GPU as a Service (GPUaaS) for accelerating DL inference applications in LArSoft

Michael Wang

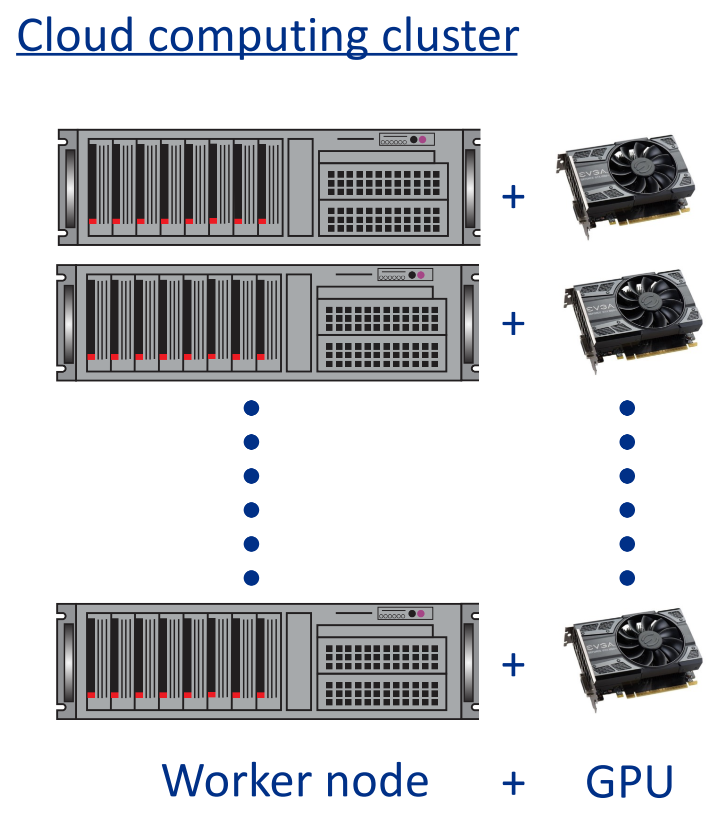
LArSoft Multi-threading and Acceleration Workshop

March 2-3, 2023

Introduction

- Deep learning methods are very well suited to the image-like data in LArTPC-based neutrino experiments
- In recent years, the use of DL applications has become quite common in the LArSoft reconstruction workflows of various experiments
- The DL task typically performed in these workflows is that of *inference*, which while not as demanding computationally as the *training* task, can still dominate a significant portion of the LArsoft workload when performed on less than optimal architectures like oridinary CPUs:
  - e.g. the CNN-based EmTrackMichelId algorithm in a ProtoDUNE-I reconstruction workflow accounted for >60% of the total execution time.
- Since the use of DL applications in LArSoft workflows will only become more widespread in coming years, it is important to find a practical solution to address the increased computational requirements this will entail

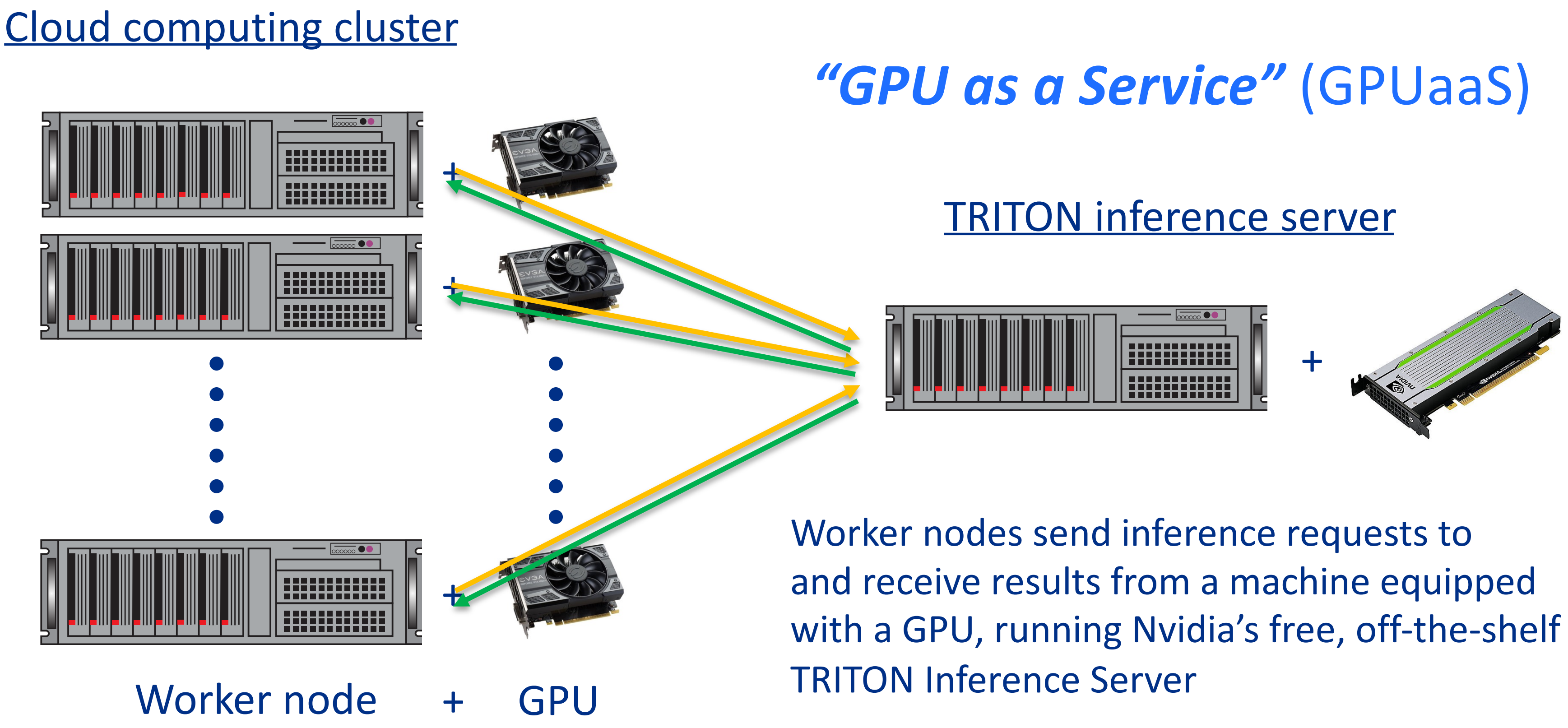
Naïve solution: deploy GPUs on every worker node



