TENSORRT INFERENCE SERVER

DU-08994-001 _v0.7.0 | 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. We will refer to the NVIDIA TensorRT Inference Server simply as Inference Server for the remainder of this guide.

The server provides an inference service via an HTTP or gRPC 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, TensorFlow GraphDef, TensorFlow SavedModel and Caffe2 NetDef model formats. Also supports TensorFlow-TensorRT integrated models.

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
: For models that support batching, the server can accept requests for a batch of inputs and respond with the corresponding batch of outputs. The server also supports dynamic batching where individual inference requests are dynamically combined together to improve inference throughput. Dynamic batching is transparent to the client requesting inference.

Model repositories may reside on a locally accessible file system or in Google Cloud Storage.

Readiness and liveness health endpoints suitable for Kubernetes-style orchestration

Prometheus metric support

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

1.1. Architecture

The following figure shows the high-level architecture of the inference server. The Model Repository is a file-system based store of the models that the inference server will make available for inferencing. Inference requests arrive at the server via either HTTP or gRPC and are then routed to the appropriate per-model scheduler queue. The scheduler performs fair scheduling and Dynamic Batching for each model's requests. As execution contexts become available the scheduler passes each request to the framework backend corresponding to the model type. The framework backend performs inferencing using the inputs provided in the request to produce the requested outputs. The outputs are then formatted and a response is sent.

