登临 Hamming V2

Triton 发布包使用说明

DL-DG/SW-051A-08

2025-01-13

商标和许可

和其它苏州登临科技有限公司的其它登临科技的图标为苏州登临科技有限公司的商标。本手册中提及的所有其他商标均为其各自所有者的财产。

通知

所购买的产品、服务和特性由苏州登临科技有限公司与客户签订的合同规定。本文件中描述的所有或部分产品、服务和特性可能不在采购范围或使用范围内。除非合同中另有规定，本文件中的所有声明、信息和建议均按“原样”提供，无任何明示或暗示的保证或陈述。
本手册中的信息如有更改，恕不另行通知。本文件在编制过程中已尽一切努力确保内容的准确性，本文件中的所有声明、信息和建议不构成任何明示或暗示的保证。

苏州登临科技有限公司

苏州工业园区扬富路11号南岸新地一期商务楼栋5号1101室，江苏，中国

http://www.denglin.ai

email：support@denglin.ai

更新历史

版本	更新描述
08	适配Driver SDK集成到登临Triton image特性。 文档更名为《登临 Hamming V2 Triton 发布包使用说明》。
07	文档更名为《登临Triton发布包使用说明-V2》。
06	新增dlnne backend配置及使用说明。
05	新增支持qwen2和openchat大模型类型。
04	新增支持qwen和bloom大模型类型。
03	新增章节***第三方开源软件License***。
02	新增依赖安装，新增模型type说明。
01	第一次发布。

目录

1 简介

本文主要介绍`triton`发布包的使用以及基于该发布包如何配置、运行`fastertransformer`大模型。同时，介绍登临dlnne backend支持的模型配置文件，以及如何将dlnne backend挂载到triton server。

2 加载镜像

```
docker load < dl-triton_${tag}.tar
```

通过上面的命令可以把发布包加载为docker镜像，该镜像主要包含以下内容：

1. /opt/tritonserver `triton server`的可执行程序及相关library。
2. /opt/tritonclient `triton`的`c++`和`python`的client相关library和源码。
3. /opt/nccl `nccl`的library、header file以及测试程序。对应的`lib`和`include`目录已经安装至`/usr/local/`目录。
4. /opt/examples `triton`的模型配置示例以及python client的示例。

3 安装依赖环境

3.1 安装登临驱动环境

根据《登临 Hamming V2 驱动安装指南（Linux）》进行安装，无需额外配置。

3.2 安装登临容器运行时（可选）

根据登临容器运行时文档《登临Hamming容器组件使用手册》安装。

4 启动容器

4.1 已安装登临容器运行时

如果已经安装登临容器运行时，请按照下面的命令启动容器。

```
docker run -it --rm --runtime dlrt --shm-size 1g --network host dl-triton:${tag} /bin/bash
```

4.2 未安装登临容器运行时

如果没有安装登临容器运行时，请按照下面的方式启动容器。

运行dlsmi保证已经生成需要的device file

运行下面的命令可以加载登临SDK环境，并且列出当前主机上识别到的登临device，同时也会生成必要的device file。

```
# <sdk_path>表示host端需要加载的登临sdk所在路径
source ${sdk_path}/env.sh
dlsmi
```

启动容器

在没有安装登临容器运行时的情况下，可以通过下面的方式手动将device file和登临SDK挂载到容器内部。

```
# <sdk_path>表示host端需要挂载到容器内的登临SDK所在路径
# <tag>表示需要启动的repository的名为dl-triton的标签（TAG）
docker run -it --rm --shm-size=1g --privileged --network host  dl-triton:${tag} /bin/bash
```