Assuming a moderately sized cluster with 100 nodes:

- Even with low to mid-range "gamer-class" GPUs, easily cost $20k-30k
- Increased hardware and software maintenance
- Increased power and cooling requirements
- Inefficient use of GPU resources
- Less flexible and more costly to upgrade or replace

Alternative: Offload inference task to GPU on a remote server

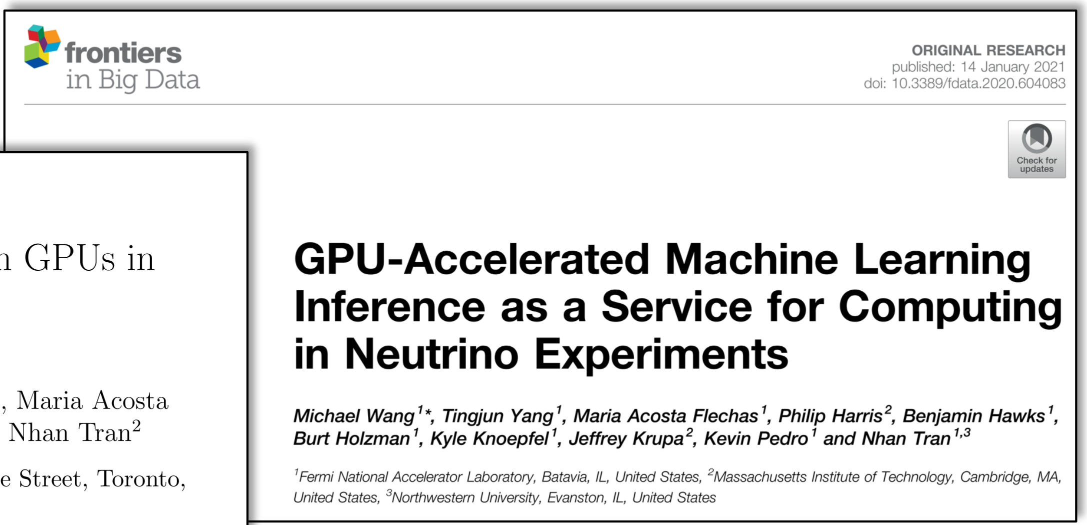


"GPU as a Service" (GPUaaS)

Worker nodes send inference requests to and receive results from a machine equipped with a GPU, running Nvidia's free, off-the-shelf TRITON Inference Server

The SONIC approach

- This "GPUaaS" approach has been taken in the CMS experiment:
  - SONIC: Services for Optimized Network Inference on Coprocessors
- It is also implemented in LArSoft as NuSONIC and there are two papers that describe GPUaaS in LArSoft:



frontiers in Big Data

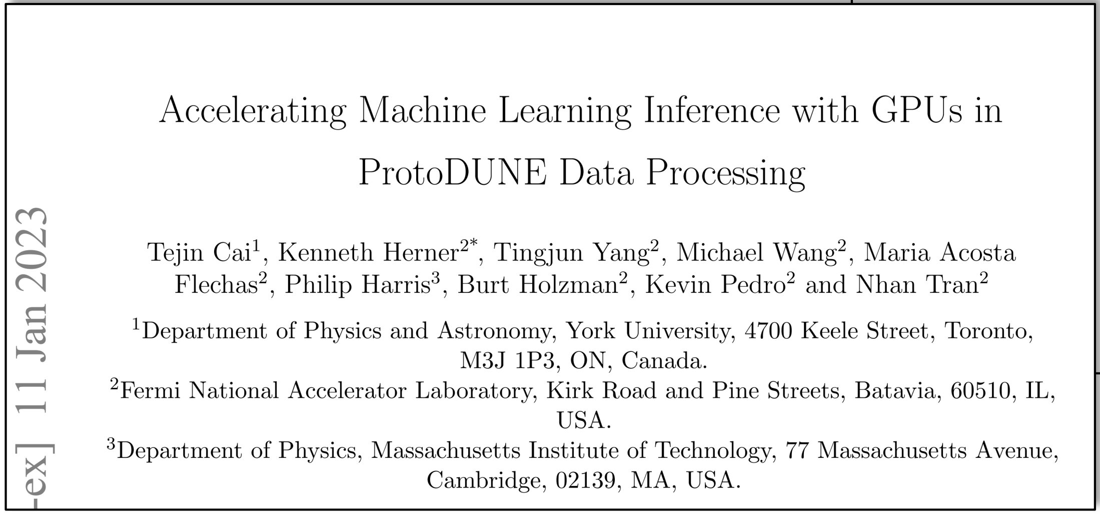
ORIGINAL RESEARCH
published: 14 January 2021
doi: 10.3389/fdata.2020.604083

Check for updates

GPU-Accelerated Machine Learning Inference as a Service for Computing in Neutrino Experiments

Michael Wang[1], Tingjun Yang[1], Maria Acosta Flechas[1], Philip Harris[2], Benjamin Hawks[1], Burt Holzman[1], Kyle Knoepfel[1], Jeffrey Krupa[2], Kevin Pedro[1] and Nhan Tran[1,3]*

