Introduction to SONIC + Triton (ML) Inference as a Service

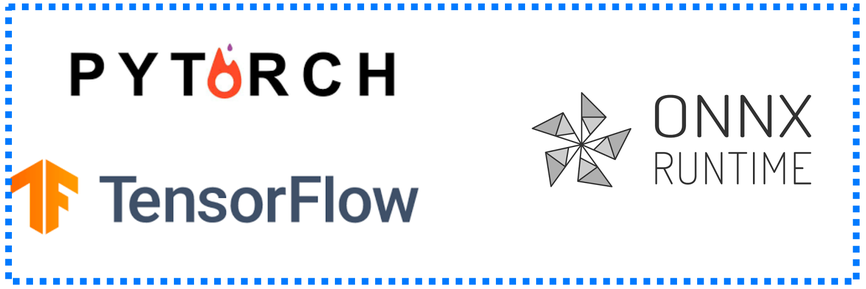
Yongbin Feng, Kevin Pedro, Nhan Tran (Fermilab)

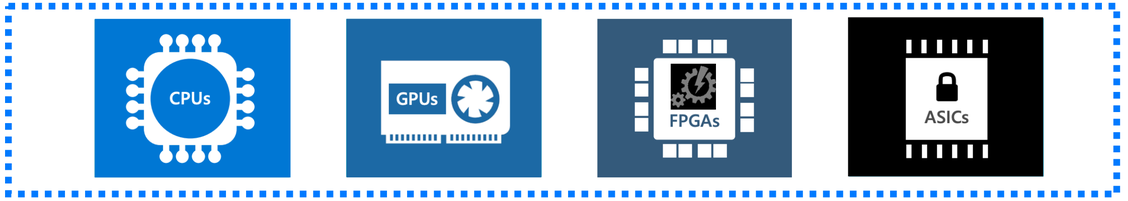
Computational HEP Traineeship Summer School

May 22nd, 2024

Introduction

- After having trained a ML model, I need to put it into (large-scale) **production**. What should I do?
  - Does your software stack (e.g. CMSSW) supports these different **ML framework and operations** - TensorFlow, PyTorch, ONNX, XGBoost/TMVA
  - Does your software stack (e.g. CMSSW) supports **different hardware** - CPUs/GPUs/TPUs/IPUs/DPUs…?
  - Does your computing cluster have these hardware? What if you have some **remote** computing hardware available but not accessible to the production cluster?
  - My algorithms can be accelerated on these hardware, but the fraction of these tasks are very small, not worth the effort. But I have **lots of jobs to process**
- I don't like ML. I have algorithms written in CUDA/ROCm. I want to run these on NVIDIA/AMD GPUs

