NVIDIA Tensor Cores for Accelerating Machine Learning Workload

Most of the operations in machine learning workload are large dense matrix (tensor) operations, both for training models and make prediction using the trained model (i.e., inference). Several companies have been developing specialized hardware to boost performance in tensor operations which can accelerate overall machine learning workload. Graphics Processing Units (GPUs) becomes more popular to satisfy the need of huge compute power for machine learning workload and are responsible for the booming of machine learning today. During the 2017 GPU Technology Conference, NVIDIA announced Volta GPU microarchitecture as a successor for Pascal. The notable new feature is Tensor Cores, a specialized compute unit inside the GPU that perform 4x4 matrix multiplication in FP16 and accumulate the result in FP16 or FP32. The Tensor Core can dramatically improve matrix multiplication performance for up-to 8 times of standard FP32 matrix multiplication and can theoretically deliver performance up-to 125 TFLOP/s.

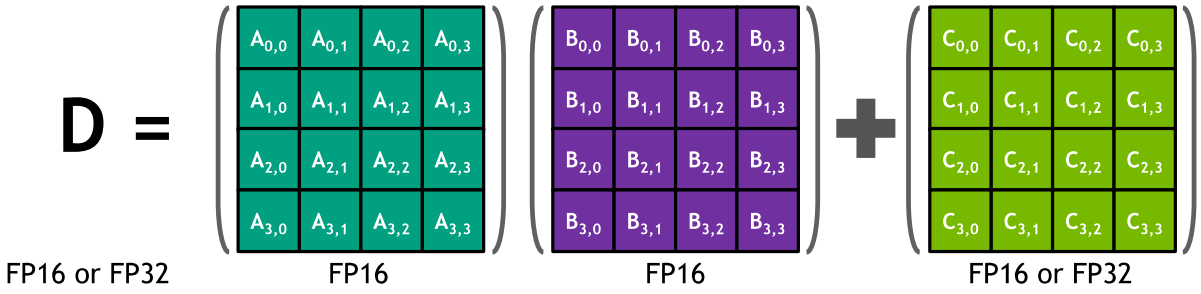
In this assignment, we will try to use the Tensor Core to accelerate our machine learning workload. Topics that will be covered in this assignment are listed below.

- Overview of the Tensor Cores.
- Train a neural network model using Pytorch to do image classification.
- Make predictions using the trained model (inference) in Pytorch.
- Modify the training and inference using FP16 to take advantage of the Tensor Cores.
- Perform profiling using NVPROF to observe the usage of Tensor Cores.