[1]Fermi National Accelerator Laboratory, Batavia, IL, United States, [2]Massachusetts Institute of Technology, Cambridge, MA, United States, [3]Northwestern University, Evanston, IL, United States



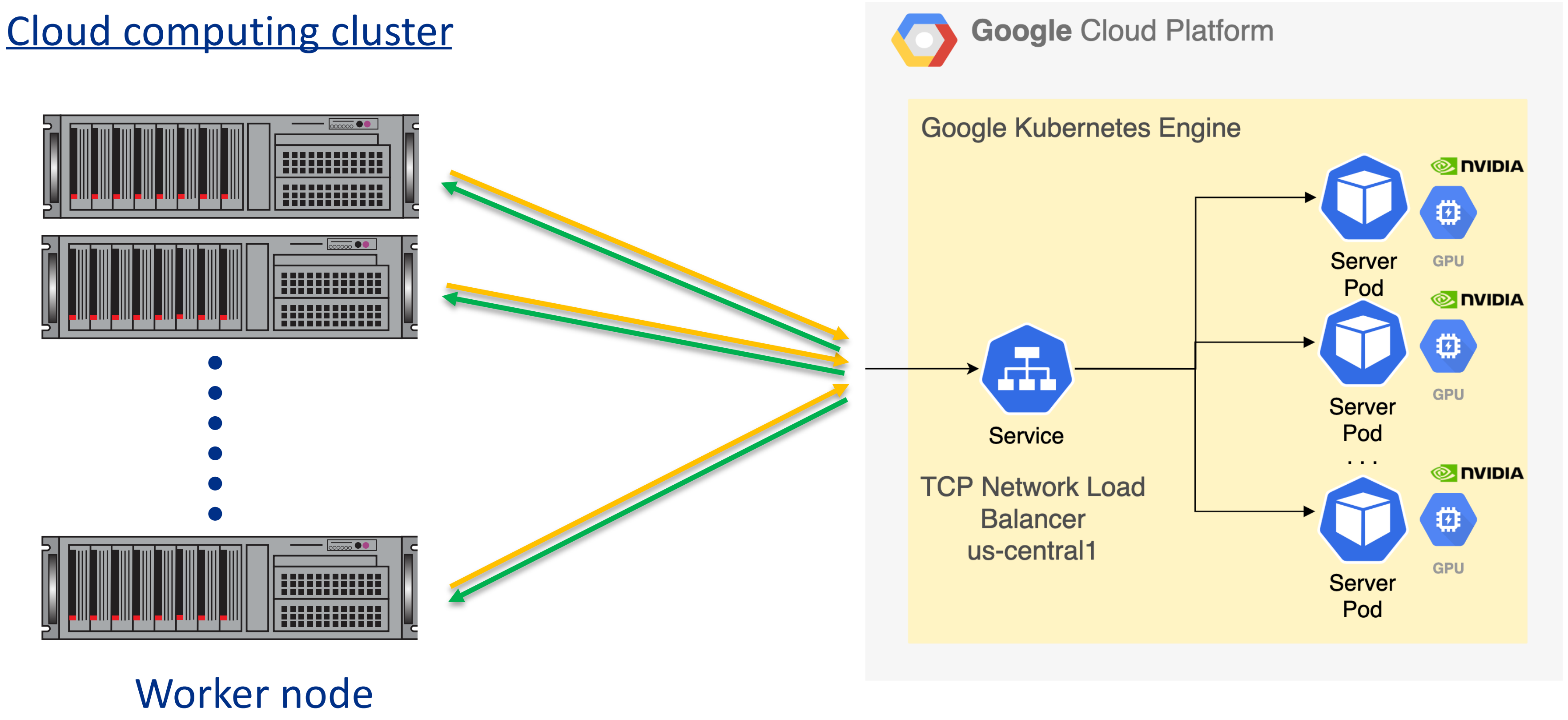
-ex] 11 Jan 2023

Accelerating Machine Learning Inference with GPUs in ProtoDUNE Data Processing

Tejin Cai[1], Kenneth Herner[2*], Tingjun Yang[2], Michael Wang[2], Maria Acosta Flechas[2], Philip Harris[3], Burt Holzman[2], Kevin Pedro[2] and Nhan Tran[2]

[1]Department of Physics and Astronomy, York University, 4700 Keele Street, Toronto, M3J 1P3, ON, Canada.
[2]Fermi National Accelerator Laboratory, Kirk Road and Pine Streets, Batavia, 60510, IL, USA.
[3]Department of Physics, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, 02139, MA, USA.

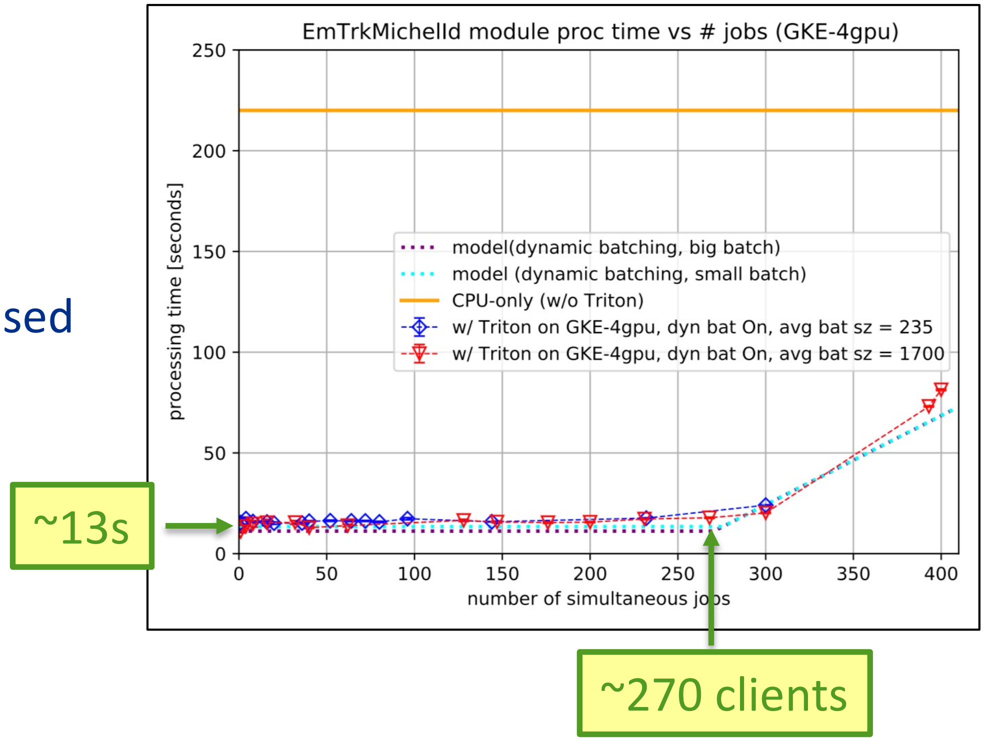
Google Cloud Platform based inference server