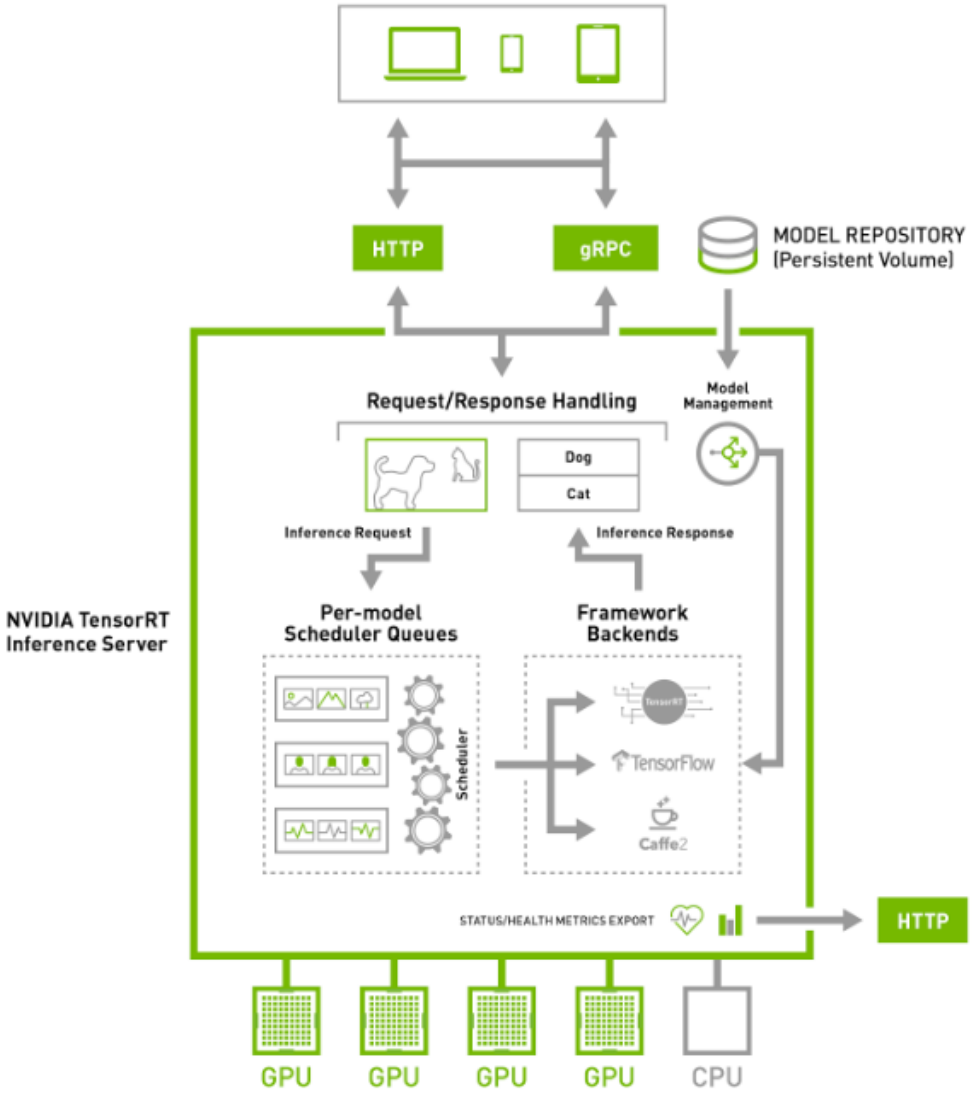

Figure 1 High-level architecture of the inference server

1.2. Contents Of The inference server Container

The TensorRT inference server architecture allows multiple models and/or multiple instances of the same model to execute in parallel on a single GPU. The following figure shows an example with two models; **`model0`** and **`model1`**. Assuming the inference server is not currently processing any request, when two requests arrive simultaneously, one for each model, the inference server immediately schedules both of them onto the GPU and the GPU's hardware scheduler begins working on both computations in parallel.

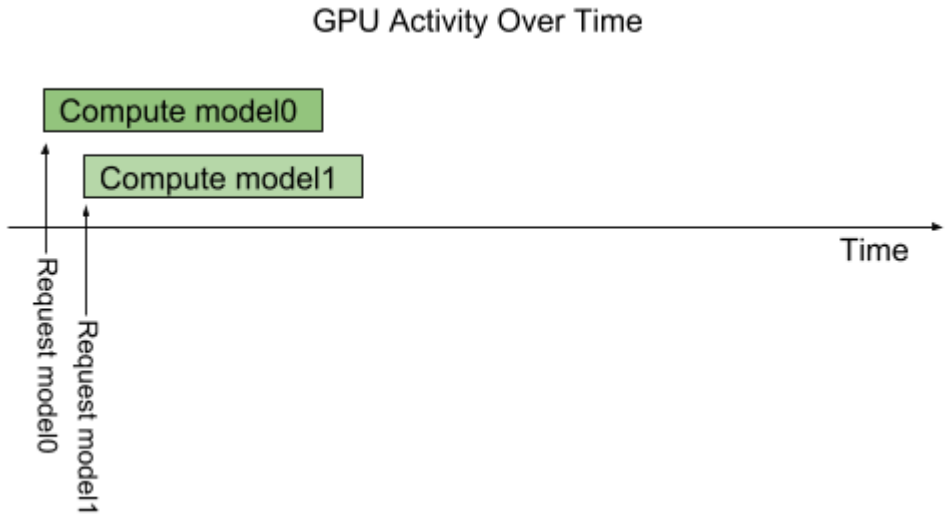

Figure 2 Multiple models each sending a request to the inference server.

By default, if multiple requests for the same model arrive at the same time, the inference server will serialize their execution by scheduling only one at a time on the GPU, as shown in the following figure.

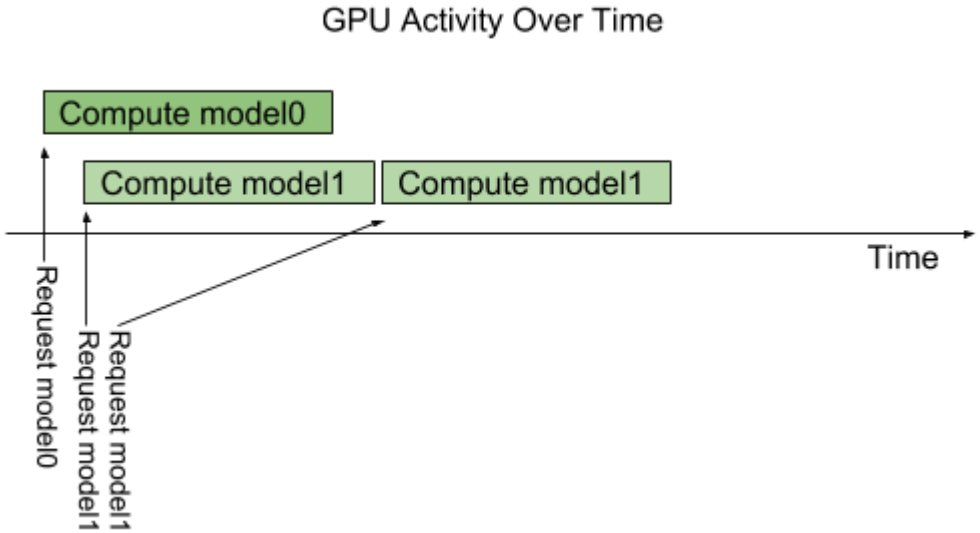

Figure 3 Multiple requests from the same model are sent to the inference server.

