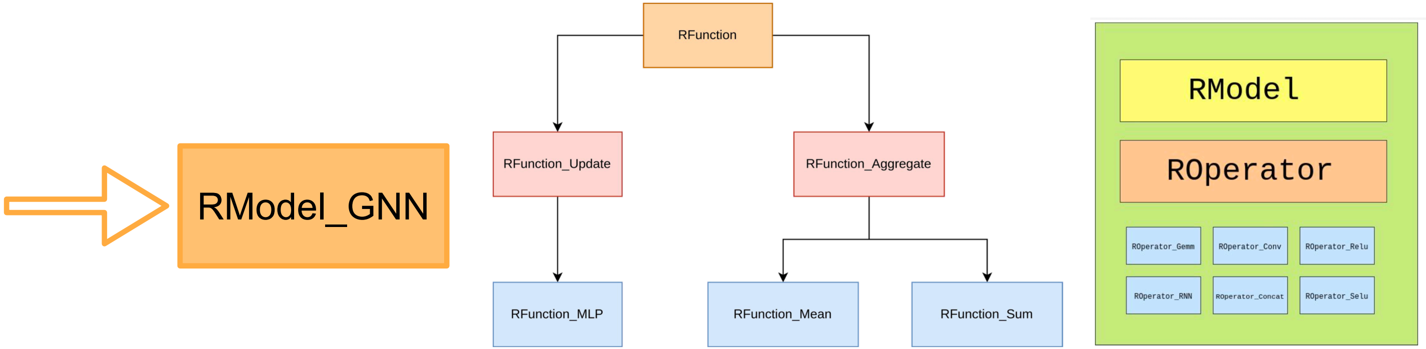
```cpp
namespace TMVA_SOFIE_Linear_event{

struct Session {

Session(std::string filename ="") {
   if (filename.empty()) filename = "Linear_event.dat";
   std::ifstream f;
   f.open(filename);
   // read weight data file
   ………………..
}
std::vector<float> infer(float* tensor_input1){
```

SOFIE GNN Support

- Developed **C++ classes** for representing **GNN structure**.
  - based on SOFIE **`RModel`** and the **`ROperator`** classes developed for supporting ONNX.
  - SOFIE classes provide the functionality to generate C++ inference code
- **Python code** (based on PyROOT) for initialising SOFIE classes from the Graph Nets models

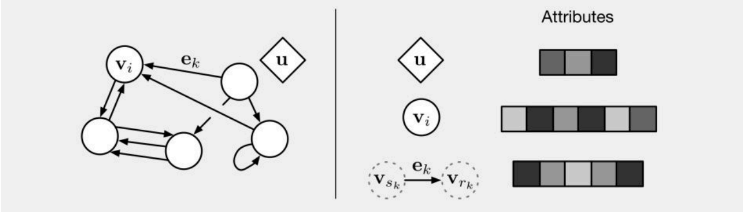


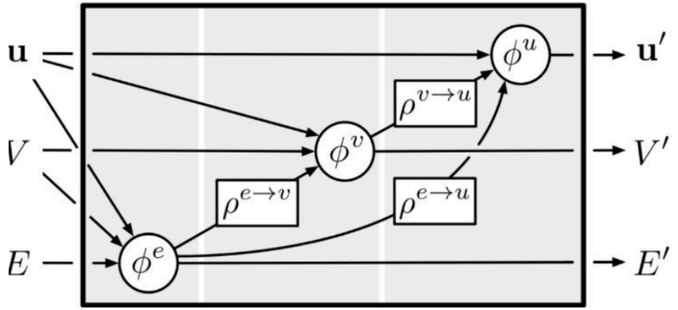
Graph Nets GNN



GNN Support

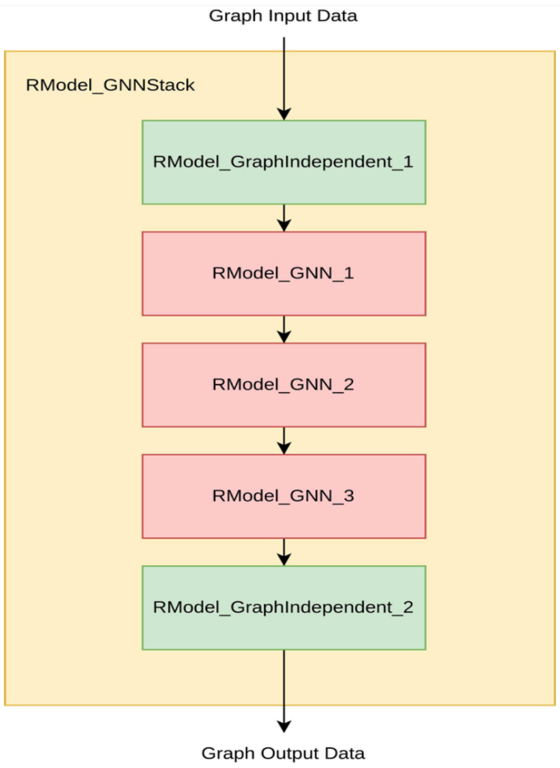
- Follow **Graph Nets** architecture
  - A model is described by
    - number of nodes and edges
    - sender/receiver list of edges
    - number of features (for node, edge and global)
  - Updating functions on node, edge and global features
    - MLP (Multi-Layer Perceptron)
      - including activation functions and layer normalisation
    - Aggregation functions
      - Mean, Sum,…