Fermilab

Initial study: using simulated ProtoDUNE events

Using a 4-GPU
GCP Kubernetes-based
Triton server



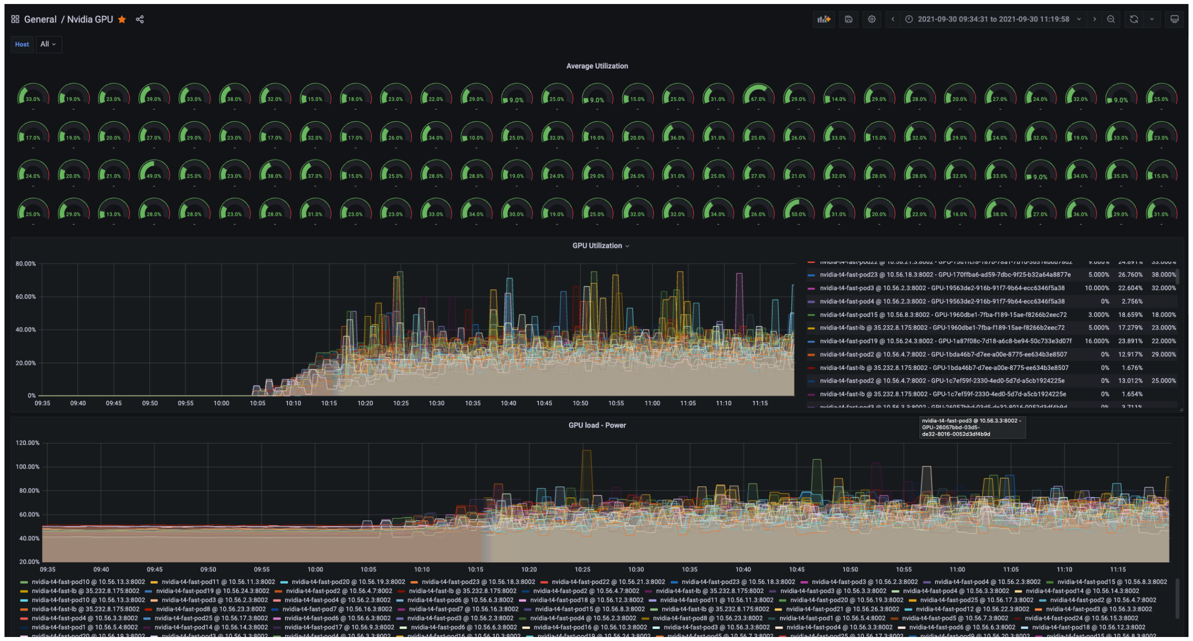
220 s CPU time for ML module (EmTrkMichelId) is reduced to ~13 s

~17 x speedup for the ML module and ~2.7 x overall speedup

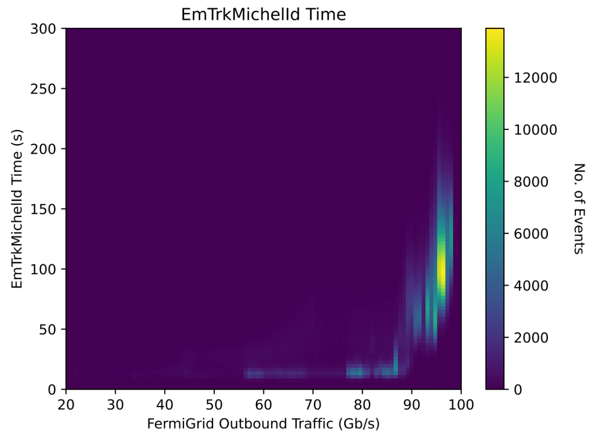
Saturation point at ~270 simultaneous clients – GPU:CPU ratio of ~1:68

Updated study: Using 7.2M real ProtoDUNE events

- 6.4M events using Triton
- 0.8M events using CPU only for comparison
- 100 GPU GCP Kubernetes based Triton inference server (vs 4GPU in initial study)
- Up to ~6000 concurrent jobs