5 在容器内部加载登临 SDK 运行环境

以下指令需要在容器内执行，用于加载登临SDK的环境。

```
source /denglinsdk/sdk/env.sh
```

加载登临SDK环境后，运行下面的指令测试docker环境是否OK。

```
dlsmi
```

如果可以看到所有的登临device列表（如下所示），则说明docker环境已经正常启动。否则请检查上面的步骤是否已经正确执行。

```
root@ubuntu:/opt# dlsmi
Fri Nov 24 15:55:37 2023
+-----------------------------------------------------------------------------+
| DL-SMI 11.0                  Driver version 0.83.0                    |
+-----------------------------------------------------------------------------+
| GPU Product-Type           Bus-Id    Cluster-Memory-Usage    GPU-Util |
| Fan  Temp  Perf  Pwr:Usage/Cap   Memory-Usage  0   1   2   3   MIG M. |
|=============================================================================|
|  0  Goldwasser II XL512          0000:01:00.0                      0% |
|  0 36C  P0  29W / 220W  27237 / 32768  6809  6809  6809 6809 Disabled |
+-----------------------------------------------------------------------------+

+-----------------------------------------------------------------------------+
| Processes                             Cluster-Memory-Usage  GPU-Memory |
| GPU GI     PID  TASK Process name          0    1    2    3   Usage   |
|=============================================================================|
|  No running processes found                                             |
+-----------------------------------------------------------------------------+
```

6 测试 NCCL

由于运行大模型依赖于NCCL的多卡互连，因此请按照 `/opt/nccl/readme.md` 文件里的**运行测试用例**章节的内容进行多卡功能测试。

7 基于 Triton 运行大模型

目前登临支持基于Triton的大模型类型如下：

- Llama

  Llama 类型支持llama和llama2模型。

- baichuan2

  baichuan2类型支持Baichuan2模型。

- chatglm2

  chatglm2类型支持Chatglm2模型和Chatglm3模型。

- qwen

  qwen类型支持qwen模型。

- qwen2

  qwen2类型支持qwen2模型。

- bloom

  bloom类型支持bloom模型。

- openchat

  openchat类型支持openchat模型。

huggingface Llama模型格式转换脚本需要安装依赖：

```
pip install torch --index-url https://download.pytorch.org/whl/cpu
pip install transformers
pip install bfloat16
```

下面将以`Baichuan2-13B-Chat`为例来展示如何使用`triton`基于单机多卡运行大模型，在此之前请确保上面的NCCL测试结果正确。

7.1 准备 Baichuan2-13B-Chat 的模型

`Baichuan2-13B-Chat`的模型可以从huggingface获取。

```
git lfs install
git clone https://huggingface.co/baichuan-inc/Baichuan2-13B-Chat
```

下面的操作都需要在容器内部完成。

7.2 准备 fastertransformer 需要的模型

使用`fastertransformer`提供的工具，根据推理需要的GPU个数，对模型进行拆分。

```
# <save_dir>表示拆分模型后保存的路径，<train_gpu_count>表示用于训练的GPU个数
# <inference_gpu_count>表示用于推理的GPU个数，Baichuan2-13B-Chat_dir>表示使用的Baichuan2-13B-
Chat模型所在的路径
python3 /opt/examples/fastertransformer/convert_scripts/huggingface_baichuan2_convert.py -o
${save_dir} -t_g ${train_gpu_count} \
    -i_g ${inference_gpu_count} -model_name baichuan2 -i ${Baichuan2-13B-Chat_dir} -
weight_data_type fp16
```

7.3 准备 Triton 的模型配置文件

为了方便server的启动，把模型仓库目录`baichuan2`下的配置文件拷贝到一个单独的目录进行修改。

此处以`baichuan2`模型为例，配置文件内的`model_type`已经配置为`baichuan2`。

如果是另外两种模型，请参照以下命令，拷贝目录`/opt/examples/fastertransformer/model_repository/`对应模型名称的配置文件到单独的目录进行修改。

```
# /opt/examples/fastertransformer/model_repository目录下存放了baichuan2，llama，chatglm2模型的
配置文件。
# 以baichuan2模型为例，拷贝模型目录baichuan2下的所有文件和文件夹到单独的目
录/opt/ft_triton_configs/fastertransformer
mkdir -p /opt/ft_triton_configs
cp -r /opt/examples/fastertransformer/model_repository/baichuan2
/opt/ft_triton_configs/fastertransformer
```

