INFERENCE SERVER BETA RELEASE

DU-08994-001_v0.1 | October 2018

User Guide

TABLE OF CONTENTS

Chapter 1. OVERVIEW OF THE INFERENCE SERVER

The NVIDIA Inference Server™ provides a cloud inferencing solution optimized for NVIDIA GPUs. The server provides an inference service via an HTTP endpoint, allowing remote clients to request inferencing for any model being managed by the server. The Inference Server provides the following features:

Multiple model support
: The server can manage any number and mix of models (limited by system disk and memory resources). Supports TensorRT and TensorFlow GraphDef model formats.

Multi-GPU support
: The server can distribute inferencing across all system GPUs.

Multi-tenancy support
: Multiple models (or multiple instances of the same model) can run simultaneously on the same GPU.

Batching support

The Inference Server itself is provided as a pre-built container. External to the server, API schemas, C++ and Python client libraries, and related documentation are provided in source at: GitHub Inference Server.

1.1. Contents Of The Inference Server Container

This image contains the inference server in **`/opt/inference_server`**. The executable is **`/opt/inference_server/bin/inference_server`**.

Chapter 2. PULLING THE INFERENCE SERVER CONTAINER

You can pull (download) an NVIDIA container that is already built, tested, tuned, and ready to run. Each NVIDIA deep learning container includes the code required to build the framework so that you can make changes to the internals. The containers do not contain sample data-sets or sample model definitions unless they are included with the source for the framework.

Currently, you can access NVIDIA GPU accelerated containers in one of two ways depending upon where you doing your training. If you own a DGX-1™ or a DGX Station™ , then you should use the NVIDIA® DGX™ container registry located at https://compute.nvidia.com. You can pull the containers from there and you can also push containers there into your own account on the nvidia-docker repository, **`nvcr.io`**.

If you are accessing the NVIDIA containers from a Cloud Server Provider such as Amazon Web Services (AWS), then you should first create an account at the NVIDIA NGC Cloud Services located at https://ngc.nvidia.com. After you create an account, the commands to use containers are the same for the DGX-1 and the DGX Station. However, currently, you cannot save any containers to the NVIDIA® GPU Cloud™ (NGC) container registry, **`nvcr.io`** if you are using NVIDIA® GPU Cloud™ (NGC). Instead you have to save the containers to your own Docker repository.

The containers are exactly the same, whether you pull them from the NVIDIA DGX container registry or the NGC container registry.

Before you can pull a container you must have Docker and nvidia-docker installed as explained in Preparing to use NVIDIA Containers Getting Started Guide. You must also have access and logged into the NGC container registry as explained in NGC Getting Started Guide.

For step-by-step instructions, see Container User Guide.

Chapter 3. RUNNING THE INFERENCE SERVER CONTAINER

Before running the Inference Server, you must first set up a model store containing the models that the server will make available for inferencing. The Model Store, describes how to create a model store. For this example, assume the model store is created on the host system directory **/path/to/model/store**. The following command will launch the inference server using that model store.

```
$ nvidia-docker run
--rm --shm-size=1g --ulimit memlock=-1 --ulimit
stack=67108864 -p8000:8000 --mount
type=bind,source=/path/to/model/store,target=/tmp/models <container>
/opt/inference_server/bin/inference_server
--model_base_path=/tmp/models
```

Where **<container>** is the name of the docker container that was pulled from the NVIDIA DGX or NGC container registry as described in Pulling The Inference Server Container.

The **nvidia-docker --mount** option maps **/path/to/model/store** on the host into the container at **/tmp/models**, and the **--model_base_path** option to the Inference Server is used to point to **/tmp/models** as the model store.

The Inference Server listens on port 8000 and the above command uses the **-p** flag to map container port 8000 to host port 8000. A different host port can be used by modifying the **-p** flag, for example **-p9000:8000** will cause the Inference Server to be available on host port 9000.

The **--shm-size** and **--ulimit** flags are recommended to improve Inference Server performance. For **--shm-size** the minimum recommended size if 1g but larger sizes may be necessary depending on the number and size of models being served.