The inference server provides an Instance Groups feature that allows each model to specify how many parallel executions of that model should be allowed. Each such enabled parallel execution is referred to as an *execution instance*. By default, the inference server gives each model a single execution instance, which means that only a single execution of the model is allowed to be in progress at a time, as shown in the above figure. By using instance-group the number of execution instances for a model can be increased. The following figure shows model execution when **`model1`** is configured to allow three execution instances. As shown in the figure, the first three **`model1`** inference requests are immediately executed in parallel on the GPU. The fourth **`model1`** inference request must wait until one of the first three executions completes before beginning.

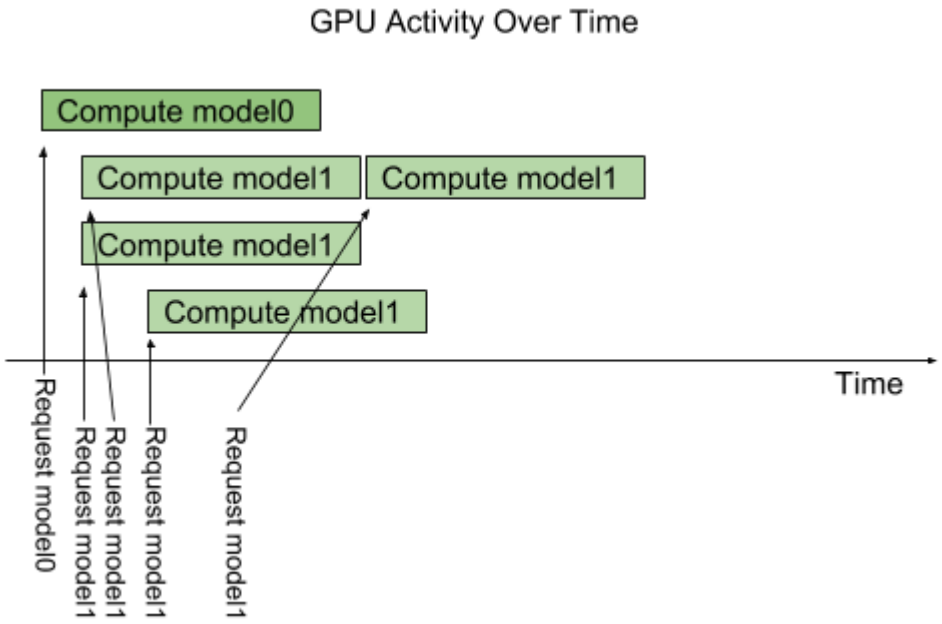

Figure 4 Showing model execution; one model is configured to allow three execution instances.

To provide the current model execution capabilities shown in the above figures, the inference server uses CUDA streams to exploit the GPU's hardware scheduling capabilities. CUDA streams allow the inference server to communicate independent sequences of memory-copy and kernel executions to the GPU. The hardware scheduler in the GPU takes advantage of the independent execution streams to fill the GPU with independent memory-copy and kernel executions. For example, using streams allows the GPU to execute a memory-copy for one model, a kernel for another model, and a different kernel for yet another model at the same time.

The following figure shows some details of how this works within the inference server. Each framework backend (for example, TensorRT, TensorFlow, Caffe2) provides an API for creating an *execution context* that is used to execute a given model (each framework uses different terminology for this concept but here we refer to them generally as execution contexts). Each framework allows an execution context to be associated with a CUDA stream. This CUDA stream is used by the framework to execute all memory copies and kernels needed for the model associated with the execution context. For a given model, the inference server creates one execution context for each execution

instance specified for the model. When an inference request arrives for a given model, that request is queued in the model scheduler associated with that model. The model scheduler waits for any execution context associated with that model to be idle and then sends the queued request to the context. The execution context then issues all the memory copies and kernel executions required to execute the model to the CUDA stream associated with that execution context. The memory copies and kernels in each CUDA stream are independent of memory copies and kernels in other CUDA streams. The GPU hardware scheduler looks across all CUDA streams to find independent memory copies and kernels to execute on the GPU.