然后修改`/opt/ft_triton_configs/fastertransformer/config.pbtxt`文件。

1. 修改`tensor_para_size`为想要运行模型的GPU个数，如果是2个GPU，则设置为2。注意这里的个数必须和上面拆分模型的用于推理的GPU个数（inference_gpu_count）一致。

   ```
   parameters {
     key: "tensor_para_size"
     value: {
       string_value: "2"
     }
   }
   ```

2. 修改`model_checkpoint_path`为上面拆分模型的时候指定的save_dir。因为是在容器里面运行`triton server`，所以路径需要设置为该目录在容器里面的路径。

```
parameters {
  key: "model_checkpoint_path"
  value: {
    string_value: "/ft_workspace/models"
  }
}
```

3. 修改`max_batch_size`，指定支持的最大batch size。

```
max_batch_size: 64
```

4. 修改`max_queue_delay_microseconds`来指定server在收到第一个请求后，等待多少时间来继续等待后续的请求并组合成一个batch执行。
比如，下面的配置表示请求将等待10000微秒，如果这个时间内有新的请求到达，并且总的batch size不超过max_batch_size，将会被合并到同一个batch内进行推理。

```
dynamic_batching {
    max_queue_delay_microseconds: 10000
}
```

7.4 启动 Triton Server

配置修改完成之后，就可以使用下面的命令启动`triton server`了。

需要注意的是，如果我们机器上有4个GPU，但是希望以2个GPU运行，可以在命令行前面加上`CUDA_VISIBLE_DEVICES=0,1`来指定2个GPU，否则启动会报错。

```
/opt/tritonserver/bin/tritonserver --model-repository /opt/ft_triton_configs
```

启动成功后会有如下输出：

```
I1127 04:07:48.371074 48870 grpc_server.cc:2451] Started GRPCInferenceService at
0.0.0.0:8001
I1127 04:07:48.377830 48870 http_server.cc:3558] Started HTTPService at 0.0.0.0:8000
I1127 04:07:48.420399 48870 http_server.cc:187] Started Metrics Service at 0.0.0.0:8002
```

在一台多卡机器上启动多个`triton`实例

我们可以选择启动一个`triton`实例占用所有GPU，也可以选择启动多个`triton`实例每个实例占用部分GPU。

下面将以在8卡机器上启动4个2卡实例来举例介绍如何在一台机上启动多个`triton`实例。

> **注意：**
>
> 不支持在同样的卡上启动多个实例。

这里建议把上面构建好的`/opt/ft_triton_configs`配置目录拷贝到宿主机上，然后启动4个container，同时挂载配置目录和大模型的目录到正确的位置（大模型挂载的目录要和上面修改的config.pbtxt的`model_checkpoint_path`匹配）。

举例如下所示，其中`triton_config_dir`是上面的配置在宿主机的目录，`model_dir`是大模型的目录。

```
# <sdk_path>表示host端需要挂载到容器内的登临sdk所在路径
# <triton_config_dir>表示需要挂载到容器内的triton相关配置的路径
# <model_dir>表示需要挂载到容器内的模型所在路径
# dl-triton表示需要启动的repository的名称为dl-triton
# <tag>表示需要启动的repository的名为dl-triton的tag
docker run -it --rm --shm-size=1g --privileged --network host \
    -v ${triton_config_dir}:/opt/ft_triton_configs -v ${model_dir}:/ft_workspace/models \
    dl-triton:${tag} /bin/bash
```