Updated study: network saturation

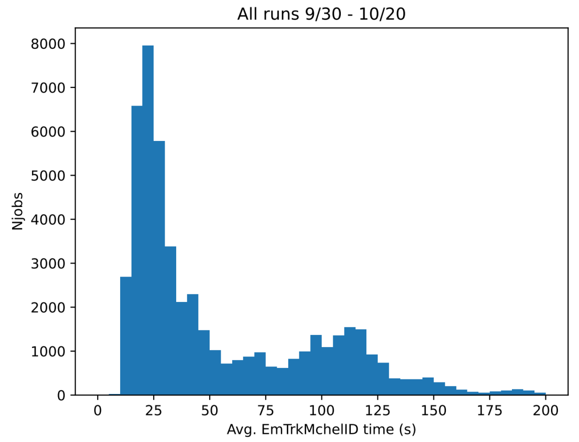


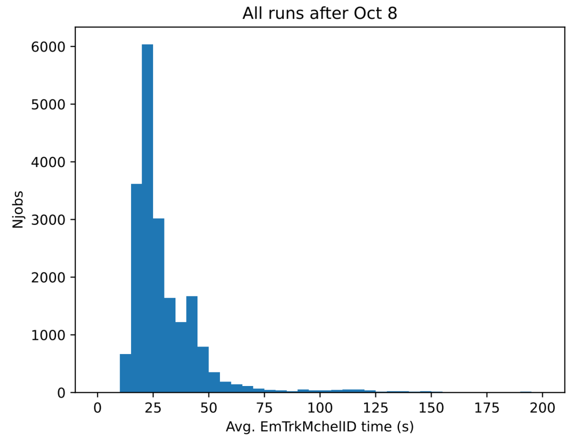
ML module execution time vs network traffic indicates network saturation.

ML module execution time:

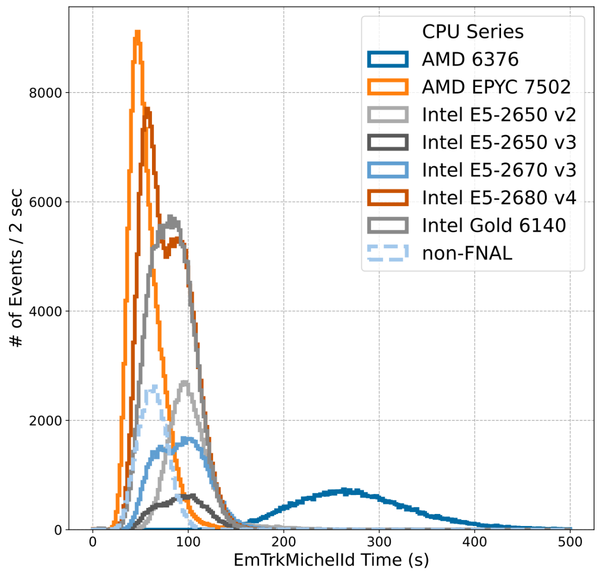
Dual peak structure with long tail due to network saturation

Taking into account 100 Gb/s limit of FNAL switch for outbound traffic, and amount of data transmitted per event, imposed a limit to the number of concurrent Jobs = 600.





Updated study: GPU vs CPU speedup



Distribution of ML module execution times for CPU-only runs.

<u>For FNAL runs:</u>

- ML module take ~79s to execute on CPU
  - Compare with ~25s to execute on GPU (see slide 9)
  - Speedup of ~3x
- In the initial study using simulated events, speedup was more like ~17x (see slide 7)
- Discrepancy: newer version of tensorflow appears to have been optimized to take advantage of CPU extensions

NuSONIC

- For the proof-of-principle work described on slide 7, we used the Nvidia Triton C++ client library API directly in the relevant LArSoft code for EmTrackMichelId
- Since then, we introduced the NuSONIC inference client library, which is a layer sitting on top of the Triton C++ API:
  - Based on CMSSW's SONIC (Services for Optimized Network Inference on Coprocessors), essentially a pared down version retaining only components most relevant for LArSoft
  - Higher level interface that abstracts away Triton specific details, making it easier to use
  - Insulates the LArSoft user from future updates to the Triton API, simplifying maintenance
  - Found in LArSoft since v09_27_00, specifically since larrecodnn v09_08_00
- In the following slides, we will give a brief overview of how to use this in LArSoft code, just to give you some idea. For more details, please refer to:

  https://cdcvs.fnal.gov/redmine/projects/larsoft/wiki/GPU_as_a_Service

Prerequisites for using Nu SONIC

***Server side*:**

- Must have an Nvidia Triton inference server running:
  - Either locally on computer, in a docker container for example
  - Or remotely on a dedicated server
- Machine learning model for your algorithm must be available in the model repository of the Triton inference server
  - Must be configured in a specific way which is described in some detail on the wiki page

Prerequisites for using NuSonic

Client side software:

- Modify the CMakeLists.txt file to include line for NuSonic Triton library:

```
art_make(
          TOOL_LIBRARIES
                 .
                 .
          larrecodnn_ImagePatternAlgs_NuSonic_Triton
                 .
                 .
        )
```

- Include the following header in your code:

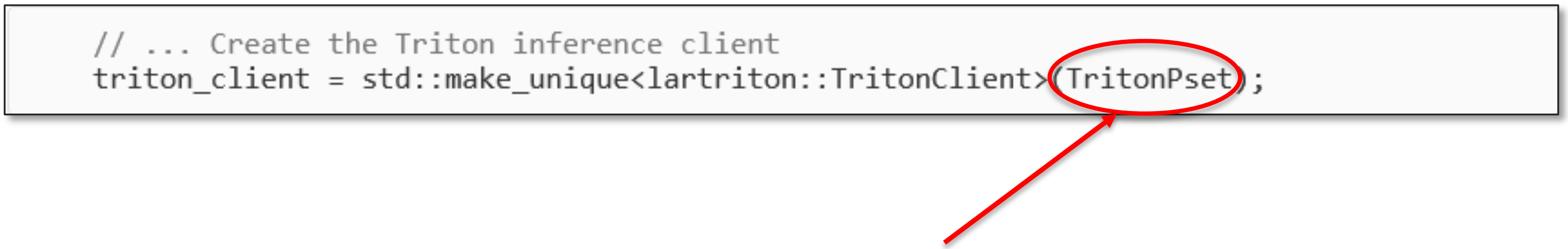
```
#include "larrecodnn/ImagePatternAlgs/NuSonic/Triton/TritonClient.h"
```