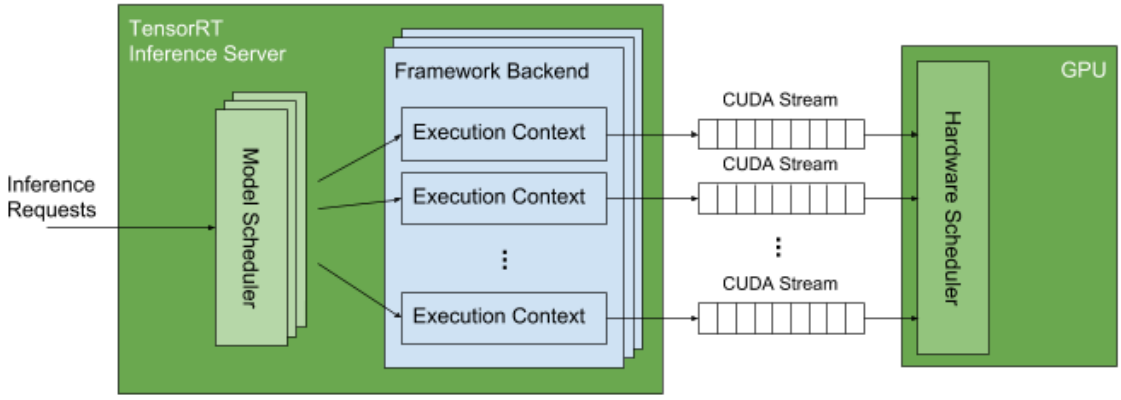

Figure 5 The inference server uses CUDA streams to exploit the GPU's hardware scheduling capabilities.

Chapter 2. PULLING THE INFERENCE SERVER CONTAINER

Before you can pull a container from the NGC container registry, you must have Docker and nvidia-docker installed. For DGX users, this is explained in Preparing to use NVIDIA Containers Getting Started Guide.

For users other than DGX, follow the NVIDIA® GPU Cloud™ (NGC) container registry nvidia-docker installation documentation based on your platform.

You must also have access and be logged into the NGC container registry as explained in the NGC Getting Started Guide.

There are the four repositories where you can find the NGC docker containers.

`nvcr.io/nvidia/`
: The deep learning framework containers are stored in the **`nvcr.io/nvidia/`** repository.

`nvcr.io/hpc`
: The HPC containers are stored in the **`nvcr.io/hpc`** repository.

`nvcr.io/nvidia-hpcvis`
: The HPC visualization containers are stored in the **`nvcr.io/nvidia-hpcvis`** repository.

`nvcr.io/partner`
: The partner containers are stored in the **`nvcr.io/partner`** repository. Currently the partner containers are focused on Deep Learning or Machine Learning, but that doesn't mean they are limited to those types of containers.

Chapter 3. RUNNING THE INFERENCE SERVER CONTAINER

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

```
$ nvidia-docker run --rm --shm-size=1g --ulimit memlock=-1 --ulimit
 stack=67108864 -p8000:8000 -p8001:8001 -p8002:8002 -v/path/to/model/
repository:/tmp/models tensorrtserver:18.xx-py3 /opt/tensorrtserver/bin/
trtserver --model-store=/tmp/models
```

Where **tensorrtserver:18.xx-py3** is the container that was pulled from the NGC container registry as described in Pulling The inference server Container.

The **nvidia-docker -v** option maps **/path/to/model/repository** on the host into the container at **/tmp/models**, and the **--model-store** option to the inference server is used to point to **/tmp/models** as the model repository.

The inference server:

- Listens for HTTP requests on port 8000
- Listens for gRPC requests on port 8001
- Provides Prometheus metrics on port 8002

and the above command uses the **-p** flag to map container ports 8000, 8001, 8002 to host ports 8000, 8001, 8002. A different host port can be used by modifying the **-p** flag, for example **-p9000:8000** will cause the inference server HTTP endpoint 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 is 1g but larger sizes may be necessary depending on the number and size of models being served.

For more information, see Inference Server HTTP API and Inference Server gRPC API. For more information on the Prometheus metrics provided by the inference server, see Metrics.