然后在不同的container里面启动`triton`，并且指定不同的监听端口。

```
# container 1
source /denglinsdk/sdk/env.sh
CUDA_VISIBLE_DEVICES=0,1 /opt/tritonserver/bin/tritonserver --model-repository
/opt/ft_triton_configs --grpc-port 8001 --http-port 8000 --metrics-port 8002

# container 2
source /denglinsdk/sdk/env.sh
CUDA_VISIBLE_DEVICES=2,3 /opt/tritonserver/bin/tritonserver --model-repository
/opt/ft_triton_configs --grpc-port 8011 --http-port 8010 --metrics-port 8012

# container 3
source /denglinsdk/sdk/env.sh
CUDA_VISIBLE_DEVICES=4,5 /opt/tritonserver/bin/tritonserver --model-repository
/opt/ft_triton_configs --grpc-port 8021 --http-port 8020 --metrics-port 8022

# container 4
source /denglinsdk/sdk/env.sh
CUDA_VISIBLE_DEVICES=6,7 /opt/tritonserver/bin/tritonserver --model-repository
/opt/ft_triton_configs --grpc-port 8031 --http-port 8030 --metrics-port 8032
```

7.5 使用 Client 进行测试

Triton server启动后，另外启动一个terminal，运行下面的命令进入容器：

```
# <container_id>为需要进入的容器的ID
docker exec -it ${container_id} bash
```

然后运行下面的命令可以向server发起请求并进行测试：

```
cd /opt/examples/fastertransformer/client
/opt/conda/bin/activate
/opt/conda/bin/python baichuan2/base.py -m fastertransformer -iid ./tests/start_ids.csv -c
1 -b 2 -u 127.0.0.1:8001 -response_count 10
```

上面用到的命令行参数解释如下：

1. `-c` 指定发送请求的个数
2. `-b` 指定发送请求的batch数
3. `-u` 指定server的地址和端口
4. `-response_count` 指定要计算的输出token长度
5. `-m` 指定triton里面模型配置的名称
6. `-iid` 指定请求的token所在的文件

下面简单介绍一下 `/opt/example/client/baichuan2/base.py` 的基本逻辑。

python client的详细使用可以参考 `/opt/tritonclient/client/src/python/library/tritonclient/` 的注释和 `/opt/tritonclient/install/python/` 里面的示例程序。

1. python client library. client library安装包已经包含在发布的镜像里面，并且镜像已经安装了发布包，位置在 `/opt/tritonclient/install/python/tritonclient-r23.07-py3-none-manylinux1_x86_64.whl` 。

   如果要在主机安装，可以从container里面拷贝出来安装。 也可以使用下面的命令安装，注意，目前不支持cuda_shared_memory。

   ```
   pip install tritonclient[all]
   ```

2. 创建client。因为当前使用的是stream模式，所以只能使用grpcclient。

   ```
   import tritonclient.grpc as grpcclient
   from tritonclient.utils import np_to_triton_dtype

   client = grpcclient.InferenceServerClient(flags.url, flags.verbose, False)
   ```