Four basic steps in using NuSonic client in code

- Once the requirements in the previous slides are satisfied, implementing an inference client in your code with NuSonic involves only the following 4 simple steps:
  1. Creating the client
  2. Describe or prepare the data for the corresponding model input
  3. Send the inference request to the server
  4. Retrieve the inference results from the server

Step 1: client constructor

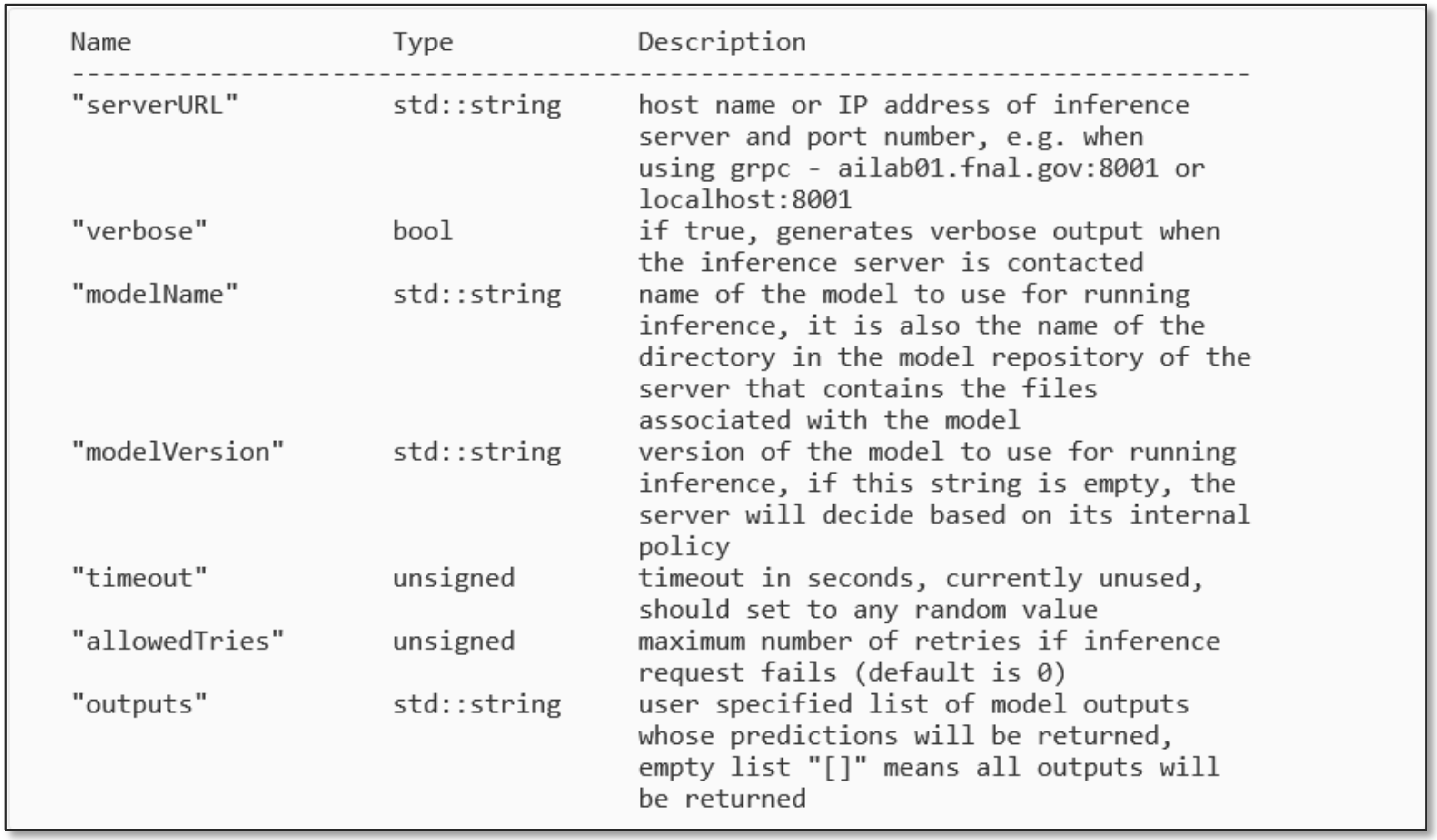
- Step 1: create a Triton client by constructing a lartriton::TritonClient object:



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

- The argument for the constructor is a fhicl::Parameter set with the user-specified parameters described on the next slide.

Step 1: client constructor (continued)

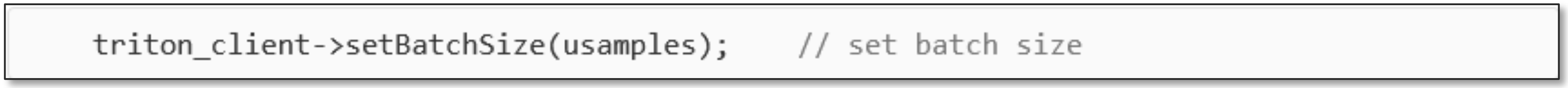


```
Name                Type           Description
----------------------------------------------------------------------------
"serverURL"         std::string    host name or IP address of inference
                                   server and port number, e.g. when
                                   using grpc - ailab01.fnal.gov:8001 or
                                   localhost:8001
"verbose"           bool           if true, generates verbose output when
                                   the inference server is contacted
"modelName"         std::string    name of the model to use for running
                                   inference, it is also the name of the
                                   directory in the model repository of the
                                   server that contains the files
                                   associated with the model
"modelVersion"      std::string    version of the model to use for running
                                   inference, if this string is empty, the
                                   server will decide based on its internal
                                   policy
"timeout"           unsigned       timeout in seconds, currently unused,
                                   should set to any random value
"allowedTries"      unsigned       maximum number of retries if inference
                                   request fails (default is 0)
"outputs"           std::string    user specified list of model outputs
                                   whose predictions will be returned,
                                   empty list "[]" means all outputs will
                                   be returned
```