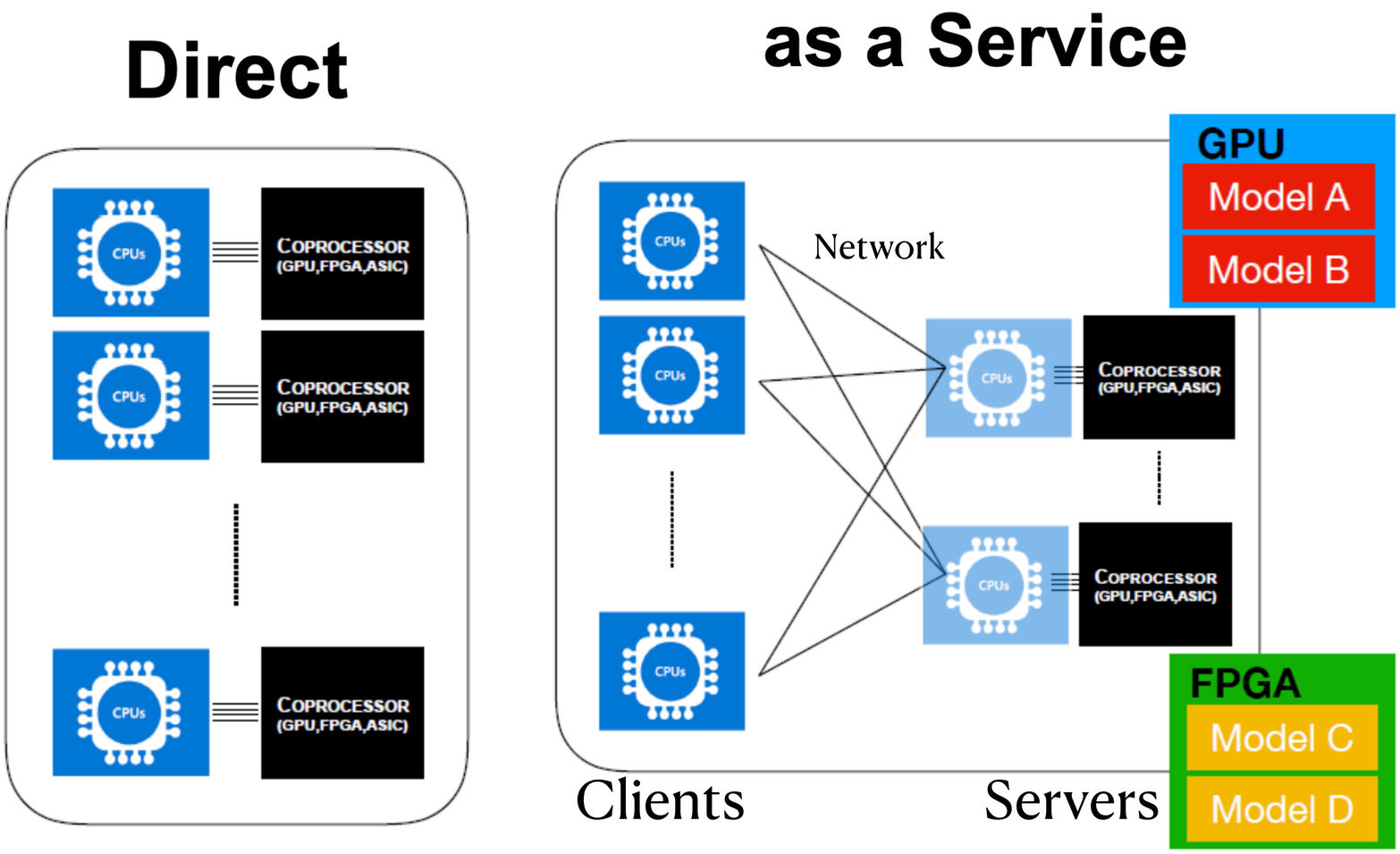




ML Inference Infrastructure

Two ML Inference Infrastructures:

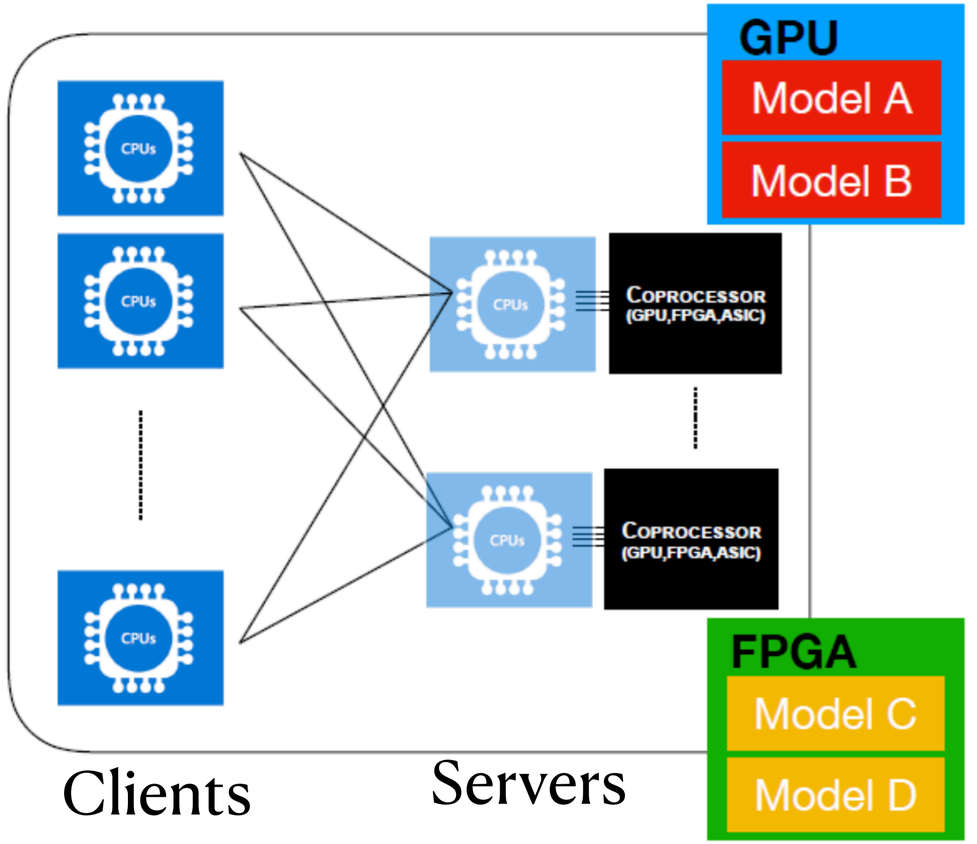
- **Directly connect CPUs and coprocessors**
  - Inference running on the coprocessors directly connected to the CPU
  - Simple connection; no network load
- **Inference as a service (aaS)**
  - Clients communicate with the server, prepare the model inputs to the server and receive model outputs from the server
  - Server directs the coprocessor for model inference



