NuGraph Inference on Triton Server

Rishi Singhal

NuGraph All Hands Meeting

22/07/2024

Presentation Overview

- Goals of the project
- Triton Server Setup & Implementation
- Changes(Hacks) done in NuGraph
- Python Client Implementation
- Standalone C++ Client Implementation
- C++ with Larsoft Client Implementation
- C++ with Larsoft and NuSonic Triton Client Implementation (In Progress)
- Further Work
- Repositories links
- Acknowledgement

Goals of the project

- The first goal is to set up an on the fly inference for the NuGraph network (NuGraph2 and NuGraph3) by deploying the same using Nvidia's triton server.
- Secondly, we want the client side of the inference setup to be implemented in C++ using the larsoft products.
- Lastly, we want the triton setup to be adapted to the NuSonic framework to simplify user interface without worrying about triton specific details and make code maintenance easier.

Triton Server Setup & Implementation

- We deploy the NuGraph2 and NuGraph3 models on the already running Triton Server at the EAF-server here at Fermilab.

- We make use of Triton's Python Backend framework to write the server side scripts(in python) for doing inference. This is done because we want to use a normal checkpoint saved model without it being jittable.

- Before setting up the server, we also install a conda environment with python version 3.10 which needs to strictly match the python stub version of triton 24, i.e., 3.10. In the conda environment we install all the necessary libraries for running inference and then zip & store it. **(Note: Please ensure that numpy version is 1.26.4 in the conda-environment for compatibility purpose)**

Contd.. (NuGraph Model folder structure & Other Details)

The triton server expects a specific folder structure for deployment which is as follows:

```
NuGraph2/
   1/
     model.py
     model.ckpt
config.pbtxt
numl.tar.gz
```

Here,

model.py - receives inputs from client to run inference and sends it back to the client.

config.pbtxt - stores the inputs and outputs configuration(shape, name) and path to conda environment

numl.tar.gz - zipped conda environment with all the dependencies installed

Contd.. (Inference Procedure at the server end)

1. After running the server, we initialize the NuGraph2_model which loads the model and also initializes the HitGraphProducer class for pre-processing.

```
# Instantiate the PyTorch model
self.NuGraph2_model = NuGraph2_model()
```

2. Now, the server receives the inputs from the client and pass it to the forward function for pre-processing and inference

```
output1, output2, output3, output4, output5, output6 = \
                            self.NuGraph2_model(hit_table_hit_id.as_numpy(), hit_table_local_plane.as_numpy(), \
                                        hit_table_local_time.as_numpy(), \
        hit_table_local_wire.as_numpy(), hit_table_integral.as_numpy(), hit_table_rms.as_numpy(), \
        spacepoint_table_spacepoint_id.as_numpy(), spacepoint_table_hit_id_u.as_numpy(), spacepoint_table_hit_id_v.as_numpy(), \
        spacepoint_table_hit_id_y.as_numpy())
```

Contd.. (Inference Procedure at the server end)

3. Inside the forward() function we then create the pre-processed graph and its HeteroData object and then passes it for inference via the loaded NuGraph model

```
gnn_hetero_data = self.hitgraph.create_graph(hit_table_hit_id, hit_table_local_plane, hit_table_local_time, \
                                             hit_table_local_wire, hit_table_integral, hit_table_rms, \
                                             spacepoint_table_spacepoint_id, spacepoint_table_hit_id_u, spacepoint_table_hit_id_v, \
                                             spacepoint_table_hit_id_y)

transform = Compose((ng.util.PositionFeatures(self.planes),
                     ng.util.FeatureNorm(self.planes, self.norm),
                     ng.util.HierarchicalEdges(self.planes),
                     ng.util.EventLabels()))
hetero_dataset = HeteroDataset(gnn_hetero_data, transform=transform)
data = hetero_dataset.get()
self.model.step(data.to(self.device))
x = self.model.data
return  x['u']['x_semantic'].cpu().detach().numpy(), \
        x['v']['x_semantic'].cpu().detach().numpy(), x['y']['x_semantic'].cpu().detach().numpy(), \
        x['u']['x_filter'].cpu().detach().numpy(), x['v']['x_filter'].cpu().detach().numpy(), \
        x['y']['x_filter'].cpu().detach().numpy()
```

Contd.. (Inference Procedure at the server end)

