Accelerating Machine Learning Inference on GPUs with SYCL using SOFIE

Lorenzo Moneta, Ioanna Panagou, Sanjiban Sengupta

Speaker: V. E. Padulano

ACAT 2024

ROOT

Data Analysis Framework

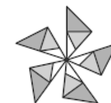
https://root.cern

Machine Learning Inference

- Fast Evaluation of Machine Learning models is more and more relevant
- ML tools like Tensorflow/PyTorch have functionality for inference
  - can run only for their models
  - usage in a C++ environment can be cumbersome (API, dependencies etc.)
- A standard for describing deep learning models:
  - **ONNX** (“*Open Neural Network Exchange*”)
  - cannot describe all possible deep learning models (e.g. GNN) fully

ONNX

- ONNXRuntime: an efficient inference engine based on ONNX
  - can work in both C++ and Python
  - supporting both CPU and GPU
  - can be challenging to integrate in the HEP ecosystem
    - control of threads, dependencies, etc..
    - not optimised for single-event evaluation



ONNX RUNTIME

Idea for Inference Code Generation

- An inference engine that…
  - **Input**: trained ONNX model file
    - Common standard for ML models
    - Supported by PyTorch natively
    - Converters available for Tensorflow and Keras
  - **Output**: C++ code of the inference function
    - Easy integration in other C++ projects
    - Minimal dependency (on BLAS only)
    - Can be compiled on the fly using Cling JIT

- **SOFIE : S**ystem for **O**ptimised **F**ast **I**nference code **E**mit

SOFIE

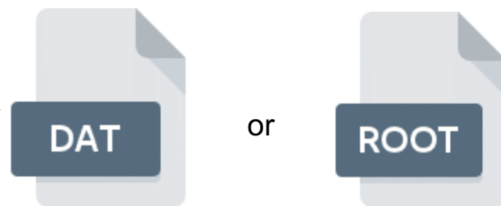
Input: Trained ML Model
`(.onnx, .pt, .h5)`

Parser: From ONNX (or Pytorch or Keras) to `SOFIE::RModel`

SOFIE Parser → RModel

Outputs

1. Weight File



2. C++ header file

Code Generation

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

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

- **Code Generation**: from **`RModel`** to a **C++ header** and a weight file

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

C++ code

```
namespace TMVA_SOFIE_Model{

struct Session {

  Session(std::string filename) {
    ………………..
  }
  std::vector<float> infer(float* input)
  {
     ……………….
    //— implementation of all operators
    ……………….

    return output_tensor;
   }
};
}
```

weight files

Generated code depends only on BLAS (no ROOT)

Using the Generated code: in C++

- SOFIE generated code can be easily used in compiled C++ 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
   auto input = GetInput();
   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 C++ interpreter or 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

SOFIE Integration with RDataFrame

- **SOFIE Inference** code provides a Session class with this signature:

```
vector<float> ModelName::Session::infer(float* input);
```

- **RDataFrame**(RDF) interface requires a functor with this signature:

```
FunctorObj::operator()(T x1, T x2, T x3,….);
```

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

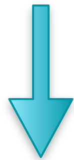
```
ROOT::RDataFrame df("tree", “inputDataFile.root”);
auto h1 = df.DefineSlot("DNN_Value",
SofieFunctor<7,TMVA_SOFIE_higgs_model_dense::Session>(nslots),
{"m_jj", "m_jjj", “m_lv", “m_jlv","m_bb","m_wbb","m_wwbb"}).
Histo1D(“DNN_Value”);
h1->Draw();
```

See full Example tutorial code in C++ or Python

GPU Extension of SOFIE

- Extend 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

What is SYCL ?

- SYCL is a single-source, high-level, standard C++ programming model
- can target a wide range of heterogeneous platforms (CPUs, GPUs, FPGAs)

One-MKL
One-DNN
OneDPC
SYCL-BLAS
SYCL-Eigen
SYCL-DNN
SYCL Parallel STL
...