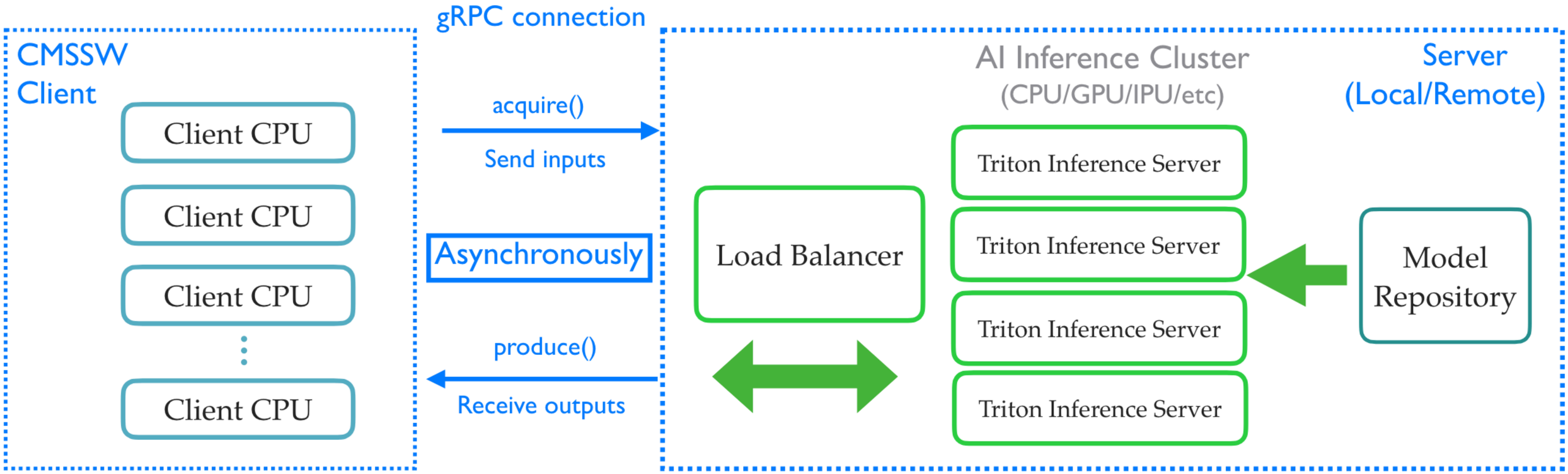
Benefits of Inference aaS

- **Factorize the ML framework out of your main software stack**
  - Only need to handle input and output conversions on the client side (i.e., in CMSSW). Different frameworks supported on the server side.
- **Simple support for different coprocessors:**
  - No need to rewrite algorithms in processor-specific languages
- **More flexibility, better efficiency**
  - One coprocessor can serve many CPU clients
  - ML models can be deployed on different coprocessors simultaneously; choose the best coprocessor for each specific job
- **Dynamic Batching:**
  - Server can batch inference requests from different clients
- **Allow access remote GPUs**

as a Service



SONIC in CMSSW As an Example



- SONIC (Service for Optimized Network Inference on Coprocessors) available in CMSSW
- The Client in CMSSW **sends the inference request** with inputs for the model, and **receives outputs from** the server
- NVIDIA Triton server runs the inference

Modesl for the server

- One example can be found here
- Model directory structured as:
  - Model config file
  - Version/model_file

```
deeptau_nosplit/
    config.pbtxt
    1/
        model.graphdef
```

- Model config file specifies:
  - The model names and types
  - Inputs and outputs names and dimensions
  - Some parameters for version control and optimizations (max batch size, latency, dynamic batching, precision reduction, etc)

- Model files: TF, ONNX, PyTorch, Scikit-learn, etc
- More information about the model configs and these parameters can be found here and here

```
name: "deepmet"
platform: "tensorflow_graphdef"
max_batch_size: 100
input [
  {
    name: "input"
    data_type: TYPE_FP32
    dims: [ 4500, 8 ]
  },
  {
    name: "input_cat0"
    data_type: TYPE_FP32
    dims: [ 4500, 1 ]
  },
  {
    name: "input_cat1"
    data_type: TYPE_FP32
    dims: [ 4500, 1 ]
  },
  {
    name: "input_cat2"
    data_type: TYPE_FP32
    dims: [ 4500, 1 ]
  }
]
output [
  {
    name: "output/BiasAdd"
    data_type: TYPE_FP32
    dims: [ 2 ]
  }
]

version_policy: { all { }}
dynamic_batching {
    preferred_batch_size: [ 8, 16 ]
}
```

Client

- NVIDIA Triton provides PerfClient to mimic (multiple) clients communicating with server, in order to **benchmark (ML) model inference performance:**
  - **I**nference batch sizes, request concurrency, number of model instances on the servers, etc
  - Optimizing model configs: TensorRT optimization, Just-In-Time compilation/XLA, quantization/reduced precision, etc

- Python and cpp based clients sending/receiving gRPC calls

- PyTriton building and testing both server and clients in one shot

- In practice, will probably need to integrate these into your (experiment) software stack: CMSSW, Coffea, (Proto-)DUNE, either via a build, or pip install

SONIC Producers in CMSSW

- One SONIC producer example can be found here. More examples here
- The producer inherits from the TritonEDProducer:
- Core Parts:
  - Acquire function sending inputs

```
30  DeepMETSonicProducer::DeepMETSonicProducer(const edm::ParameterSet& cfg)
31      : TritonEDProducer<>(cfg),
32        pf_token_(consumes<std::vector<pat::PackedCandidate>>(cfg.getParameter<edm::InputTag>("pf_src"))),
33        norm_(cfg.getParameter<double>("norm_factor")),
34        ignore_leptons_(cfg.getParameter<bool>("ignore_leptons")),
35        max_n_pf_(cfg.getParameter<unsigned int>("max_n_pf")),
36        scale_(1.0 / norm_) {
37    produces<pat::METCollection>();
38  }
```

```
void DeepMETSonicProducer::acquire(edm::Event const& iEvent, edm::EventSetup const& iSetup, Input& iInput) {
  // one event per batch
  client_->setBatchSize(1);
  px_leptons_ = 0.;
  py_leptons_ = 0.;

  auto const& pfs = iEvent.get(pf_token_);

  auto& input = iInput.at("input");
  auto pfdata = input.allocate<float>();
  auto& vpfdata = (*pfdata)[0];
```

```
100     // fill the remaining with zeros
101     // resize the vector to 4500 for zero-padding
102     vpfdata.resize(8 * max_n_pf_);
103     vpfchg.resize(max_n_pf_);
104     vpfpdgId.resize(max_n_pf_);
105     vpffromPV.resize(max_n_pf_);
106
107     input.toServer(pfdata);
108     input_cat0.toServer(pfchg);
109     input_cat1.toServer(pfpdgId);
110     input_cat2.toServer(pffromPV);
111   }
```