After starting, the Inference Server will log initialization information to the console. Initialization is complete and the server is ready to accept requests after the console shows the following:

```
Starting server listening on :8000
```

Chapter 4. VERIFYING THE INFERENCE SERVER

The simplest way to verify that the Inference Server is running correctly is to use the Server Status API to query the server's status. For more information about the Inference Server API, see Inference Server API. From the host system use **`curl`** to request server status. The response is protobuf text showing the status for the server and for each model being served, for example:

```
$ curl localhost:8000/api/status
version: "18.04"
model_status {
  key: "resnet50"
  value {
    config {
      name: "resnet50"
      model_platform: "tensorflow_graphdef"
      max_batch_size: 128
      input {
        name: "input"
        data_type: TYPE_FP32
        format: FORMAT_NHWC
        dims: 224
        dims: 224
        dims: 3
      }
      output {
        name: "output"
        data_type: TYPE_FP32
        dims: 1000
        label_filename: "resnet50_labels.txt"
      }
...
```

Chapter 5. MODEL STORE

The Inference Server accesses models from a locally accessible file path. This path is specified when the server is started using the **`--model_base_path`** option. The model store must be organized as follows:

```
<model_base_path>/
  model_0/
    config.pbtxt
    output0_labels.txt
    1/
      model.plan
    2/
      model.plan
  model_1/
    config.pbtxt
    output0_labels.txt
    output1_labels.txt
    3/
      model.graphdef
  model_2/
    …
  model_n/
```

Any number of models may be specified. The name of the model directory (for example, **`model_0`**, **`model_1`**) must match the name of the model specified in the required configuration file, **`config.pbtxt`**. This model name is used in the client and server APIs to identify the model. Each model directory must have at least one numeric subdirectory (for example, **`model_0/1`**). Each of these subdirectories holds a version of the model with the version number corresponding to the directory name. Within the version directory is the model definition file. The name must be **`model.plan`** for TensorRT models, and **`model.graphdef`** for TensorFlow GraphDef models.

The configuration file, **`config.pbtxt`**, for each model must be protobuf text adhering to the ModelConfig schema defined and explained below. The **`*_labels.txt`** files are optional and are used to provide labels for outputs that represent classifications.

5.1. Model Configuration Schema

Each model in the model store must include a file called **`config.pbtxt`** that contains the configuration information for the model. The model configuration must be specified as protobuf text using the ModelConfig schema described at GitHub: Inference Server model_config.proto.

The following example configuration file is for a TensorRT MNIST model that accepts a single “data” input tensor of shape [1,28,28] and produces a single “prob” output vector. The output vector is a classification and the labels associated with each class are in **`mnist_labels.txt`**. The Inference Server will run two instances of this model on GPU 0 so that two **`trt_mnist`** inference requests can be handled simultaneously. Batch sizes up to 8 will be accepted by the server.

```
name: "trt_mnist"
model_platform: "tensorrt_plan"
max_batch_size: 8
input [
  {
    name: "data"
    data_type: TYPE_FP32
    format: FORMAT_NCHW
    dims: [ 1, 28, 28 ]
  }
]
output [
  {
    name: "prob"
    data_type: TYPE_FP32
    dims: [ 10, 1, 1 ]
    label_filename: "mnist_labels.txt"
  }
]
instance [
  {
    gpus: [ 0 ]
  },
  {
    gpus: [ 0 ]
  }
]
```

The next example configuration file is for a TensorFlow ResNet-50 GraphDef model that accepts a single input tensor named “input” in HWC format with shape [224,224,3] and produces a single output vector named “output”. The Inference Server will run two instances of this model, one on GPU 0 and one on GPU 1. Batch sizes up to 128 will be accepted by the server.

```
name: "resnet50"
model_platform: "tensorflow_graphdef"
max_batch_size: 128
input [
  {
    name: "input"
    data_type: TYPE_FP32
    format: FORMAT_NHWC
    dims: [ 224, 224, 3 ]
  }
]
output [
  {
    name: "output"
    data_type: TYPE_FP32
    dims: [ 1000 ]
  }
]
instance [
  {
    gpus: [ 0 ]
  },
  {
    gpus: [ 1 ]
  }
]
```

Chapter 6. INFERENCE SERVER API

The Inference Server can be accessed directly using two exposed HTTP endpoints:

`/api/status`
: The server status API for getting information about the server and about the models being served.