Fermilab

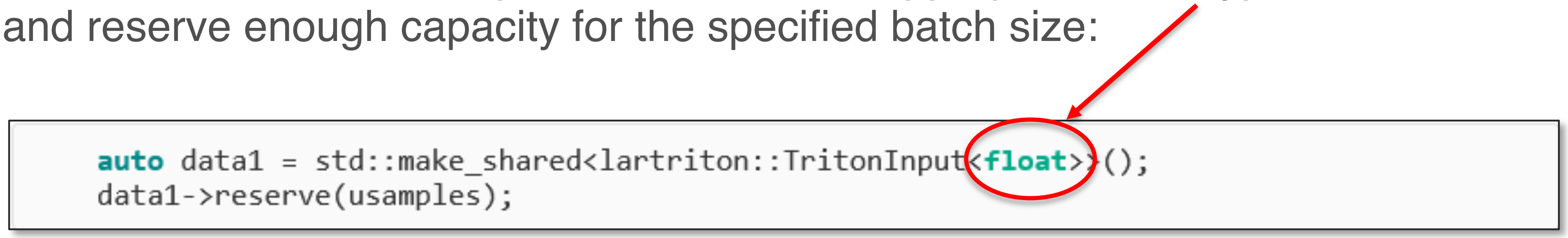
Step 2: Preparing the data

- Set the batch size



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

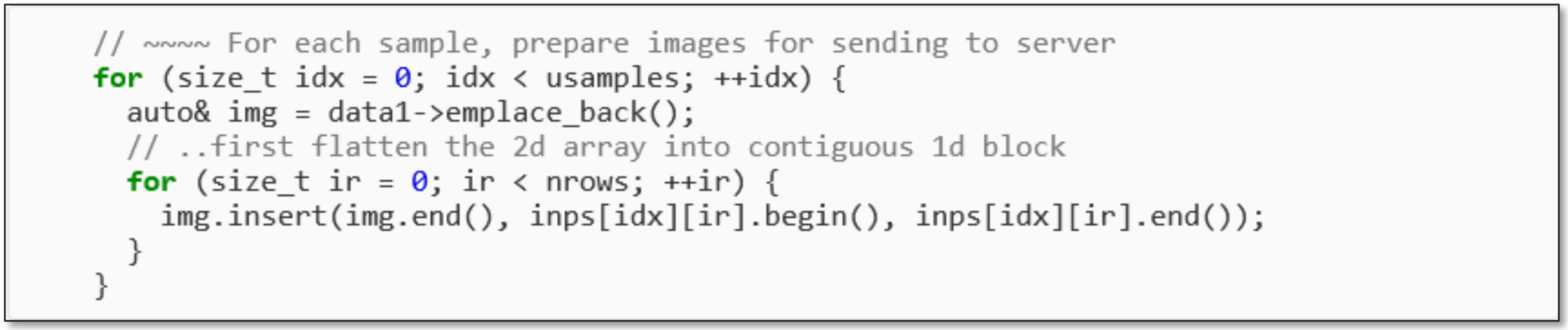
- Create a lartriton::TritonInput container of the appropriate data type for the model and reserve enough capacity for the specified batch size:



```
auto data1 = std::make_shared<lartriton::TritonInput<float>>();
data1->reserve(usamples);
```

Step 2: Preparing the data (continued)

- Fill each element of container with flattened 1D representation of image:



```cpp
// ~~~~ For each sample, prepare images for sending to server
for (size_t idx = 0; idx < usamples; ++idx) {
  auto& img = data1->emplace_back();
  // ..first flatten the 2d array into contiguous 1d block
  for (size_t ir = 0; ir < nrows; ++ir) {
    img.insert(img.end(), inps[idx][ir].begin(), inps[idx][ir].end());
  }
}
```

Step 2: Preparing the data (continued)

- Get the TritonInputData instance associated with the model input to which data will be sent:

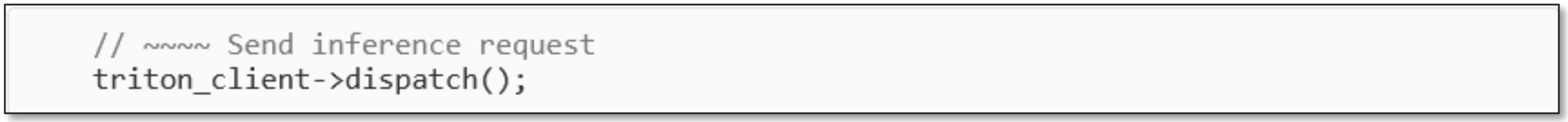
```
auto& triton_input = triton_client->input().at("main_input");
```

- Convert data into native format of the inference server:

```
triton_input.toServer(data1);    // convert to server format
```

Step 3: Send the inference request to the Triton server

- Send the entire batch of images prepared in the previous step to the server with the dispatch method of the Triton client:

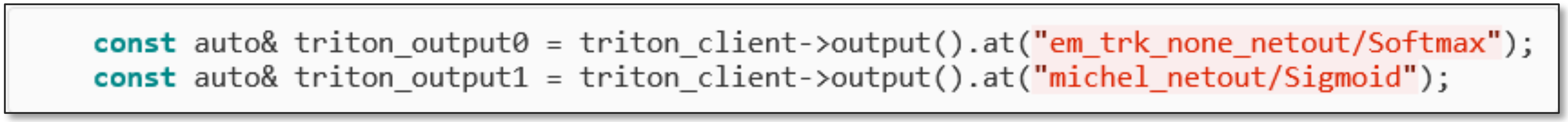


```
// ~~~~ Send inference request
triton_client->dispatch();
```