C++ Libraries

Standard C++ Application Code

ML Frameworks

TensorFlow

Complex ML frameworks can be directly compiled and accelerated

C++ Template Libraries

C++ Template Libraries

C++ Template Libraries

C++ templates and lambda functions separate host & accelerated device code

C++ Kernel Fusion can give better performance on complex apps and libs than hand-coding

SYCL

SYCL Compiler

CPU Compiler

LLVM

GCC

Visual C++

OpenCL

Other Backends

CPU

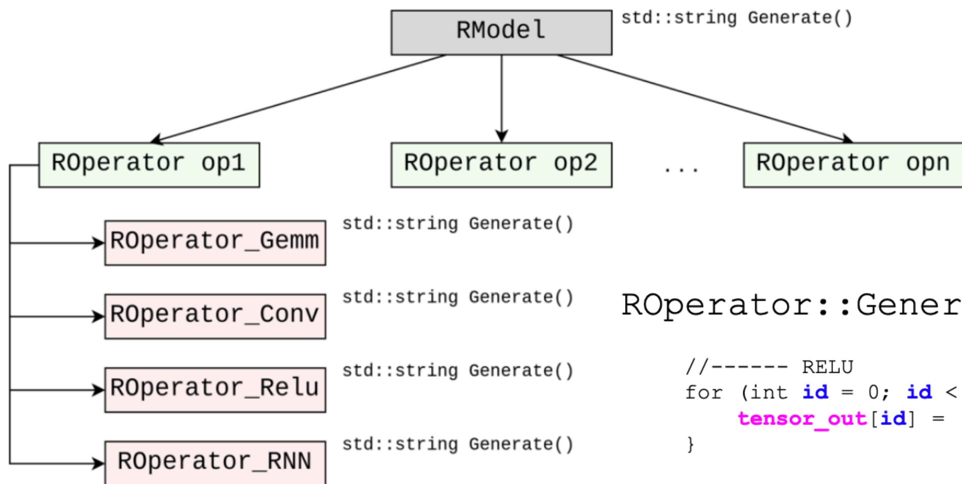
GPU | CPU | DSP | FPGA

AI/Tensor HW | Custom Hardware

Accelerated code passed into device OpenCL compilers

SYCL is ideal for accelerating C++ based engines and applications with performance portability

C++ to SYCL



`RModel::GenerateGPU` **generates code for:**

- **Device Selection**
- **Queue Initialization**
- **Host & Device storage**

`ROperator::GenerateGPU` **generates code for the kernel execution**

```
//------ RELU
for (int id = 0; id < length ; id++){
    tensor_out[id] = ((tensor_in[id] > 0 )? tensor_in[id] : 0);
}
```

```
//------ RELU
q.submit([&](cl::sycl::handler &cgh){
    auto acc_tensor_in = cl::sycl::accessor{buf_tensor_in, cgh, cl::sycl::read_only};
    auto acc_tensor_out = cl::sycl::accessor{buf_tensor_out, cgh, cl::sycl::write_only,
    cl::sycl::no_init};
    cgh.parallel_for<class op_17>(cl::sycl::range<1>(length), [=](cl::sycl::id<1> id){
        acc_tensor_out[id] = cl::sycl::max(acc_tensor_in[id], 0.0f);
    });
```

GPU Implementation

Performance considerations

- Minimise overhead of data transfers between host and device
  - implement all on GPU and transfer data only at the beginning and at the end of the computation
- Manage buffers efficiently, declaring them at the beginning
- Use libraries for GPU Offloading:
  - GPU BLAS implementation from Intel oneAPI and portBLAS for other GPUs
- Fuse operators when possible (e.g. a layer op. with activation) in a single kernel
- Whenever possible, avoid code branches

ONNX Supported Operators

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

- current CPU support available in ROOT 6.30
- GPU/SYCL is implemented in a ROOT PR

CPU Benchmark for Different Models

- Test event performance of SOFIE vs ONNXRuntime (using batch size = 1)

Performance on CPU vs GPU

Linear_64

Conv_d100_L14_B32

resnet18v1_B1

Time/event (ms)

10^3
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}

Linear_64: 0.001, 0.002, 0.006, 0.035, 0.15, 0.16

Conv_d100_L14_B32: 8.7, 16.7, 722, 2.8, 10.6, 4.8

resnet18v1_B1: 6.8, 15.4, 415, 30.7, 138, 135

Platform

- Intel CPU + ONNXRUNTIME (32 Threads)
- Intel CPU + C++ w/ MKL BLAS
- Intel CPU + C++ w/ NETLIB
- NVIDIA GPU + SYCL w/ portBLAS
- Intel GPU + SYCL w/ portBLAS
- Intel GPU + SYCL w/ MKL BLAS