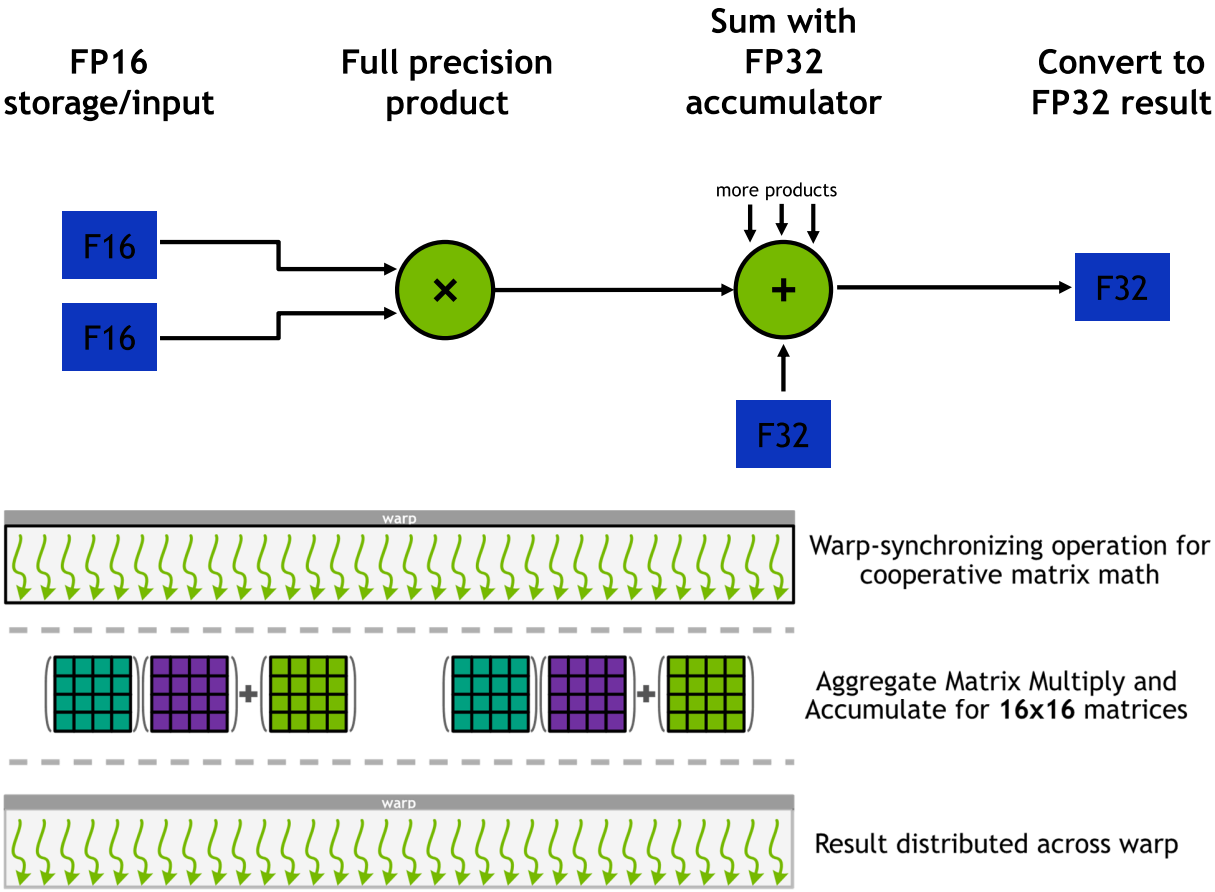
A. Overview of The Tensor Cores

Tensor Cores are programmable matrix-multiply-and-accumulate units available in NVIDIA Volta and Turing GPUs. In Volta, each tensor core provides a 4x4x4 matrix processing array which performs the following operation.

$$D = A * B + C$$



All of A, B, C, and D are 4x4 matrices. The matrix multiply inputs A and B are FP16 matrices while the accumulation matrices C and D can be FP16 or FP32 matrices. Each Tensor Cores perform 64 floating point FMA mixed-precision operations per clock. Each Streaming Multiprocessor (SM) has 8 Tensor Cores that can perform a total of 1024 floating point operations per clock which is 8x increase in throughput for deep learning applications per SM compared to Pascal using standard FP32 operations.



Multiple Tensor Cores are used concurrently by a full warp of execution where the threads within a warp provide a larger 16x16x16 matrix operation to be processed by the Tensor Cores. CUDA exposes these operations as warp-level matrix operations in the CUDA C++ WMMA API. Fortunately, in this assignment, we will not program the low-level function to utilize the Tensor Cores. We will use PyTorch which is a deep learning framework that is already able to utilize the Tensor Core if we use it correctly. If you want to look at how to program the low-level function using CUDA C++ WMMA API, please refer to [1].

GPU	# SM	# Tensor Cores	Peak FP16	Peak INT8	Peak INT4	Peak INT1
Tesla V100 (Volta)	80	640 (8 per SM)	125 TFlop/s	-	-	-
Titan RTX (Turing)	72	576 (8 per SM)	130 TFlop/s	261 Tops	522 Tops	2088 Tops

In Turing, the Tensor Cores are updated. As an addition to FP16, the Tensor Cores in Turing can perform INT8 and INT4 multiplication and accumulate them into INT32 result for inference workloads that can tolerate quantization and don't require FP16 precision. The Turing Tensor Cores can provide 2048 INT8 operations per clock and 4096 INT4 operations per clock. There is also INT1 computation which is still experimental.

Here is the list of NVIDIA GPU that has Tensor Cores and may be used for working on this assignment. Please take a note that we never run this assignment using Turing and there may be unknown problem when running this assignment.

