Optimization

The Triton Inference Server has many features that you can use to decrease latency and increase throughput for your model. This section discusses these features and demonstrates how you can use them to improve the performance of your model. As a prerequisite you should follow the QuickStart to get Triton and client examples running with the example model repository.

This section focuses on understanding latency and throughput tradeoffs for a single model. The Model Analyzer section describes a tool that helps you understand the GPU memory utilization of your models so you can decide how to best run multiple models on a single GPU.

Unless you already have a client application suitable for measuring the performance of your model on Triton, you should familiarize yourself with Performance Analyzer. The Performance Analyzer is an essential tool for optimizing your model's performance.

As a running example demonstrating the optimization features and options, we will use a ONNX Inception model that you can obtain by following the QuickStart. As a baseline we use perf_analyzer to determine the performance of the model using a basic model configuration that does not enable any performance features.

```
$ perf_analyzer -m inception_graphdef --percentile=95 --concurrency-range
1:4
...
Inferences/Second vs. Client p95 Batch Latency
Concurrency: 1, throughput: 62.6 infer/sec, latency 21371 usec
Concurrency: 2, throughput: 73.2 infer/sec, latency 34381 usec
Concurrency: 3, throughput: 73.2 infer/sec, latency 50298 usec
Concurrency: 4, throughput: 73.4 infer/sec, latency 65569 usec
```

The results show that our non-optimized model configuration gives a throughput of about 73 inferences per second. Note how there is a significant throughput increase going from one concurrent request to two concurrent requests and then throughput levels off. With one concurrent request Triton is idle during the time when the response is returned to the client and the next request is received at the server. Throughput increases with a concurrency of two because Triton overlaps the processing of one request with the communication of the other. Because we are running perf_analyzer on the same system as Triton, two requests are enough to completely hide the communication latency.

Optimization Settings

For most models, the Triton feature that provides the largest performance improvement is dynamic batching. This example sheds more light on conceptual details. If your model does not support batching then you can skip ahead to Model Instances.

Dynamic Batcher

The dynamic batcher combines individual inference requests into a larger batch that will often execute much more efficiently than executing the individual requests independently. To enable the dynamic batcher stop Triton, add the following line to the end of the model configuration file for inception_graphdef, and then restart Triton.

```
dynamic_batching { }
```

The dynamic batcher allows Triton to handle a higher number of concurrent requests because those requests are combined for inference. To see this run perf_analyzer with request concurrency from 1 to 8.

```
$ perf_analyzer -m inception_graphdef --percentile=95 --concurrency-range
1:8
...
Inferences/Second vs. Client p95 Batch Latency
Concurrency: 1, throughput: 66.8 infer/sec, latency 19785 usec
Concurrency: 2, throughput: 80.8 infer/sec, latency 30732 usec
Concurrency: 3, throughput: 118 infer/sec, latency 32968 usec
Concurrency: 4, throughput: 165.2 infer/sec, latency 32974 usec
Concurrency: 5, throughput: 194.4 infer/sec, latency 33035 usec
Concurrency: 6, throughput: 217.6 infer/sec, latency 34258 usec
Concurrency: 7, throughput: 249.8 infer/sec, latency 34522 usec
Concurrency: 8, throughput: 272 infer/sec, latency 35988 usec
```

With eight concurrent requests the dynamic batcher allows Triton to provide 272 inferences per second without increasing latency compared to not using the dynamic batcher.

Instead of having perf_analyzer collect data for a range of request concurrency values we can instead use a couple of simple rules that typically applies when perf_analyzer is running on the same system as Triton. The first rule is that for minimum latency set the request concurrency to 1 and disable the dynamic batcher and use only 1 model instance. The second rule is that for maximum throughput set the request concurrency to be `2 * <maximum batch size> * <model instance count>`. We will discuss model instances below, for now we are working with one model instance. So for maximum-batch-size 4 we want to run perf_analyzer with request concurrency of `2 * 4 * 1 = 8`.

```
$ perf_analyzer -m inception_graphdef --percentile=95 --concurrency-range
8
...
Inferences/Second vs. Client p95 Batch Latency
Concurrency: 8, throughput: 267.8 infer/sec, latency 35590 usec
```

Model Instances

Triton allows you to specify how many copies of each model you want to make available for inferencing. By default you get one copy of each model, but you can specify any number of

instances in the model configuration by using instance groups. Typically, having two instances of a model will improve performance because it allows overlap of memory transfer operations (for example, CPU to/from GPU) with inference compute. Multiple instances also improve GPU utilization by allowing more inference work to be executed simultaneously on the GPU. Smaller models may benefit from more than two instances; you can use perf_analyzer to experiment.

To specify two instances of the inception_graphdef model: stop Triton, remove any dynamic batching settings you may have previously added to the model configuration (we discuss combining dynamic batcher and multiple model instances below), add the following lines to the end of the model configuration file, and then restart Triton.