Performance on GPU vs CPU (ResNet)

Batch Size	Intel CPU w/ MKL BLAS	NVIDIA GPU w/ portBLAS
1	15.35	30.71
16	11.92	8.458
32	10.56	1.044

Time/event (ms)

Using ResNet Model (rather heavier model, > 10 conv. layers with image sizes ~ 200x200)

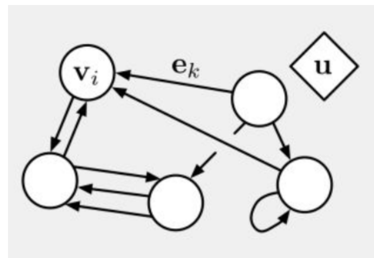
Varying Batch size

SOFIE for Graph Networks

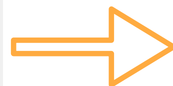
- Added SOFIE support GNN models
- Initiated with a network developed by LHCb:
  - Message Passing GNN built and trained using DeepMind's **Graph Nets** library
    - model plan to be used in LHCb trigger using full event interpretation (*see ACAT2024* contribution)
    - important to have efficient implementation and with minimal dependencies
- Available now in ROOT master
  - support for a dynamic number of nodes/edges

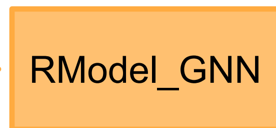
SOFIE GNN Support

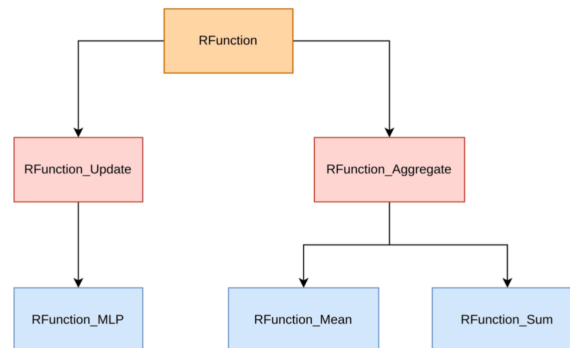
- Developed **C++ classes** for representing **GNN structure**.
  - based on SOFIE **`RModel`** and the **`ROperator`** classes developed for supporting ONNX.
  - SOFIE classes provide the functionality to generate C++ inference code
- **Python code** (based on PyROOT) for initialising SOFIE classes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