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 HTTP API. From the host system use **`curl`** to the HTTP endpoint 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
id: "inference:0"
version: "0.6.0"
uptime_ns: 23322988571
model_status {
  key: "resnet50_netdef"
  value {
    config {
      name: "resnet50_netdef"
      platform: "caffe2_netdef"
    }
    ...
    version_status {
      key: 1
      value {
        ready_state: MODEL_READY
      }
    }
  }
}
ready_state: SERVER_READY
```

This status shows configuration information as well as indicating that version 1 of the **`resnet50_netdef`** model is **`MODEL_READY`**, indicating the Inference Server is ready to accept inferencing requests for version 1 of that model. A model version **`ready_state`** will show up as **`MODEL_UNAVAILABLE`** if the model failed to load for some reason or if it was unloaded due to the model version policy discussed here.

Chapter 5. HEALTH ENDPOINTS

The Inference Server provides readiness and liveness HTTP endpoints that are useful for determining the general state of the service. These endpoints are useful for orchestration frameworks like Kubernetes. For more information on the health endpoints, see Inference Server HTTP API section.

Chapter 6. MODEL REPOSITORY

The inference server accesses models from a locally accessible file path or from Google Cloud Storage. This path is specified when the server is started using the **--model-store** option.

For a locally accessible file path the absolute path must be specified, for example, **--model-store=/path/to/model/repository**. For a model repository residing in Google Cloud Storage, the path must be prefixed with **gs://**, for example, **--model-store=gs://bucket/path/to/model/repository**. The model store must be organized as follows:

```
<model-repository path>/
  model_0/
    config.pbtxt
    output0_labels.txt
    1/
      model.plan
    2/
      model.plan
  model_1/
    config.pbtxt
    output0_labels.txt
    output1_labels.txt
    0/
      model.graphdef
    7/
      model.graphdef
  model_2/
    …
  model_n/
```

Any number of models may be specified, however, after the inference server is started, models cannot be added to or removed from the model repository. To add or remove a model:

1. Stop the inference server.
2. Update the model repository.
3. Start the inference server.

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, **`mymodel/1`**). Each of these subdirectories holds a version of the model with the version number corresponding to the directory name. Version subdirectories can be added and removed from the model repository while the inference server is running.

For more information about how the model versions are handled by the server, see Model Versions. Within each version subdirectory, there is one or more model definition files. For more information about the model definition files contained in each version subdirectory, see Model Definition.

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.

6.1. Model Versions

Each model can have one or more versions available in the model repository. Each version is stored in its own, numerically named, subdirectory where the name of the subdirectory corresponds to the version number of the model. Version subdirectories can be added and removed while the inference server is running to add and remove the corresponding model versions. Each model specifies a *version policy* that controls which of the versions in the model repository are made available by the inference server at any given time. The **`ModelVersionPolicy`** portion, as described in Model Configuration Schema specifies one of the following policies.

All
: All versions of the model that are specified in the model repository are available for inferencing.

Latest
: Only the latest **`n`** versions of the model specified in the model repository are available for inferencing. The latest versions of the model are the numerically greatest version numbers.

Specific
: The specifically listed versions of the model are available for inferencing.

If no version policy is specified, then **`Latest`** (with **`num_version = 1`**) is used as the default, indicating that only the most recent version of the model is made available by the inference server. In all cases, the addition or removal of version subdirectories from the model repository can change which model version is used on subsequent inference requests.

6.2. Model Definition

Each model version subdirectory must contain at least one model definition. By default, the name of this file or directory must be:

- **`model.plan`** file for TensorRT models
- **`model.graphdef`** file for TensorFlow GraphDef models

- **model.savedmodel** directory for TensorFlow SavedModel models
- **model.netdef, init_model.netdef** files for Caffe2 Netdef models

The default can be overridden using the **default_model_filename** property in Model Configuration Schema.

Optionally, a model can provide multiple model definition files, each targeted at a GPU with a different Compute Capability. Most commonly, this feature is needed for TensorRT and TensorFlow to TensorRT integrated models where the model definition is valid for only a single compute capability. See the **trt_mnist** configuration in Model Configuration Schema for an example.

An example model repository is available at Deep Learning Inference Server Clients.

6.3. Model Configuration Schema

Each model in the model repository 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. Two model definition files are provided for each version of this model, one for compute capability 6.1 called **model6_1.plan** and another for compute capability 7.0 called **model7_0.plan**.