4. Once we get the outputs, we convert them to pb_utils.Tensor() and store the result as pb_utils.InferenceResponse object that can be sent back to the client.

```
out_tensor_1 = pb_utils.Tensor("x_semantic_u", output1.astype(x_semantic_u_dtype))
out_tensor_2 = pb_utils.Tensor("x_semantic_v", output2.astype(x_semantic_v_dtype))
out_tensor_3 = pb_utils.Tensor("x_semantic_y", output3.astype(x_semantic_y_dtype))
out_tensor_4 = pb_utils.Tensor("x_filter_u", output4.astype(x_filter_u_dtype))
out_tensor_5 = pb_utils.Tensor("x_filter_v", output5.astype(x_filter_v_dtype))
out_tensor_6 = pb_utils.Tensor("x_filter_y", output6.astype(x_filter_y_dtype))

inference_response = pb_utils.InferenceResponse(
    output_tensors=[out_tensor_1, out_tensor_2, out_tensor_3, out_tensor_4, out_tensor_5, out_tensor_6]
)
responses.append(inference_response)
```

Changes(Hacks) done in Nugraph

1. Currently in Nugraph repo, the **HitGraphProducer class** requires ***"file:pynuml.io.File"*** as one of the arguments to the constructor for pre-processing. But since we don't have the access to the h5 file at the server end, so we create our **own HitGraphProducer class** without the need of h5 file as the argument for pre-processing.

2. In the **EventLabels_class** of the nugraph repo, we first need to check if **data["evt"] has attribute 'y' or not.** This is because during inference we won't have access to the ground truth labels of the event. Hence as a hack, we have added an if-condition which checks the same.

3. In the current version of **Nugraph3.py** and **Nugraph2.py** we are calculating the loss corresponding to the event even during inference. But since, we don't have access to the y-labels during inference we **comment these lines**. Also, we store the output of the model in the form of a class attribute **data** which can be accessed later.

Python Client Implementation

1. For the python client implementation, we first read the h5 file for an event and extract the **particle table** and the **spacepoint table** information & store it in form of a dictionary.

```
with h5py.File("NeutrinoML_rishi.h5", 'r') as test_data:
    test_inputs = {
        "hit_table_hit_id": np.array(test_data['hit_table']['hit_id']).flatten().reshape(-1).astype(np.int32),
        "hit_table_local_plane": np.array(test_data['hit_table']['local_plane']).flatten().reshape(-1).astype(np.int32),
        "hit_table_local_time": np.array(test_data['hit_table']['local_time']).flatten().reshape(-1).astype(np.float32),
        "hit_table_local_wire": np.array(test_data['hit_table']['local_wire']).flatten().reshape(-1).astype(np.int32),
        "hit_table_integral": np.array(test_data['hit_table']['integral']).flatten().reshape(-1).astype(np.float32),
        "hit_table_rms": np.array(test_data['hit_table']['rms']).flatten().reshape(-1).astype(np.float32),

        "spacepoint_table_spacepoint_id": np.array(test_data["spacepoint_table"]["spacepoint_id"]).flatten().reshape(-1).astype(np.int32),
        "spacepoint_table_hit_id_u": np.array(test_data["spacepoint_table"]["hit_id"][:,0:1]).flatten().reshape(-1).astype(np.int32),
        "spacepoint_table_hit_id_v": np.array(test_data["spacepoint_table"]["hit_id"][:,1:2]).flatten().reshape(-1).astype(np.int32),
        "spacepoint_table_hit_id_y": np.array(test_data["spacepoint_table"]["hit_id"][:,2:3]).flatten().reshape(-1).astype(np.int32)
    }


    test_inputs_triton = {}
    c = 0
    for k in test_inputs.keys():
        test_inputs_triton[k] = test_inputs[k]
        c += 1
```

2. Next we create a GRPC connection with the triton server hosted on the EAF-server and send the inputs to the NuGraph model by calling client.infer().

```
response = client.infer(model_name, inputs, request_id=str(1), outputs=outputs)
```

Contd….