Arch	Model	# SM	# Tensor Cores	Memory
Volta				
	Titan V	80	640	12GB HBM2 @ 652.8 GB/s
	Quadro GV100	80	640	32GB HBM2 @ 870.0 GB/s
	Tesla V100	80	640	32GB HBM2 @ 900 GB/s
Turing				
	GeForce RTX 2060	30	240	6GB GDDR6 @ 336.0 GB/s
	GeForce RTX 2060 Super	34	272	8GB GDDR6 @ 448.0 GB/s
	GeForce RTX 2070	36	288	8GB GDDR6 @ 448.0 GB/s
	GeForce RTX 2070 Super	40	320	8GB GDDR6 @ 448.0 GB/s
	GeForce RTX 2080	46	368	8GB GDDR6 @ 448.0 GB/s
	GeForce RTX 2080 Super	48	384	8GB GDDR6 @ 496.0 GB/s
	GeForce RTX 2080 Ti	68	544	11GB GDDR6 @ 616.0 GB/s
	Titan RTX	72	576	24GB GDDR6 @ 672.0 GB/s
	Quadro RTX 4000	36	288	8GB GDDR6 @ 416 GB/s
	Quadro RTX 5000	48	384	16GB GDDR6 @ 448 GB/s
	Quadro RTX 6000	72	576	24GB GDDR6 @ 576 GB/s
	Quadro RTX 8000	72	576	48GB GDDR6 @ 672GB/s
	Tesla T4	40	320	16GB GDDR6 @ 320GB/s

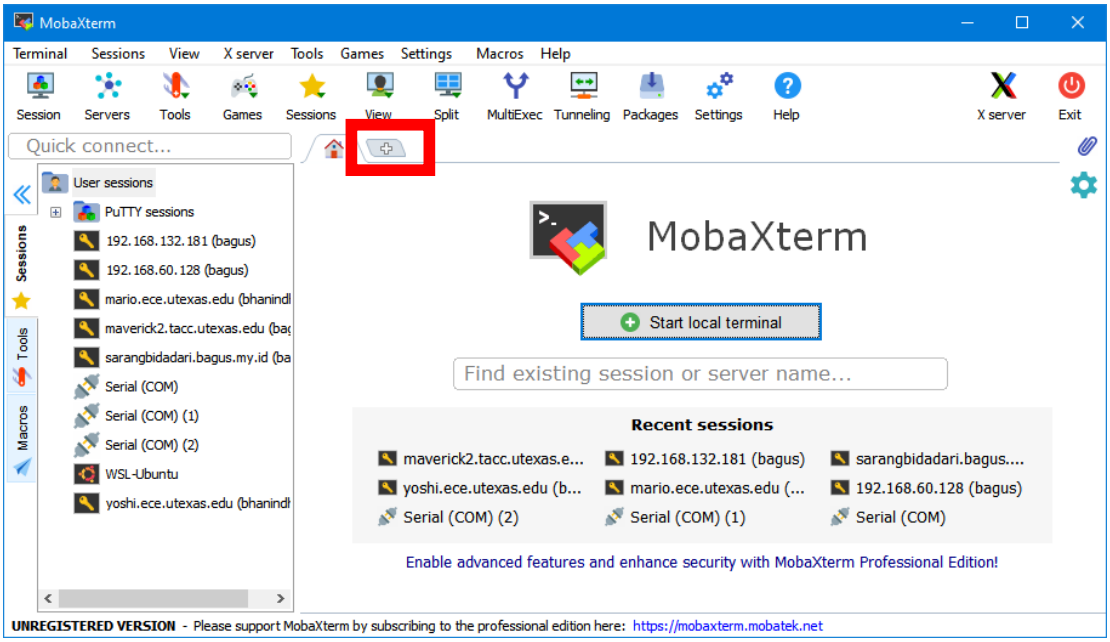
B. Preparing The Environment

In this section, we will prepare the environment for this assignment. We will use Maverick2 on TACC although you are free to use your own machine as long as you have NVIDIA GPU with Tensor Cores. If you are the first time on using Maverick2 on TACC, you can read the user guide on [2]. There are three types of compute nodes available in Maverick2: gtx, p100, and v100. For this assignment, we will use the v100 nodes since they are the only node equipped with two NVIDIA Tesla V100 GPUs which have Tensor Cores. Since TACC only has 4 of those nodes, we encourage you to start the assignment earlier and use the allocation as efficient as possible so the others can work on the assignment too (e.g., kill the job after you have finished working on your assignment).

1. Login into Maverick2 Login Node

   For Windows user, we recommend to install MobaXterm as your SSH client although you are free to use your favourite SSH client (e.g., PuTTY). You can download MobaXterm from [3]. After you have installed MobaXterm, you can login to the Maverick2 Login Node as follows.

a. Start the MobaXterm and press + icon on the terminal tab bar. We refer this as the **first** terminal.



b. Type the following command to login into the Maverick2 Login Node. Replace `TACC_USERNAME` with your TACC Username.