`/api/infer`
: The inference API that accepts model inputs, runs inference and returns the requested outputs.

The HTTP endpoints can be used directly as described in this section, but for most use-cases, the preferred way to access the Inference Server is via the C++ and Python client API libraries. The libraries are available at GitHub: Inference Server.

6.1. Server Status API

Performing an **`HTTP GET`** to **`/api/status`** returns status information about the server and all the models being served. Performing an **`HTTP GET`** to **`/api/status/<model name>`** returns information about the server and the single model specified by **`<model name>`**. An example is shown in Verifying The Inference Server.

The server status is returned in the HTTP response body in either text format (the default) or in binary format if query parameter **`format=binary`** is specified (for example, **`/api/status?format=binary`**). The status schema is defined by the protobuf schema given in **`server_status.proto`** defined at GitHub: Inference Server server_status.proto.

The success or failure of the server status request is indicated in the HTTP response code and the **`NV-Status`** response header. The **`NV-Status`** response header returns a text protobuf formatted status following the **`status.proto`** schema defined at GitHub: Inference Server status.proto. If the request is successful the HTTP status is 200 and the **`NV-Status`** response header will indicate no failure:

```
NV-Status: code: SUCCESS
```

If the server status request fails, the response body will be empty, a non-200 HTTP status will be returned and the **`NV-Status`** header will indicate the failure reason, for example:

```
NV-Status: code: NOT_FOUND msg: "no status available for unknown model \'x\'"
```

6.2. Infer

Performing an HTTP POST to **`/api/infer/<model name>`** performs inference using the **`<model name>`** model. The request uses the **`NV-InferRequest`** header to communicate an **`InferRequestHeader`** protobuf message that describes the input tensors and the requested output tensors as defined at GitHub: Inference Server api.proto. For example, for the ResNet-50 example shown in Model Configuration Schema the following **`NV-InferRequest`** header indicates that a batch-size 1 request is being made with input size of 602112 bytes (3 * 224 * 224 * sizeof(FP32)), and that the result of the “output” tensor should be returned as the top-3 classification values.

```
NV-InferRequest: batch_size: 1 input { name: "input" byte_size: 602112 } output
 { name: "output" byte_size: 4000 cls { count: 3 } }
```

The input tensor values are communicated in the body of the HTTP POST request as raw binary in the order as the inputs are listed in the request header.

The inference results are returned in the body of the HTTP response to the POST request. For outputs where full result tensors were requested, the result values are communicated in the body of the response in the order as the outputs are listed in the request header. After those, the binary encoding of an **`InferResponseHeader`** message is appended to the response body. For example, assuming outputs specified in the **`InferRequestHeader`** in order are **`output0`**, **`output1`**, …, **`outputn`**, the response body would contain:

```
<raw binary tensor values for output0, if raw output was requested for output0>
<raw binary tensor values for output1, if raw output was requested for output1>
...
<raw binary tensor values for outputn, if raw output was requested for outputn>
<binary encoded InferResponseHeader proto>
```

The success or failure of the inference request is indicated in the HTTP response code and the **`NV-Status`** response header. The **`NV-Status`** response header returns a text protobuf formatted status following the **`status.proto`** schema. If the request is successful the HTTP status is 200 and the **`NV-Status`** response header will indicate no failure:

```
NV-Status: code: SUCCESS
```

If the inference request fails, a non-200 HTTP status will be returned and the **`NV-Status`** header will indicate the failure reason, for example:

```
NV-Status: code: NOT_FOUND msg: "no status available for unknown model \'x\'"
```

Chapter 7. SUPPORT

For questions, view bug reports and ask for feature requests for the Inference Server, create your ask in the Inference Server issue tracker at GitHub: Inference Server Issues.