Multi_Device_Plugin_and_the_DevCloud

July 1, 2020

1 Walkthrough: Multi Device Plugin and the DevCloud

This notebook is a demonstration showing you how to request an edge node with an Intel i5 CPU and load a model on the CPU, GPU, and VPU (Intelő Neural Compute Stick 2) at the same time using the Multi Device Plugin on Udacity's workspace integration with Intel's DevCloud. This notebook is just to give you an overview of the process (you won't be writing any code). In the next workspace, you'll be given TODO items to complete.

Below are the six steps we'll walk through in this notebook:

1. Creating a Python script to load the model
2. Creating a job submission script
3. Submitting a job using the `qsub` command
4. Checking the job status using the `liveQStat` function
5. Retrieving the output files using the `getResults` function
6. Viewing the resulting output

Click the **Introduction to Multi Device Plugin and the DevCloud** button below for a quick overview of the overall process. We'll then walk through each step of the process.

Introduction to Multi Device Plugin and the DevCloud

IMPORTANT: Set up paths so we can run Dev Cloud utilities You *must* run this every time you enter a Workspace session.

```
In [1]: %env PATH=/opt/conda/bin:/opt/spark-2.4.3-bin-hadoop2.7/bin:/opt/conda/bin:/usr/local/sb
        import os
        import sys
        sys.path.insert(0, os.path.abspath('/opt/intel_devcloud_support'))
        sys.path.insert(0, os.path.abspath('/opt/intel'))
```

```
env: PATH=/opt/conda/bin:/opt/spark-2.4.3-bin-hadoop2.7/bin:/opt/conda/bin:/usr/local/sbin:/usr/
```

1.1 The Model

We will be using the `vehicle-license-plate-detection-barrier-0106` model for this exercise.

Remember to use the appropriate model precisions for each device:

- IGPU - `FP16`

- VPU - `FP16`
- CPU - It is prefered to use `FP32`, but we have to use `FP16` since **GPU** and **VPU** use `FP16`

The model has already been downloaded for you in the `/data/models/intel` directory on Intel's DevCloud.

We will be running inference on an image of a car. The path to the image is `/data/resources/car.png`.

2 Step 1: Creating a Python Script

The first step is to create a Python script that you can use to load the model and perform an inference. I have used the `%%writefile` magic command to create a Python file called `load_model_to_device.py`. This will create a new Python file in the working directory.

Note: The advantage of using the **Multi device plugin** is that it does not require us to change our application code. So we will be using the same Python script we used in the previous VPU walkthrough.

Click the **Writing a Python Script** button below for a demonstration.

Writing a Python Script

```
In [2]: %%writefile load_model_to_device.py

        import time
        from openvino.inference_engine import IENetwork
        from openvino.inference_engine import IEPlugin
        import argparse

        def main(args):
            model=args.model_path
            model_weights=model+'.bin'
            model_structure=model+'.xml'

            start=time.time()
            model=IENetwork(model_structure, model_weights)

            plugin = IEPlugin(device=args.device)

            net = plugin.load(network=model, num_requests=1)
            print(f"Time taken to load model = {time.time()-start} seconds")

        if __name__=='__main__':
            parser=argparse.ArgumentParser()
            parser.add_argument('--model_path', required=True)
            parser.add_argument('--device', default=None)

            args=parser.parse_args()
            main(args)
```

```
Writing load_model_to_device.py
```

2.1 Step 2: Creating a Job Submission Script

To submit a job to the DevCloud, we need to create a shell script. Similar to the Python script above, I have used the `%%writefile` magic command to create a shell script called `load_multi_model_job.sh`.

This script does a few things. 1. Writes stdout and stderr to their respective .log files 2. Creates the `/output` directory 3. Creates `DEVICE` and `MODELPATH` variables and assigns their value as the first and second argument passed to the shell script 4. Calls the Python script using the `MODELPATH` and `DEVICE` variable values as the command line argument 5. Changes to the `/output` directory 6. Compresses the stdout.log and stderr.log files to `output.tgz`

Note: Just like our Python script, our job submission script also does not need to change when using the **Multi device plugin**. Step 3, where we submit our job to the DevCloud, is where we have to make a minor change.

Click the **Creating a Job Submission Script** button below for a demonstration.