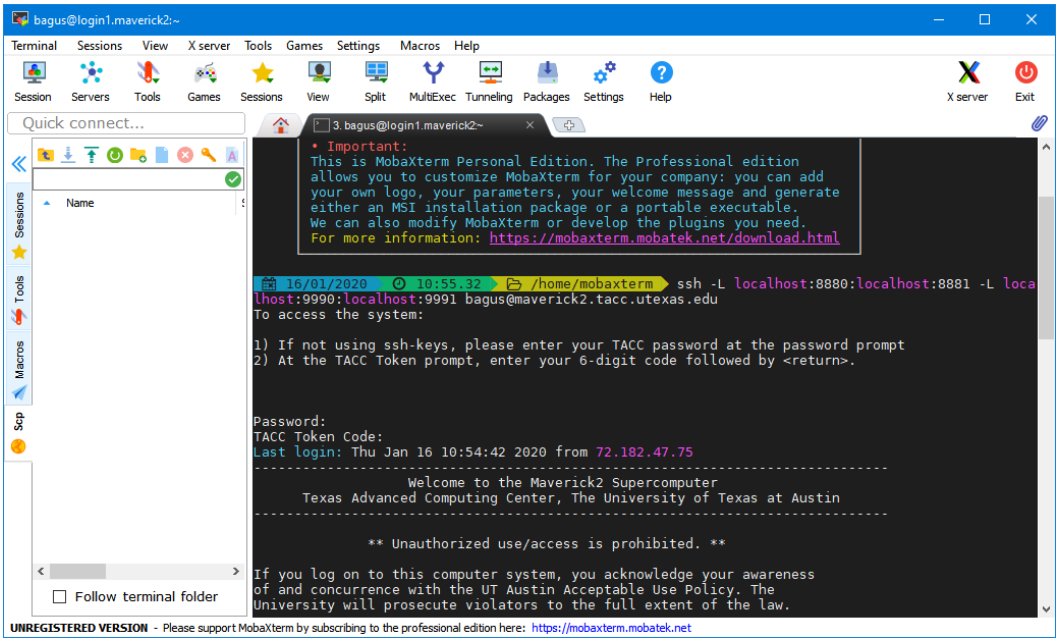
```
ssh -q -L localhost:8880:localhost:8881 -L localhost:9990:localhost:9991
TACC_USERNAME@maverick2.tacc.utexas.edu
```

This SSH login will forward port 8881 and 9991 on Maverick2 Login Node to port 8880 and 9990 on your computer. The port forward is needed to run Jupyter Notebook (JN) and TensorBoard (TB) for our assignment. If you have a port conflict, you are free to change the port number. The diagram of the port forward is shown below.

	Your Computer		TACC Maverick2 Login Server		TACC Maverick2 V100 Compute Node
JN	8880	←	8881	←	8882
TB	9990	←	9991	←	9992

We will run Jupyter Notebook (JN) and TensorBoard (TB) on Compute Node. The Compute Node is only accessible from the Login Node once we have the allocation, and hence two hops of port forward.

c. Login is successful after you see the following welcome message.



2. Installing Anaconda

Anaconda is free and open source platform for Python data science and is popular for use in machine learning. We recommend to install Conda on your `$WORK` directory since your `$HOME` only has `10GB` capacity while your `$WORK` has `1024GB` capacity.

a. Go to your `$WORK` directory. Please take a note of your `$WORK` path which is in the form of `/work/xxxxx/TACC_USERNAME`.

```
cd $WORK
cd ..
pwd
```

b. Download the installation package of Anaconda. If you want to download newest version for Linux distribution, you can replace the download link by looking at the download page of Anaconda [4].

```
wget https://repo.anaconda.com/archive/Anaconda3-2019.10-Linux-
x86_64.sh -O anaconda3.sh
```