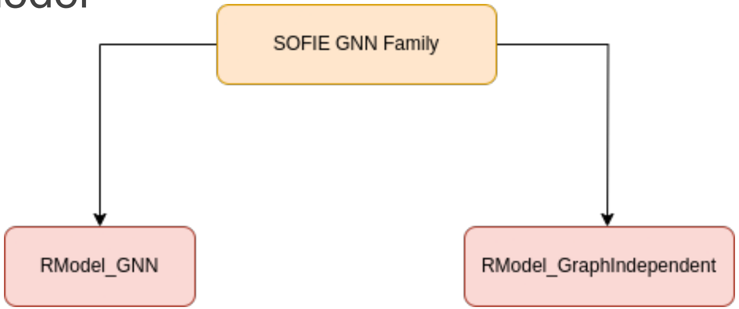




GNN Inference

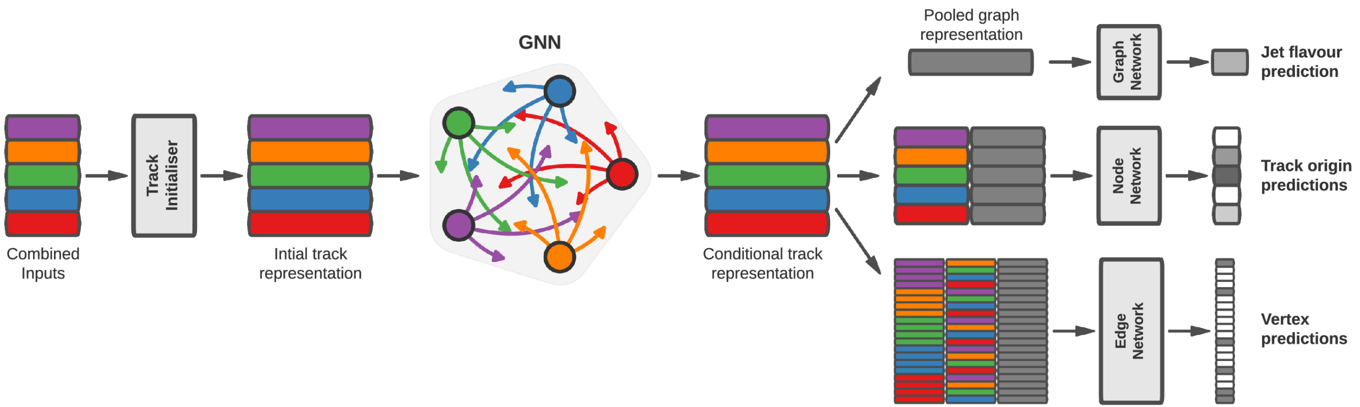
- Final model is composed by several blocks chained together
  - SOFIE can generate C++ code for each single GNN block
  - a C++ `struct` of `RTensor`'s represents the GNN data flowing trough the model
  - Users can stuck the GNN blocks according to the desired architecture in the inference function for the full model



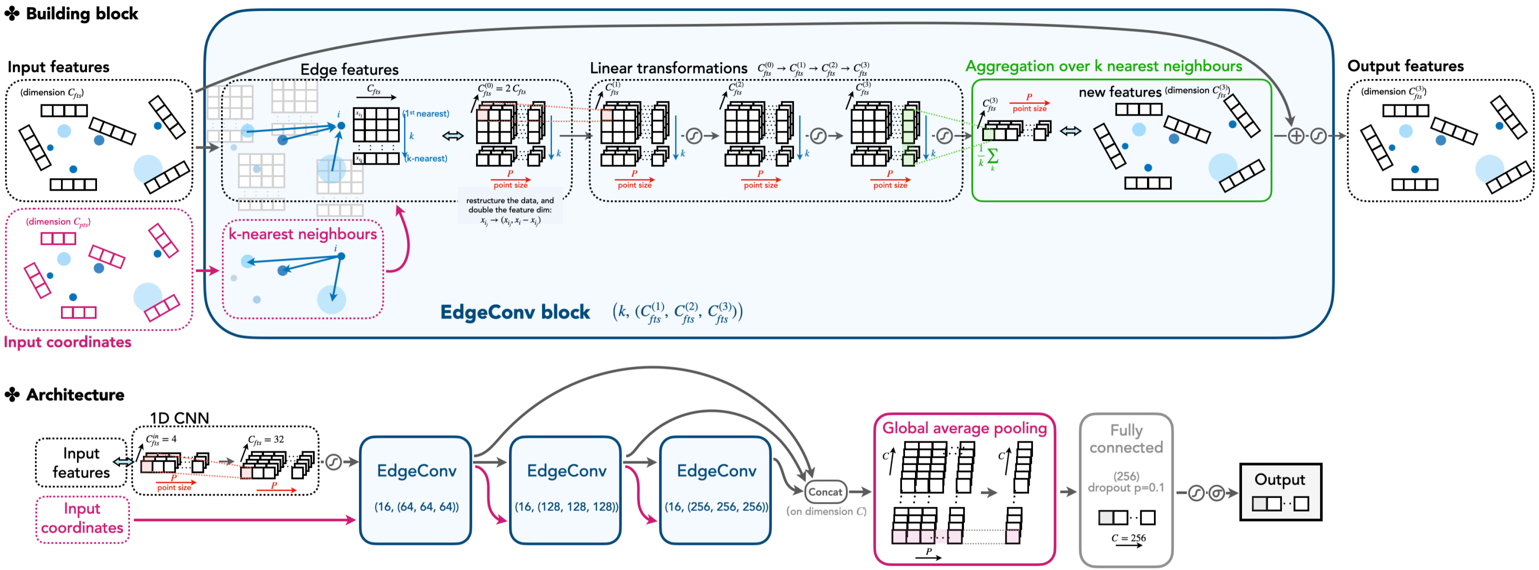


ParticleNet Architecture

- The architecture of ATLAS GNN (see ATL-PHYS-PUB-2022-027)



- The architecture of Particle Net based on DGCNN and EdgeConv ([see link](#))



Performance on CPU vs GPU