3. 构建input的tensor数据。使用numpy.ndarray构建input数据。

   ```
   def generate_input(batch_size: int, start_id: int, end_id: int, input_ids_file_name:
   str, request_len: int,
                      bad_words_file_name: str = None, stop_words_file_name: str = None,
                      temperature: float = 1.0, len_penalty: float = 1.0, min_length: int
   = 0,
                      repetition_penalty: float = 1.0, topk: int = 1, topp: float = 0.0):
       inputs = []
       input_ids, input_ids_len, max_input_ids_len = ft_utils.read_start_ids(batch_size,
   end_id, 1, input_ids_file_name)
       inputs.append(prepare_tensor("input_ids", input_ids.astype(np.uint32)))
       inputs.append(prepare_tensor("input_lengths", input_ids_len.reshape((batch_size,
   1))))

       output_len = max_input_ids_len + request_len
   ```

```
    inputs.append(prepare_tensor("request_output_len", (np.ones((batch_size, 1)) *
request_len).astype(np.uint32)))

    if bad_words_file_name:
        bad_words = ft_utils.read_word_ids(bad_words_file_name)
        bad_words = np.tile(bad_words, (batch_size, 1)).astype(np.int32)
        inputs.append(prepare_tensor("bad_words_list", bad_words))

    if stop_words_file_name:
        stop_words = ft_utils.read_word_ids(stop_words_file_name)
        stop_words = np.tile(stop_words, (batch_size, 1)).astype(np.int32)
        inputs.append(prepare_tensor("stop_words_list", stop_words))

    temperatures = (np.ones((batch_size, 1)) * temperature).astype(np.float32)
    inputs.append(prepare_tensor("temperature", temperatures))

    len_penalties = (np.ones((batch_size, 1)) * len_penalty).astype(np.float32)
    inputs.append(prepare_tensor("len_penalty", len_penalties))

    min_length_np = (np_ones_batch_one(batch_size) * min_length).astype(np.int32)
    inputs.append(prepare_tensor("min_length", min_length_np))

    start_ids = (np_ones_batch_one(batch_size) * start_id).astype(np.uint32)
    inputs.append(prepare_tensor("start_id", start_ids))

    end_ids = (np_ones_batch_one(batch_size) * end_id).astype(np.uint32)
    inputs.append(prepare_tensor("end_id", end_ids))

    if repetition_penalty != 1.0:
        repetition_penalties = (np_ones_batch_one(batch_size) *
repetition_penalty).astype(np.float32)
        inputs.append(prepare_tensor("repetition_penalty", repetition_penalties))


    random_seed = (0 * np_ones_batch_one(batch_size)).astype(np.uint64)
    inputs.append(prepare_tensor("random_seed", random_seed))

    runtime_top_k = (topk * np_ones_batch_one(batch_size)).astype(np.uint32)
    inputs.append(prepare_tensor("runtime_top_k", runtime_top_k))

    runtime_top_p = (topp * np_ones_batch_one(batch_size)).astype(np.float32)
    inputs.append(prepare_tensor("runtime_top_p", runtime_top_p))

    return inputs, output_len
```

4. 启动stream并设置回调函数，当收到server的响应时，回调函数会被调用。

```
class UserData:
    def __init__(self):
        self._completed_requests = queue.Queue()

# Callback function used for async_stream_infer()
def completion_callback(user_data, result, error):
    # passing error raise and handling out
    user_data._completed_requests.put((result, error))


user_data = UserData()
client.start_stream(partial(completion_callback, user_data))
```

5. 使用`async_stream_infer`发送请求，并从`user_data._completed_requests`获取response。

```
client.async_stream_infer(flags.model, inputs, request_id=get_id(my_id, str(i)))

while True:
    result, error = user_data._completed_requests.get(True, 30 * 20)
    if error is not None:
        print("got error. {}".format(error))
        break
    output0 = result.as_numpy("output_ids")
    output1 = result.as_numpy("sequence_length")
    meta = result.get_response(True)
    results.set_count(meta["id"], output1, output0)
    if results.is_end():
        print("id: ", meta["id"])
        print(output0)
        print(output1)
        break
```

6. 不再使用的时候，使用`stop_stream`关闭stream。

```
client.stop_stream()
```

8 基于 Triton 运行 dlnne backend

Triton server通过`dlnne backend`支持了登临SDK中dlNNE模块的推理功能，`dlnne backend`支持所有dlNNE支持的模型。如果需要使用`dlnne backend`进行推理，需要准备`dlnne backend`相关的Triton的模型配置文件。