```
name: "trt_mnist"
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"
  }
]
cc_model_filenames [
  {
    key: "6.1"
    value: "model6_1.plan"
  },
  {
    key: "7.0"
```

```
    value: "model7_0.plan"
  }
]
instance_group [
  {
    count: 2
    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"
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_group [
  {
    gpus: [ 0 ]
  },
  {
    gpus: [ 1 ]
  }
]
```

6.3.1. TensorFlow GraphDef And SavedModel Models

TensorFlow saves trained models in one of two ways: *GraphDef* and *SavedModel* and the inference server supports both of these formats. Once you have a trained model in TensorFlow, you can save it as a GraphDef directly or convert it to a GraphDef by using a script like freeze_graph.py, or save it as a SavedModel using a SavedModelBuilder or tf.saved_model.simple_save.

The configuration platform value for TensorFlow GraphDef models must be **`tensorflow_graphdef`** and the model definition file must be named **`model.graphdef`** (unless the **`default_model_filename`** property is set in the model configuration).

The configuration platform value for TensorFlow SavedModel models must be **`tensorflow_savedmodel`** and the model definition directory must be named **`model.savedmodel`** (unless the **`default_model_filename`** property is set in the model configuration).

TensorFlow 1.7 and later integrates TensorRT to enable TensorFlow models to benefit from the inference optimizations provided by TensorRT. Because the Inference Server supports models that have been optimized with TensorRT, it can serve those models just like any other TensorFlow model. The Inference Server's TensorRT version (available in the Inference Server Container Release Notes must match the TensorRT version that was used when the model was created.

The following 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`**. Batch sizes up to 128 will be accepted by the server.

```
name: "resnet50"
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 ]
  }
]
```

6.3.2. TensorRT PLAN Models

The configuration platform value for TensorRT PLAN models must be **`tensorrt_plan`** and the model definition file must be named **`model.plan`** (unless the **`default_model_filename`** property is set in the model configuration).

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`**. Batch sizes up to 8 will be accepted by the server. Two model definition files are provided for each version of this model, one for compute capability 6.1 called **`model6_1.plan`** and another for compute capability 7.0 called **`model7_0.plan`**.

```
name: "trt_mnist"
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"
  }
]
cc_model_filenames [
  {
    key: "6.1"
    value: "model6_1.plan"
  },
  {
    key: "7.0"
    value: "model7_0.plan"
  }
]
```

The model repository for this model would look like:

```
model_repository/
  trt_mnist/
    config.pbtxt
    mnist_labels.txt
    1/
      model6_1.plan
      model7_0.plan
```

6.3.3. Caffe2 NetDef Models

The configuration platform value for Caffe2 NetDef models must be **`caffe2_netdef`**. NetDef model definition is split across two files: the initialization network and the predict network. These files must be named **`init_model.netdef`** and **`model.netdef`** (unless the **`default_model_filename`** property is set in the model configuration).

An example of a model repository for a Caffe2 NetDef model is:

```
model_repository/
  netdef_mnist/
    config.pbtxt
    mnist_labels.txt
    1/
      init_model.netdef
      model.netdef
```

6.3.4. ONNX Models

The Inference Server cannot directly perform inferencing using ONNX models. An ONNX model must be converted to either TensorRT PLAN or Caffe2 NetDef to be served by the Inference Server. To convert your ONNX model to a TensorRT PLAN use either the ONNX Parser included in TensorRT or the open-source TensorRT backend for ONNX. Another option is to convert your ONNX model to Caffe2 NetDef as described here.

6.3.5. Instance Groups

The inference server can provide multiple *instances* of a model so that multiple simultaneous inference requests for that model can be handled simultaneously. The

model configuration **instance-group** setting is used to specify the number of model instances that should be made available and what compute resource should be used for those instances.

For example, the following model configuration setting will cause two instances of the model to be available on each system GPU.