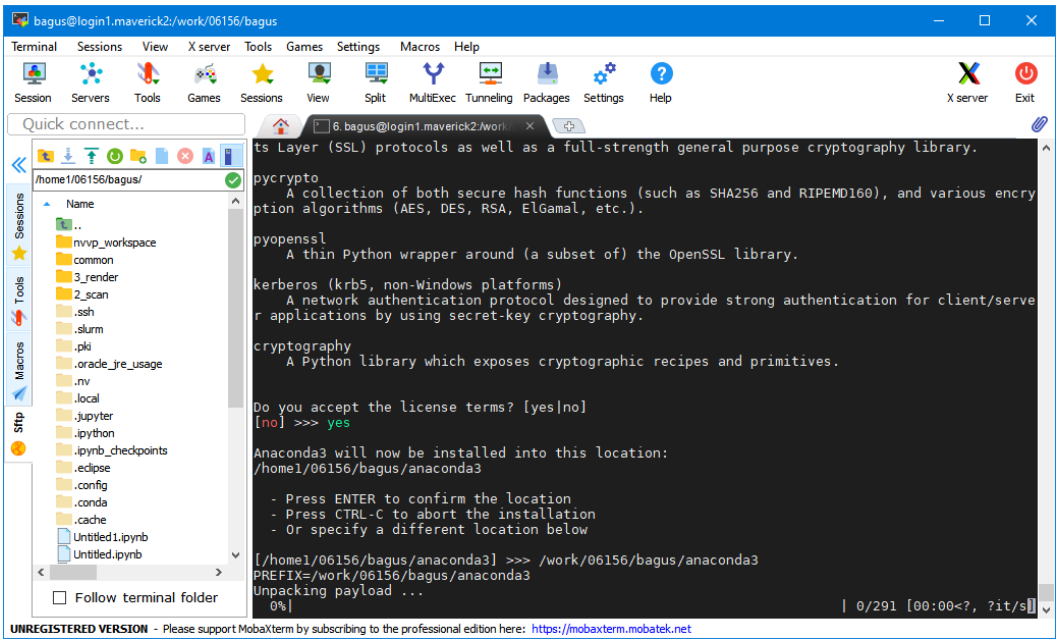
c. Make the installation package executable.

```
chmod +x anaconda3.sh
```

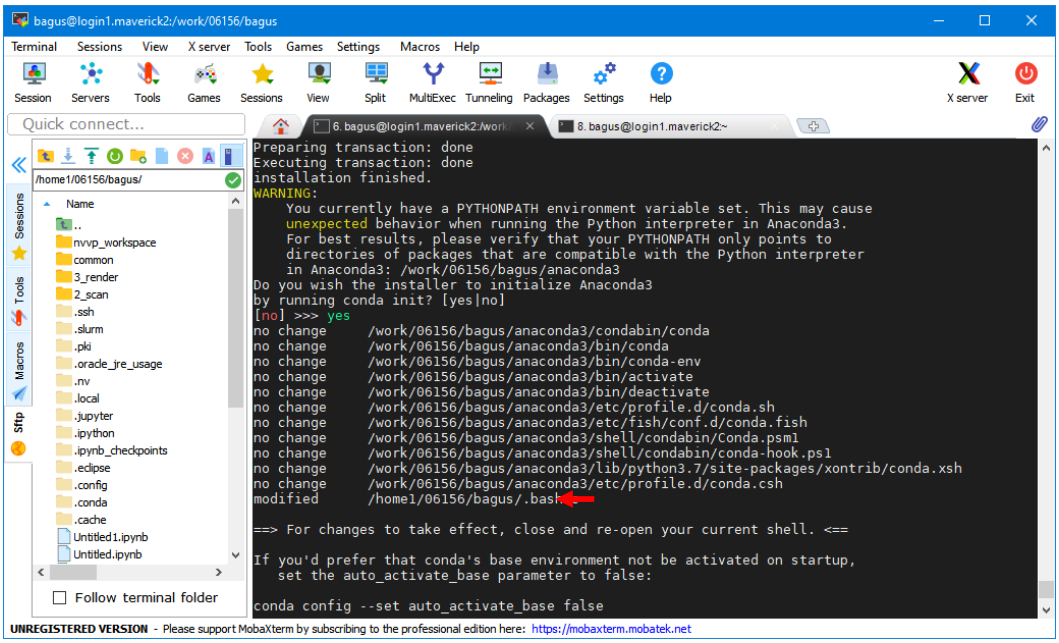
d. Run the installation package and read the end user license agreement. Type `yes` if you accept the agreement.

```
./anaconda3.sh
```

When the installation ask for the location, put your `$WORK` path to install the Anaconda in your `/work/xxxxx/TACC_USERNAME/anaconda3` directory. It will take some time to extract the installation package.