- And wait for the blocking call above to return …

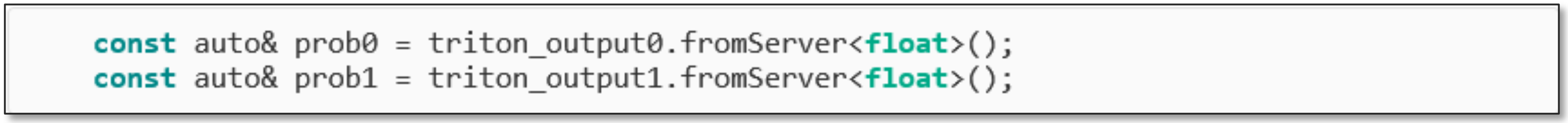
Step 4: Retrieving the results

- Once the dispatch call in the previous step returns successfully, the predictions can be retrieved by name for any output through the TritonOutputData object:



```
    const auto& triton_output0 = triton_client->output().at("em_trk_none_netout/Softmax");
    const auto& triton_output1 = triton_client->output().at("michel_netout/Sigmoid");
```

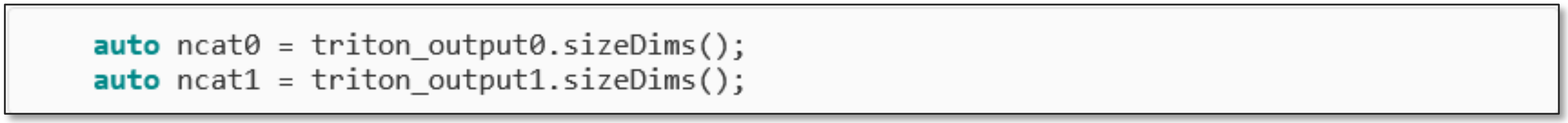
- The results need to be converted from the native format of the server back into the data type associated with each output:



```
    const auto& prob0 = triton_output0.fromServer<float>();
    const auto& prob1 = triton_output1.fromServer<float>();
```

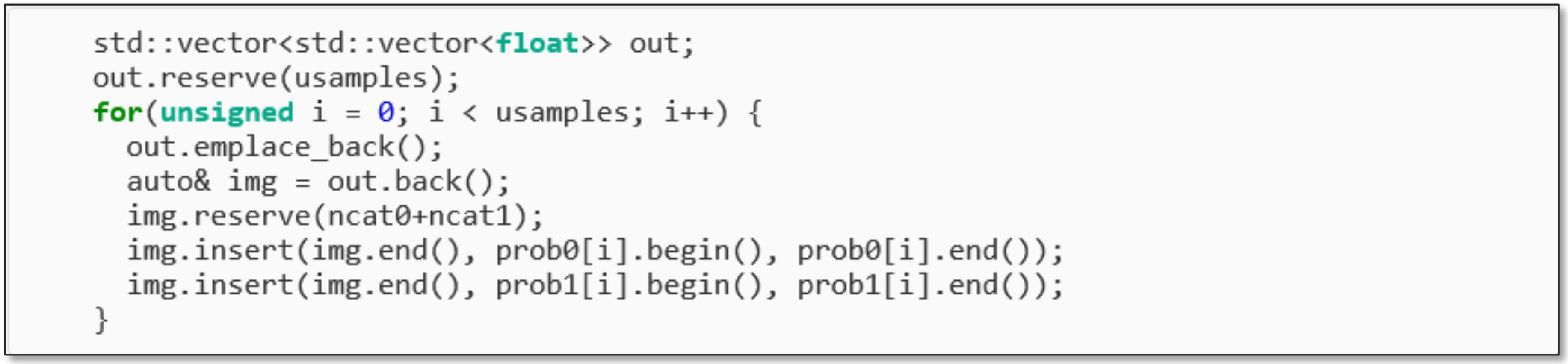
Step 4: Retrieving the results

- Get the number of classifications or categories associated with each output:



```
auto ncat0 = triton_output0.sizeDims();
auto ncat1 = triton_output1.sizeDims();
```

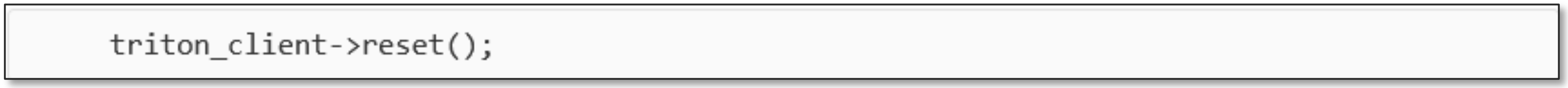
- Get predictions for each classification/category in each output for every image in the batch. In this example, we store these results in a 2D vector:



```
std::vector<std::vector<float>> out;
out.reserve(usamples);
for(unsigned i = 0; i < usamples; i++) {
  out.emplace_back();
  auto& img = out.back();
  img.reserve(ncat0+ncat1);
  img.insert(img.end(), prob0[i].begin(), prob0[i].end());
  img.insert(img.end(), prob1[i].begin(), prob1[i].end());
}
```

Resetting the client

- Before repeating the 4 steps described above for a new batch of images, you will need to reset the Triton client. This will re-initialize all TritonInputData instances by clearing out any values set by previous calls in the data preparation phase:



```
    triton_client->reset();
```

Conclusion

- NuSONIC in LArSoft: GPUaaS capability available for accelerating DL inference tasks by offloading requests to a remote nVidia TRITON based inference server:
  - GPUaaS is not limited to GPUs, the underlying accelerator or co-processor can be based on other types of hardware like FPGAs, TPUs, GraphCore IPUs, etc
- Two detailed sets of studies have been performed and published that characterize the performance of this DL inference acceleration service in LArSoft using ProtoDUNE data
- Provided a brief overview of how to use this service in LArSoft
- To make this really useful for experimenters, we will need to provide TRITON servers either at the lab or in the cloud
  - Ongoing discussion on making such servers available

End

Fermilab