8.1 准备配置文件

以`bge_large_zh_v1_5`为例，`Triton server`加载`bge_large_zh_v1_5`模型前，需要准备相应的配置。`bge_large_zh_v1_5`模型的`config.pbtxt`示例如下：

```
backend: "dlnne"
name: "bge_large_zh_v1_5"
default_model_filename: "bge_large_zh_v1_5.onnx"
max_batch_size: 32

instance_group [
    {
      count: 1
      kind: KIND_GPU
    }
]

input [
  {
    name: "input_ids"
    data_type: TYPE_INT64
    dims: [ 512 ]
    reshape { shape: [ 1,512 ] }
  },
  {
    name: "attention_mask"
    data_type: TYPE_INT64
    dims: [ 512 ]
    reshape { shape: [ 1,512 ] }
  }
]

output [
  {
    name: "last_hidden_state"
    data_type: TYPE_FP32
    dims: [ 512,1024 ]
    reshape { shape: [ 1,512,1024 ] }
  }
]

## denglin model config (对于bge_large_zh_v1_5模型，不用配置，因此为空)
# 配置 user_op plugin等（如yolov3）.
parameters {
  key: "user_op.library"
```

```
  value: {
    string_value: ""
  }
}

parameters {
  key: "user_op.module"
  value: {
    string_value: ""
  }
}

parameters {
  key: "user_op.first_op_name"
  value: {
    string_value: ""
  }
}

parameters {
  key: "plugin_so_paths"
  value: {
    string_value: ""
  }
}

parameters {
  key: "plugin_so_env"
  value: {
    string_value: ""
  }
}
```

配置文件的配置项说明如下：

- `backend`：字符串，指定backend名称，这里固定使用 `dlnne` ，表示使用dlNNE推理引擎。
- `name`：字符串，指定模型名称。
- `default_model_filename`：字符串，指定要加载模型的名称。
- `max_batch_size`：整数，指定request的最大batch大小。
- `instance_group`：`instance_group`对象数组，用于为可用的每个 GPU 创建单个或多个模型执行实例，并指定GPU。

```
# 默认情况下，triton会为系统中可用的每个 GPU 创建单个模型执行实例。可以使用实例组设置将模型的多个执行实例放置在每个 GPU 上或仅放置在某些 GPU 上。
# 例如，以下配置将在每个系统 GPU 上放置1个模型执行实例。
instance_group [
    {
      count: 1
      kind: KIND_GPU
```

```
    }
]

# 以下配置将在 GPU 0 上放置一个执行实例，在 GPU 1 和 2 上放置两个执行实例。
instance_group [
    {
      count: 1
      kind: KIND_GPU
      gpus: [ 0 ]
    },
    {
      count: 2
      kind: KIND_GPU
      gpus: [ 1, 2 ]
   }
]
```

- `input`：input对象的数组。指定该模型的输入。单个input对象描述单个输入的信息，有如下配置项：
  - `name`：字符串，是该input的名称；
  - `data_type`：数据类型；
  - `dims`：整数数组，表示该input的维度；
  - `reshape`：`dims`对象数组，如果 Triton 在推理请求中收到的输入形状与模型预期的输入形状不匹配，则必须使用 reshape 属性。
- `outputs`：output对象的数组。指定该模型的输出。单个output对象描述单个输出的信息，有如下配置项：
  - `name`：字符串，是该output的名称；
  - `data_type`：数据类型；
  - `dims`：整数数组，表示该output的维度；
  - `reshape`：`dims`对象数组，如果模型生成的输出形状与 Triton 在响应推理请求时返回的形状不匹配，则必须使用 reshape 属性。
- `parameter`：键值对。配置登临`user_op`和`Plugin`等信息。用于配置登临SDK提供的yolov3等需要特殊配置的模型。

```
## Configure user_op and plugin for models like YOLOv3.
parameters {
  key: "user_op.library"
  value: {
    # 配置注册so
    string_value:
"/denglinsdk/sdk/samples/plugin/yolov3_opt/dlnne_plugin_build/libyolov3_opt_tvm.so"
  }
}

parameters {
  key: "user_op.module"
  value: {
    # 配置转换脚本
    string_value: "/denglinsdk/sdk/samples/plugin/yolov3_opt/front_end.py"
  }
}

parameters {
```

```
  key: "user_op.first_op_name"
  value: {
    # 配置op name
    string_value: "custom_op"
  }
}

## Separate the paths of the shared object (SO) files using commas.
parameters {
  key: "plugin_so_paths"
  value: {
    string_value:
    # 配置yolov3,yolov5的相关opt plugin的so

"/denglinsdk/sdk/samples/plugin/yolov3_opt/dlnne_plugin_build/libyolov3_opt_plugin.so,
libyolov5_opt_plugin.so"
  }
}

parameters {
  key: "plugin_so_env"
  value: {
    #  配置yolov3 plugin的kernel路径
    string_value:
"YOLOV3_PLUGIN_KERNEL_PATH=/denglinsdk/sdk/samples/plugin/yolov3_opt/dlnne_plugin/plug
in/kernel"
  }
}
```