Creating a Job Submission Script

In [3]:
```
%%writefile load_multi_model_job.sh

exec 1>/output/stdout.log 2>/output/stderr.log

mkdir -p /output

DEVICE=$1
MODELPATH=$2

# Run the load model python script
python3 load_model_to_device.py  --model_path ${MODELPATH} --device ${DEVICE}

cd /output

tar zcvf output.tgz stdout.log stderr.log
```

```
Writing load_multi_model_job.sh
```

2.2 Step 3: Submitting a Job to Intel's DevCloud

The code below will submit a job to an **IEI Tank-870** edge node with the following three devices: * **Intel Core i5 6500TE** * **Intel HD Graphics 530** * **Intel Neural Compute Stick 2**

Note: We'll pass in a device type argument of `MULTI:MYRIAD,GPU,CPU` to load our model on all three devices at the same time. We'll need to use `FP16` as the model precision since we're loading our model on a GPU and VPU even though the recommended model precison is `FP32` for CPU.

The `!qsub` command takes a few command line arguments: 1. The first argument is the shell script filename - `load_multi_model_job.sh`. This should always be the first argument. 2. The `-d` flag designates the directory where we want to run our job. We'll be running it in the current directory as denoted by `.`. 3. The `-l` flag designates the node and quantity we want to request. The default quantity is 1, so the **1** after `nodes` is optional.

4. The `-F` flag let's us pass in a string with all command line arguments we want to pass to our Python script.

Note: There is an optional flag, `-N`, you may see in a few exercises. This is an argument that only works on Intel's DevCloud that allows you to name your job submission. This argument doesn't work in Udacity's workspace integration with Intel's DevCloud.

In the cell below, we assign the returned value of the `!qsub` command to a variable `job_id_core`. This value is an array with a single string.

Once the cell is run, this queues up a job on Intel's DevCloud and prints out the first value of this array below the cell, which is the job id.

Click the **Submitting a Job to Intel's DevCloud** button below for a demonstration.

Submitting a Job to Intel's DevCloud

```
In [5]: job_id_core = !qsub load_multi_model_job.sh -d . -l nodes=1:tank-870:i5-6500te:intel-hd-
        print(job_id_core[0])
```

```
LmBg39PJEhLZebgV5XOxals5MFoAiZ5Y
```

2.3 Step 4: Running liveQStat

Running the `liveQStat` function, we can see the live status of our job. Running the this function will lock the cell and poll the job status 10 times. The cell is locked until this finishes polling 10 times or you can interrupt the kernel to stop it by pressing the stop button at the top:

- `Q` status means our job is currently awaiting an available node
- `R` status means our job is currently running on the requested node

Note: In the demonstration, it is pointed out that `W` status means your job is done. This is no longer accurate. Once a job has finished running, it will no longer show in the list when running the `liveQStat` function.

Click the **Running liveQStat** button below for a demonstration.

Running LiveQStat

```
In [6]: import liveQStat
        liveQStat.liveQStat()
```

2.4 Step 5: Retrieving Output Files

In this step, we'll be using the `getResults` function to retrieve our job's results. This function takes a few arguments.

1. `job id` - This value is stored in the `job_id_core` variable we created during **Step 3**. Remember that this value is an array with a single string, so we access the string value using `job_id_core[0]`.
2. `filename` - This value should match the filename of the compressed file we have in our `load_multi_model_job.sh` shell script. In this example, filename shoud be set to `output.tgz`.
3. `blocking` - This is an optional argument and is set to `False` by default. If this is set to `True`, the cell is locked while waiting for the results to come back. There is a status indicator showing the cell is waiting on results.

Note: The `getResults` function is unique to Udacity's workspace integration with Intel's DevCloud. When working on Intel's DevCloud environment, your job's results are automatically retrieved and placed in your working directory.

Click the **Retrieving Output Files** button below for a demonstration.

Retrieving Output Files

```
In [7]: import get_results

        get_results.getResults(job_id_core[0], filename="output.tgz", blocking=True)
```

```
getResults() is blocking until results of the job (id:LmBg39PJEhLZebgV5XOxals5MFoAiZ5Y) are read
Please wait...Success!
output.tgz was downloaded in the same folder as this notebook.
```

2.5 Step 6: Viewing the Outputs

In this step, we unpack the compressed file using `!tar zxf` and read the contents of the log files by using the `!cat` command.

`stdout.log` should contain the printout of the print statement in our Python script.

```
In [8]: !tar zxf output.tgz
```

```
In [9]: !cat stdout.log
```

```
Time taken to load model = 43.13548684120178 seconds
```

```
In [10]: !cat stderr.log
```

```
load_model_to_device.py:13: DeprecationWarning: Reading network using constructor is deprecated.
  model=IENetwork(model_structure, model_weights)
load_model_to_device.py:15: DeprecationWarning: IEPlugin class is deprecated. Please use IECore
  plugin = IEPlugin(device=args.device)
tar: stdout.log: file changed as we read it
```

```
In [ ]:
```