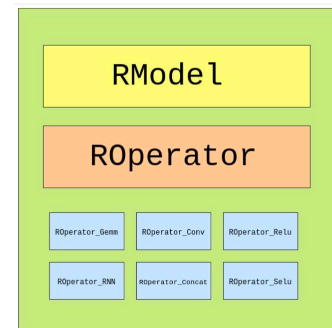


Graph Nets GNN



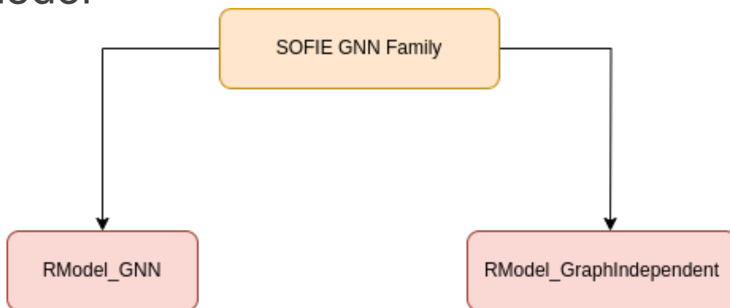


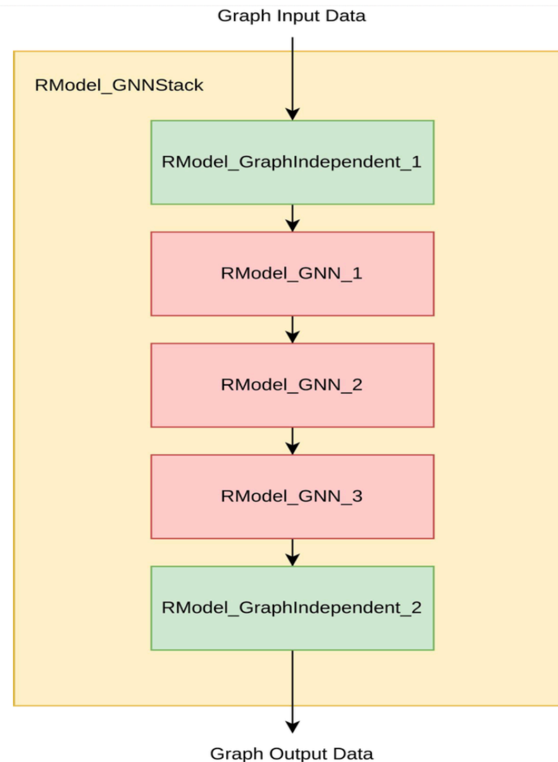




GNN Inference

- 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 stack the GNN blocks according to the desired architecture in the inference function for the full model

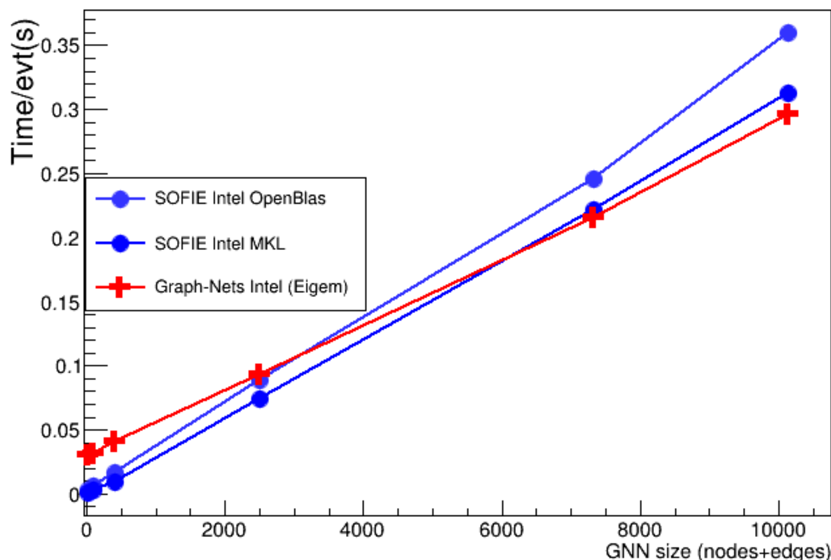




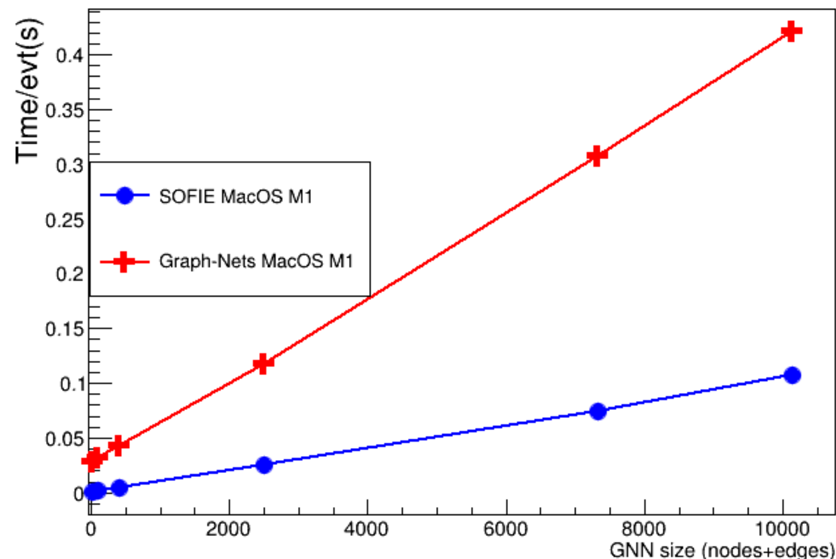
Benchmark of SOFIE GNN

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

Intel Linux Desktop



MacOS M1



Summary

- **SOFIE**, fast and easy-to-use inference engine for Deep Learning models, is available in ROOT
  - Integrated with other ROOT tools (*RDataFrame*) for ML inference in end-user analysis
  - Support for several ONNX operators and also GNN
  - A prototype implementation using SYCL has been developed
    - Plan to extend to CUDA and/or ALPAKA
    - Could also fit in the context of experiment GPU trigger systems
- Future developments according to user needs and the received feedback
  - Supporting production model from experiments (GNN and transformers)