SONIC Producers in CMSSW

- One SONIC producer example can be found here. More examples here
- The producer inherits from the TritonEDProducer:
- Core Parts:
  - Acquire function sending inputs
  - Produce function receiving outputs

```
113  void DeepMETSonicProducer::produce(edm::Event& iEvent, edm::EventSetup const& iSetup, Output const& iOutput) {
114    const auto& output1 = iOutput.begin()->second;
115    const auto& outputs = output1.fromServer<float>();
116
117    // outputs are px and py
118    float px = outputs[0][0] * norm_;
119    float py = outputs[0][1] * norm_;
120
```

Running Everything

- **Besides the regular configurations and loadings, etc, control the inference with TritonService**
  - Point to the “remote” servers: server name, address, and GRPC port number that the server is running on (by default is 8001)

- **Most of the time servers are running in containers, through docker/podman/ apptainer(singularity)**

```
process.load("HeterogeneousCore.SonicTriton.TritonService_cff")
process.TritonService.verbose = False
#process.TritonService.fallback.useDocker = True
process.TritonService.fallback.verbose = False
# uncomment this part if there is one server running at 0.0.0.0 with grpc port 8001
#process.TritonService.servers.append(
#    cms.PSet(
#        name = cms.untracked.string("default"),
#        address = cms.untracked.string("0.0.0.0"),
#        port = cms.untracked.uint32(8021),
#    )
#)
```

apptainer run --nv -B /path/to/triton/repo:/models triton_21.10.sif tritonserver --model-repository=/models

docker run -it --gpus=1 --rm -p8000:8000 -p8001:8001 -p8002:8002 -v/path/to/triton/models/:/models nvcr.io/nvidia/tritonserver:23.10-py3 tritonserver --model-repository=/models/

Useful Links (Mostly for CMS)

- Document on the NVIDIA Triton inference server:
  https://docs.nvidia.com/deeplearning/triton-inference-server/archives/triton_inference_server_230/user-guide/docs/
- SONIC Core:
  https://github.com/cms-sw/cmssw/tree/master/HeterogeneousCore/SonicCore
- SONIC Triton:
  https://github.com/cms-sw/cmssw/tree/master/HeterogeneousCore/SonicTriton
- cmsTrion script to launch the triton server:
  https://github.com/cms-sw/cmssw/blob/master/HeterogeneousCore/SonicTriton/scripts/cmsTriton
- SONIC + Triton examples:
  https://github.com/cms-sw/cmssw/tree/master/HeterogeneousCore/SonicTriton/test

Back Up

SONIC Framework in CMSSW

- **SonicCore (repo)**
  - Modules (EDProducer, EDFilter, EDAnalyzer) and client based classes
  - Synchronous and Asynchronous modes for clients
- **SonicTriton (repo)**
  - Modules, clients, data types, and services for Triton inference server
  - cmsTriton script to launch and manage the Triton server via Docker or Singularity

- Requirements for running inferences, very similar to the direct inferences:
  - One model directory with the models and configs
  - One SONIC producer to handle the pre/post processings and the IOs
  - One python config file

- Most of the materials in the slides can be found here.