1. After receiving the inference response from the server, we fetch the outputs by their names and then display the same.

```
result = response.get_response()
output_1_data = response.as_numpy("x_semantic_u")
output_2_data = response.as_numpy("x_semantic_v")
output_3_data = response.as_numpy("x_semantic_y")
output_4_data = response.as_numpy("x_filter_u")
output_5_data = response.as_numpy("x_filter_v")
output_6_data = response.as_numpy("x_filter_y")

print("Triton output: ")
print("x_semantic_u: ", output_1_data)
print("x_semantic_v: ", output_2_data)
print("x_semantic_y: ", output_3_data)
print("x_filter_u: ", output_4_data)
print("x_filter_v: ", output_5_data)
print("x_filter_y: ", output_6_data)
```

Standalone C++ Client Implementation

For the standalone C++ client implementation, we read the ascii text file of the event under consideration and follow the same procedure of formatting the input in the required triton format, sending for inference and, fetching and displaying the output from the server.

1. Setting the inputs and sending it for inference

```
// Initialize the inputs with the data.
tc::InferInput* hit_table_hit_id;

FAIL_IF_ERR(
    tc::InferInput::Create(&hit_table_hit_id, "hit_table_hit_id", hit_table_shape, "INT32"),
    "unable to get hit_table_hit_id");
std::shared_ptr<tc::InferInput> hit_table_hit_id_ptr;
hit_table_hit_id_ptr.reset(hit_table_hit_id);

FAIL_IF_ERR(
    hit_table_hit_id_ptr->AppendRaw(
        reinterpret_cast<uint8_t*>(&hit_table_hit_id_data[0]),
        hit_table_hit_id_data.size() * sizeof(float)),
    "unable to set data for hit_table_hit_id");

std::vector<tc::InferInput*> inputs = {hit_table_hit_id_ptr.get(), hit_table_local_plane_ptr.get(), hit_table_local_time_ptr.get(), \
                                       hit_table_local_wire_ptr.get(), hit_table_integral_ptr.get(), hit_table_rms_ptr.get(), \
                                       spacepoint_table_spacepoint_id_ptr.get(), spacepoint_table_hit_id_u_ptr.get(), \
                                       spacepoint_table_hit_id_v_ptr.get(), spacepoint_table_hit_id_y_ptr.get()};

std::vector<const tc::InferRequestedOutput*> outputs = {
    x_semantic_u_ptr.get(), x_semantic_v_ptr.get(), \
    x_semantic_y_ptr.get(), x_filter_u_ptr.get(), x_filter_v_ptr.get(), \
    x_filter_y_ptr.get()};

tc::InferResult* results;
FAIL_IF_ERR(
    client->Infer(
        &results, options, inputs, outputs, http_headers,
        compression_algorithm),
    "unable to run model");
std::shared_ptr<tc::InferResult> results_ptr;
results_ptr.reset(results);
```


Fermilab

Contd…

2. Receiving back the outputs from the server and displaying it:

```
// Generate the outputs to be requested.
tc::InferRequestedOutput* x_semantic_u;
FAIL_IF_ERR(
    tc::InferRequestedOutput::Create(&x_semantic_u, "x_semantic_u"),
    "unable to get 'x_semantic_u'");
std::shared_ptr<tc::InferRequestedOutput> x_semantic_u_ptr;
x_semantic_u_ptr.reset(x_semantic_u);

// Get pointers to the result returned...
const float* x_semantic_u_data;
size_t x_semantic_u_byte_size;
FAIL_IF_ERR(
    results_ptr->RawData(
        "x_semantic_u", (const uint8_t**)&x_semantic_u_data, &x_semantic_u_byte_size),
    "unable to get result data for 'x_semantic_u'");

std::cout<<"Trition output: "<<std::endl;

std::cout<<"x_semantic_u: "<<std::endl;
printFloatArray(x_semantic_u_data, x_semantic_u_byte_size/sizeof(float));
std::cout<<std::endl;
```

C++ Client with larsoft Implementation

1. For the C++ client with larsoft implementation, we directly read the event data via larsoft without saving as a h5 or ascii file. Thereby removing the redundant step of first saving the event and then running inference, making it an on-the-fly inference process.

2. After reading the event data, we follow the same process of formatting the input in the required triton format, sending for inference and, fetching and displaying the output from the server that we did in the case of standalone C++ code.

3. Writing to the output root file -
   a. https://github.com/rishi2019194/larrecodnn/blob/develop/larrecodnn/NuGraph/NuGraphInferenceTriton_module.cc#L591

C++ Client with larsoft and NuSonic Triton Implementation (In Progress)

- NuSonic Triton framework is a wrapper over Triton that helps to simplify user interface without worrying about triton specific details and make code maintenance easier.