Example Notebooks and Tutorials

- 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

Benchmark settings

- GPUs
  - NVIDIA GeForce RTX 4090
  - Intel Arctic Sound-P
- CPU: Intel 16C/32T @5GhZ

SYCL Implementations

Implementations under developments:

- Codeplay
- Intel OneAPI Data Parallel C++ (DCP++)
- AdaptiveCpp
- neoSYCL
- triSYCL

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

SYCL Source Code

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute

ISO

intel

oneAPI

DPC++
Uses LLVM/Clang
Part of oneAPI

Any CPU

NVIDIA CUDA
NVIDIA GPUs

ROCm
AMD GPUs

OpenCL SPIR
Intel CPUs
Intel GPUs
Intel FPGAs

Level Zero
Intel GPUs

codeplay

ComputeCpp

ComputeCpp
Multiple Backends

Any CPU

Experimental

OpenCL NVIDIA
NVIDIA GPUs

OpenCL SPIR
Intel CPUs
Intel GPUs
Intel FPGAs
AMD GPUs
(depends on driver stack)
Arm Mali
IMG PowerVR
Renesas R-Car

AdaptiveCpp

UNIVERSITÄT HEIDELBERG

AdaptiveCpp
Multiple Backends

OpenMP
Any CPU

Experimental

NVIDIA CUDA

ROCm
AMD GPUs

Level Zero
Intel GPUs

SYCL Application Code Structure

```
#include <CL/sycl.hpp> // header file
namespace sycl = cl::sycl;

int main() {
  // host storage (a = b * 2)
  std::vector<float> a = {1.0, 2.0, 3.0, 4.0};
  std::vector<float> b = {0.0, 0.0, 0.0, 0.0};
  auto length = a.size();
  sycl::default_selector device_selector; // device selector
  sycl::queue queue(device_selector); // queue

  { // begin scope
    // device storage
    sycl::buffer<float, 1> a_buf(a.data(), sycl::range<1>(length);
    sycl::buffer<float, 1> b_buf(b.data(), sycl::range<1>(length);

    // kernel execution
    queue.submit([&] (sycl::handler& cgh) {
      auto a_acc = a_buf.get_access<sycl::access::mode::discard_write>(cgh);
      auto b_acc = b_buf.get_access<sycl::access::mode::read>(cgh);
      cgh.parallel_for<class op>(sycl::range<1>(length), [=] (sycl::id<1> id) {
        a_acc[id] = b_acc[id] * 2;
      });
    });
  } // end scope
  return (0);
}
```