```
instance_group [
  {
    count: 2
    kind: KIND_GPU
  }]
```

For more information on instance-group settings, see the Model Configuration protobuf documentation.

6.3.6. Dynamic Batching

The inference server supports batch inferencing by allowing individual inference requests to specify a batch of inputs. The inferencing for a batch of inputs is processed at the same time which is especially important for GPUs since it can greatly increase inferencing throughput. Unfortunately, in many use-cases the individual inference requests are not batched, therefore, they do not benefit from the throughput benefits of batching.

Dynamic batching is a feature of the inference server that allows non-batched inference requests to be combined by the inference server, so that a batch is created dynamically, resulting in the same increased throughput seen for batched inference requests.

Dynamic batching is enabled and configured independently for each model using the **dynamic_batching** settings in the model configuration. The configuration can control the preferred sizes of the dynamically created batches as well as a maximum time that requests can be delayed in the scheduler to allow other requests to join the dynamic batch. The following example configuration enables dynamic batching with a preferred batch size of 8 and a maximum delay time of 100 microseconds.

```
dynamic_batching {
  preferred_batch_size: [ 8 ]
  max_queue_delay_microseconds: 100
}
```

Chapter 7. INFERENCE SERVER HTTP API

The inference server can be accessed directly using three exposed HTTP endpoints:

`/api/health`
: The server health API for determining server liveness and readiness.

`/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.

7.1. Health

Performing an **`HTTP GET`** to **`/api/health/live`** returns a 200 status if the server is able to receive and process requests. Any other status code indicates that the server is still initializing or has failed in some way that prevents it from processing requests.

Once the liveness endpoint indicates that the server is active, performing an **`HTTP GET`** to **`/api/health/ready`** returns a 200 status if the server is able to respond to inference requests for some or all models (based on the **`--strict-readiness`** option explained below). Any other status code indicates that the server is not ready to respond to some or all inference requests. Typically, when the readiness endpoint returns a non-200 status you should not send any more inference requests to the server.

By default, the readiness endpoint will return 200 status only if the server is responsive and all models loaded successfully. Thus, by default, a 200 status indicates that an inference request for any model can be handled by the server. For some use cases, you want the readiness endpoint to return 200 status even if all models are not available. In this case, use the **`--strict-readiness=false`** option to cause the readiness endpoint to report 200 status as long as the server is responsive (even if one or more models are not available).

7.2. Server Status

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\'"
```

7.3. Infer

Performing an HTTP POST to **/api/infer/<model name>** performs inference using the latest available version of **<model name>** model. The latest available version is the numerically greatest version number. Performing an HTTP POST to **/api/infer/<model name>/<model version>** performs inference using a specific version of the model.

In either case, 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, an **InferResponseHeader** message is appended to the response body. The **InferResponseHeader** message is returned in either text format (the default) or in binary format if query parameter **format=binary** is specified (for example, **/api/infer/foo?format=binary**).

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>
<text or 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 8. INFERENCE SERVER GRPC API

The Inference Server can be accessed directly using gRPC endpoints defined at grpc_service.proto.

- **`GRPCServer.Status`**: The server status API for getting information about the server and about the models being served.
- **`GRPCServer.Infer`**: The inference API that accepts model inputs, runs inference and returns the requested outputs.

The gRPC endpoints can be used via the gRPC generated client (demonstrated in the image classification example at grpc_image_client.py) or via the C++ and Python client API libraries. Build instructions for the gRPC client libraries are available at Deep Learning Inference Server clients.

Chapter 9. METRICS

The inference server provides Prometheus metrics indicating GPU and request statistics. By default, these metrics are available at http://localhost:8002/metrics. The inference server `--metrics-port` option can be used to select a different port. The following table describes the available metrics.

Table 1 Prometheus metrics indicating GPU and request statistics

	Use Case	Granularity	Frequency
Power usage	Proxy for load on the GPU	Per GPU	Per second
Power limit	Maximum GPU power limit	Per GPU	Per second
GPU utilization	GPU utilization rate (0.0 - 1.0)	Per GPU	Per second
Request count	Number of inference requests	Per model	Per request
Execution count	▸ Number of model inference executions ▸ Request count / Execution count = Average dynamic request batching	Per model	Per request
Inference count	Number of inferences performed (one request counts as "batch size" inferences)	Per model	Per request
Latency: request time	End-to-end inference request handling time	Per model	Per request
Latency: compute time	Amount of time a request spends executing the	Per model	Per request

	Use Case	Granularity	Frequency
	inference model (in the appropriate framework)		
Latency: queue time	Amount of time a request spends waiting in the queue before being executed	Per model	Per request

Chapter 10. SUPPORT

For questions and feature requests, use the Inference Server Devtalk forum.

Notice

Trademarks

Copyright