- Some of the use cases of the same are as follows:

  – Easier way to set parameters and create triton client object

```
// ... Create parameter set for Triton inference client
fhicl::ParameterSet TritonPset;
TritonPset.put("serverURL", fTritonURL);
TritonPset.put("verbose", fTritonVerbose);
TritonPset.put("ssl", fTritonSSL);
TritonPset.put("modelName", fTritonModelName);
TritonPset.put("modelVersion", fTritonModelVersion);
TritonPset.put("timeout", fTritonTimeout);
TritonPset.put("allowedTries", fTritonAllowedTries);
TritonPset.put("outputs", "[]");

// ... Create the Triton inference client
triton_client = std::make_unique<lartriton::TritonClient>(TritonPset);

triton_client->setBatchSize(1); // set batch size
```

Contd..

- Some of the use cases of the same are as follows(Contd..):
  - We need not worry about creating **InferInput()** objects separately, but rather using NuSonic we can just call **toServer()** function that creates InferInput() objects for all the input keys for triton inference.

```
auto hit_table_hit_id_ptr = std::make_shared<lartriton::TritonInput<int32_t>>(); //ptr to the input data
hit_table_hit_id_ptr->reserve(1);  //reserving as batch-size = 1

//filling up the container of the ptr with the elements and formatting to server format
auto& hit_table_hit_id = hit_table_hit_id_ptr->emplace_back();
auto& inputs = triton_client->input();
for (auto& input_pair : inputs) {
    const std::string& key = input_pair.first;
    auto& triton_input = input_pair.second;
    if(key == "hit_table_hit_id"){
      for (size_t i = 0; i < hit_table_hit_id_data.size(); ++i) {
          hit_table_hit_id.push_back(hit_table_hit_id_data[i]);
      }
      triton_input.toServer(hit_table_hit_id_ptr);
    }
}
```

Contd..

- Some of the use cases of the same are as follows(Contd..):
  - Instead of calling **client.infer()**, we directly call **client->dispatch()** which will do the triton inference process within itself. Furthermore, we can creating & access the outputs in much easier fashion without creating **their InferRequestedOutput objects & pointers** in the main function. All this happens internally inside th NuSonic framework for better code maintenance and readability.

```
// ~~~~ Send inference request
std::cout<<"sending inference request"<<std::endl;
triton_client->dispatch();
std::cout<<"sent the request"<<std::endl;

// ~~~~ Retrieve inference results
const auto& triton_output0 = triton_client->output().at("x_semantic_u");
const auto& prob0 = triton_output0.fromServer<float>();
auto ncat0 = triton_output0.sizeDims();
```

Future Work

1. We would like to have the model deployed in the Apptainer at the gpvm machine.

2. Next, we would also want to run scalability tests of the triton-inference setup. Some of the tests could be -

    a. Performance analysis - GPU v/s CPU for inference
    b. Performance analysis - Inference on the EAF server v/s Inference on the Apptainer
    c. Performance analysis when multiple clients are sending request at once

3. Make the code adaptable to batch size > 1 by using the InferMulti() function at the C++ client end. And then do a performance analysis for varying batch sizes during inference.

Repositories links

1. https://github.com/rishi2019194/triton-python-backend-nugraph3 - Contains the necessary code and installation setup and commands for successfully implementing and running triton server and client (in python, standalone c++) and description of the hacks done in Nugraph repo.

2. https://github.com/rishi2019194/nugraph - Contains the code of the Nugraph repo with the hacks/changes to make triton-inference work successfully.

3. https://github.com/rishi2019194/larrecodnn - Contains the code for C++ client with larsoft and C++ client with larsoft & Nusonic Triton setup.

Acknowledgement

I would like to extend my gratitude to everyone who has supported and assisted me throughout this internship project:

- Mike Wang, Kevin Pedro: For their invaluable assistance with C++ and the NuSonic setup.
- Burt: For his guidance in deploying NuGraph on the EAF.
- V: For consistently clarifying my doubts about NuGraph in every meeting.
- Giusseppe Cerati: For his continuous support throughout the internship.

I also want to thank the entire NuGraph team for their valuable inputs.

Thank you, Any Questions??