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

Now run perf_analyzer using the same options as for the baseline.

```
$ perf_analyzer -m inception_graphdef --percentile=95 --concurrency-range
1:4
...
Inferences/Second vs. Client p95 Batch Latency
Concurrency: 1, throughput: 70.6 infer/sec, latency 19547 usec
Concurrency: 2, throughput: 106.6 infer/sec, latency 23532 usec
Concurrency: 3, throughput: 110.2 infer/sec, latency 36649 usec
Concurrency: 4, throughput: 108.6 infer/sec, latency 43588 usec
```

In this case having two instances of the model increases throughput from about 73 inference per second to about 110 inferences per second compared with one instance.

It is possible to enable both the dynamic batcher and multiple model instances, for example, change the model configuration file to include the following.

```
dynamic_batching { }
instance_group [ { count: 2 }]
```

When we run perf_analyzer with the same options used for just the dynamic batcher above.

```
$ perf_analyzer -m inception_graphdef --percentile=95 --concurrency-range
16
...
Inferences/Second vs. Client p95 Batch Latency
Concurrency: 16, throughput: 289.6 infer/sec, latency 59817 usec
```

We see that two instances does not improve throughput much while increasing latency, compared with just using the dynamic batcher and one instance. This occurs because for this model the dynamic batcher alone is capable of fully utilizing the GPU and so adding additional model instances does not provide any performance advantage. In general the benefit of the dynamic batcher and multiple instances is model specific, so you should experiment with perf_analyzer to determine the settings that best satisfy your throughput and latency requirements.

Framework-Specific Optimization

Triton has several optimization settings that apply to only a subset of the supported model frameworks. These optimization settings are controlled by the model configuration optimization policy. Visit this guide for an end to end discussion.

ONNX with TensorRT Optimization (ORT-TRT)

One especially powerful optimization is to use TensorRT in conjunction with an ONNX model. As an example of TensorRT optimization applied to an ONNX model, we will use an ONNX DenseNet model that you can obtain by following QuickStart. As a baseline we use perf_analyzer to determine the performance of the model using a basic model configuration that does not enable any performance features.

```
$ perf_analyzer -m densenet_onnx --percentile=95 --concurrency-range 1:4
...
Inferences/Second vs. Client p95 Batch Latency
Concurrency: 1, 113.2 infer/sec, latency 8939 usec
Concurrency: 2, 138.2 infer/sec, latency 14548 usec
Concurrency: 3, 137.2 infer/sec, latency 21947 usec
Concurrency: 4, 136.8 infer/sec, latency 29661 usec
```

To enable TensorRT optimization for the model: stop Triton, add the following lines to the end of the model configuration file, and then restart Triton.

```
optimization { execution_accelerators {
  gpu_execution_accelerator : [ {
    name : "tensorrt"
    parameters { key: "precision_mode" value: "FP16" }
    parameters { key: "max_workspace_size_bytes" value: "1073741824" }
    }]
}}
```

As Triton starts you should check the console output and wait until Triton prints the “Staring endpoints” message. ONNX model loading can be significantly slower when TensorRT optimization is enabled. In production you can use model warmup to avoid this model startup/optimization slowdown. Now run perf_analyzer using the same options as for the baseline.

```
$ perf_analyzer -m densenet_onnx --percentile=95 --concurrency-range 1:4
...
Inferences/Second vs. Client p95 Batch Latency
Concurrency: 1, 190.6 infer/sec, latency 5384 usec
Concurrency: 2, 273.8 infer/sec, latency 7347 usec
Concurrency: 3, 272.2 infer/sec, latency 11046 usec
Concurrency: 4, 266.8 infer/sec, latency 15089 usec
```

The TensorRT optimization provided 2x throughput improvement while cutting latency in half. The benefit provided by TensorRT will vary based on the model, but in general it can provide significant performance improvement.

ONNX with OpenVINO Optimization

ONNX models running on the CPU can also be accelerated by using OpenVINO. To enable OpenVINO optimization for an ONNX model, add the following lines to the end of the model's configuration file.

```
optimization { execution_accelerators {
  cpu_execution_accelerator : [ {
    name : "openvino"
  }]
}}
```

NUMA Optimization

Many modern CPUs are composed of multiple cores, memories and interconnects that expose different performance characteristics depending on how threads and data are allocated. Triton allows you to set host policies that describe this NUMA configuration for your system and then assign model instances to different host policies to exploit these NUMA properties.

Host Policy

Triton allows you to specify host policy that associates with a policy name on startup. A host policy will be applied to a model instance if the instance is specified with the same policy name by using host policy field in instance groups. Note that if not specified, the host policy field will be set to default name based on the instance property.

To specify a host policy, you can specify the following in command line option:

```
--host-policy=<policy_name>,<setting>=<value>
```

Currently, the supported settings are the following:

- *numa-node*: The NUMA node id that the host policy will be bound to, the host policy restricts memory allocation to the node specified.
- *cpu-cores*: The CPU cores to be run on, the instance with this host policy set will be running on one of those CPU cores.

Assuming that the system is configured to bind GPU 0 with NUMA node 0 which has CPU cores from 0 to 15, the following shows setting the numa-node and cpu-cores policies for "gpu_0":

```
$ tritonserver --host-policy=gpu_0,numa-node=0 --host-policy=gpu_0,cpu-
cores=0-15 ...
```