8.2 启动 Triton Server

1. 通过`docker`命令进入Triton server执行的容器。

```
# <sdk_path>表示host端需要挂载到容器内的登临SDK所在路径
# <tag>表示需要启动的repository的名为dl-triton的标签（TAG）
docker run -it --rm --shm-size=1g --privileged --network host \
       -v <path of bge
model>/bge_large_zh_v1_5:/opt/examples/modules/bge_large_zh_v1_5 \
       dl-triton:<tag> /bin/bash
```

2. 拷贝`bge_large_zh_v1_5`的相应配置文件到容器，拷贝时确保存放`bge_large_zh_v1_5.onnx`文件的文件夹已经映射到容器内，可以通过docker命令`-v`指定 。

```
# 以bge_large_zh_v1_5模型为例，拷贝模型目录bge_large_zh_v1_5下的所有文件和文件夹到单独的目录/opt/dlnne_triton_configs/
mkdir -p /opt/dlnne_triton_configs
cp -r <path of bge model>/bge_large_zh_v1_5 /opt/dlnne_triton_configs/

# tree /opt/dlnne_triton_configs 可以查看到目录结构，其中的config.pbtxt即为8.1章节所示配置
bge_large_zh_v1_5/
|-- 1
|   `-- bge_large_zh_v1_5.onnx
`-- config.pbtxt
```

3. 启动Triton server。

```
# 激活登临sdk环境相关环境变量
source /denglinsdk/sdk/env.sh
source /dl/python/bin/activate

# --backend-directory 参数指定需要加载的backend类型，./tritonserver/backends/包含dlnne backend的相关so库。
CUDA_VISIBLE_DEVICES=0 ./tritonserver/bin/tritonserver  --model-repository dlnne_triton_configs/ --backend-directory ./tritonserver/backends/
```

Triton server启动成功后，会有如下输出：

```
I0604 09:29:00.967721 778 server.cc:674]
+------------------------+---------+--------+
| Model                  | Version | Status |
+------------------------+---------+--------+
| bge_large_zh_v1_5      | 1       | READY  |
...
I0604 09:29:01.518221 778 grpc_server.cc:2451] Started GRPCInferenceService at 0.0.0.0:8001
I0604 09:29:01.518424 778 http_server.cc:3558] Started HTTPService at 0.0.0.0:8000
I0604 09:29:01.560307 778 http_server.cc:187] Started Metrics Service at 0.0.0.0:8002
```

9 第三方开源软件 License

登临Triton发布包使用了NVIDIA的Triton Server服务，登临根据其开源软件许可证向您提供服务。该许可证同时在产品中包含了特定的法律信息。本章节提供了相关内容以供您获取此类信息。

Denglin Triton product contains NVIDIA Triton Server service, which is being made available to you under its respective license, it is open source software license, it is also require specific legal information to be included in the product. This chapter provides links and license for you to obtain such information.

Triton License：https://github.com/triton-inference-server/server/blob/main/LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of NVIDIA CORPORATION nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.