Include SYCL header

Setup Host Storage

Initialize Device Selector

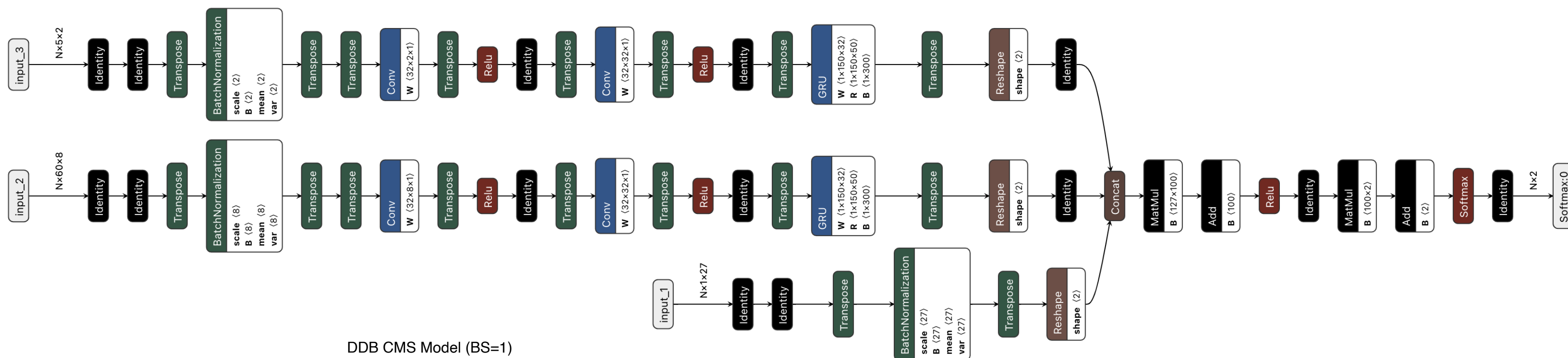
Initialize Device Queue

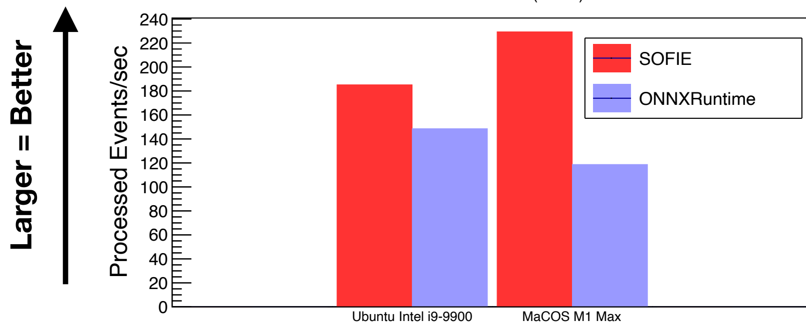
Setup Device Storage

Execute Kernel

Benchmark using a CMS Model

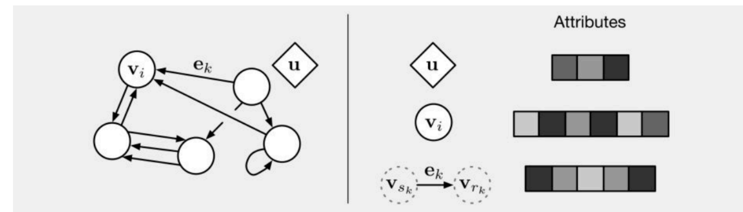
- SOFIE can parse some complex models: CMS Deep Double model (DDB.onnx)
  - 3 inputs with 1d Conv + GRU

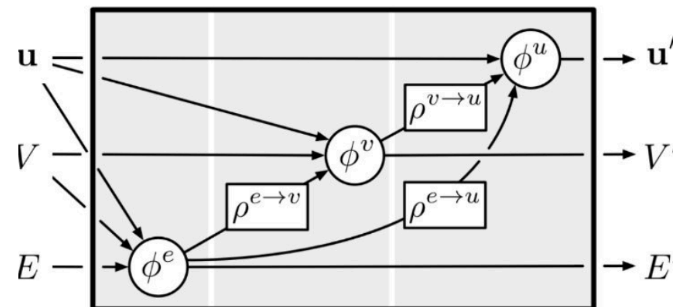




GNN Support

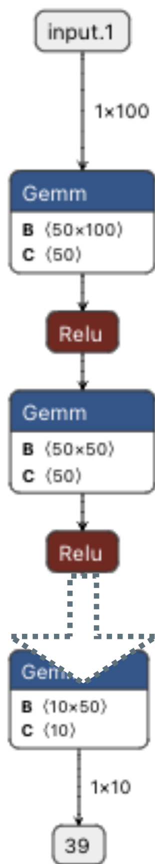
- Follow **Graph Nets** architecture
  - A model is described by
    - number of nodes and edges
    - sender/receiver list of edges
    - number of features (for node, edge and global)
  - Updating functions on node, edge and global features
    - MLP (Multi-Layer Perceptron)
      - including activation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





Benchmark: Dense Model

10 Dense layers



Time per event for different batch size, cache flushed

Lightweight Trained Neural Network (lwtnn)
ONNX Runtime
SOFIE (netlib blas)
SOFIE (OpenBLAS)

Time per event/μs

0
20
40
60
80

0
5
10
15
20
25
30

Batchsize

Benchmark with RDF

- Test on a Deep Neural Network (from TMVA_Higgs_Classification.C tutorial) 5 fully connected layers of 200 units
- Run on dataset of 5M events:
  - Single Thread, but can run also on Multi-Threads