e. After the installation finished, it will ask to run `conda init`. Answer `yes` and you should see that your `.bashrc` file will be modified.



f. Source your .bashrc file. If you see `(base)` in front of your terminal prompt (e.g., `(base) login1.maverick2(1034)$`), you have successfully installed Anaconda.

```
source $HOME/.bashrc
```

3. Download Assignment File

Go to your $WORK directory and execute these following commands. You will clone a github repository that contains files for this assignment.

```
cd $WORK
git clone https://github.com/hibagus/Tensor-Cores-Image-Classification
cd Tensor-Cores-Image-Classification/
```

4. Creating Anaconda Environment

Anaconda Environment is an isolated environment where you can install python packages in specific version. You can have multiple Anaconda Environments that have different version of python packages installed. It is a good practice to create a new Anaconda Environment for our assignment. Let say our environment name is CatsandDogsClassification (you can change the name as you wish) and assuming that you are already in $WORK/Tensor-Cores-Image-Classification, create the new environment as follows.

```
conda env create --name CatsandDogsClassification --file conda_env_spec_file.yml
```

5. Get Compute Node Allocation

You need to ask allocation for a V100 compute node to work on this assignment. Each V100 compute node has two NVIDIA Tesla V100 GPUs. We only need to use one of them for this assignment. To get allocation for a V100 compute node, run this following command. This command will run a dummy job on the compute node for a maximum time of one day.

```
sbatch runjob.sh
```

After you submit the job, you can see the status of the job by running this command. Please replace TACC_USERNAME with your TACC Username.

```
squeue -u TACC_USERNAME
```

This command will give you the following outputs.

```
(base) login1.maverick2(1059)$ squeue -u bagus
             JOBID   PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             66493        v100 Cats&Dog    bagus  R       2:20      1 c228-103
```

Please take look at the ST column which will indicate the state of your job. R means that the job is currently running and you have been allocated with the compute node you want. If the state of your job is other than R (e.g., PD), you have to wait until your job is running. If your job is currently running, please take a note of the JOBID and NODELIST.

IMPORTANT. There are only four V100 compute nodes available on TACC Maverick2 and you are sharing with other users too. If you are no longer using your compute node, please cancel the job by executing this command. Replace JOBID with the JOBID you have obtained from previous command. You should execute this command from Maverick2 Login Node. If you already in Maverick2 Compute Node, you can type exit and you will be brought back to the Login Node.

```
scancel JOBID
```

6. Login to Compute Node

After you are allocated with compute node (your job ST is R), you can login into the Compute Node where you can work on your assignment. Replace TACC_USERNAME using your TACC Username and NODELIST using the node name obtained in previous step (e.g., c228-103).

```
ssh -q -L localhost:8881:localhost:8882 -L localhost:9991:localhost:9992
TACC_USERNAME@NODELIST.maverick2.tacc.utexas.edu
```

This SSH Login will forward port 8882 and 9992 in Maverick2 Compute Node into port 8881 and 9991 in Maverick2 Login Node, respectively. If you have successfully login into the Compute Node, your terminal will show `(base) NODELIST.maverick2(xxx)$` instead of `(base) login1.maverick2(xxx) $`.

7. Activate Anaconda Environment

Activate the Anaconda Environment that you have created. Please always check that you are in the correct environment before doing anything in this assignment.

```
conda activate CatsandDogsClassification
```

After activating the environment, your terminal will show `(CatsandDogsClassification) NODELIST.maverick2(xxx)$`.

8. Run Jupyter Notebook Server

Open **second** terminal in MobaXterm by pressing + icon on the terminal bar. Login into the Maverick2 Login Node as follows (you do not need to do port forward in this login). Replace TACC_USERNAME with your TACC Username.

```
ssh TACC_USERNAME@maverick2.tacc.utexas.edu
```

Then, login into Maverick2 Compute Node as follows (you do not need to do port forward in this login). Replace TACC_USERNAME with your TACC Username and NODELIST with the name of the compute node that you have been allocated.

```
ssh TACC_USERNAME@NODELIST.maverick2.tacc.utexas.edu
```

Activate the Anaconda Environment that you have created.

```
conda activate CatsandDogsClassification
```

Go to your work directory.

```
cd $WORK/Tensor-Cores-Image-Classification/
```

Run the Jupyter Notebook server using this following command. Adjust the port number as needed if you have changed how you forward the port during SSH login.

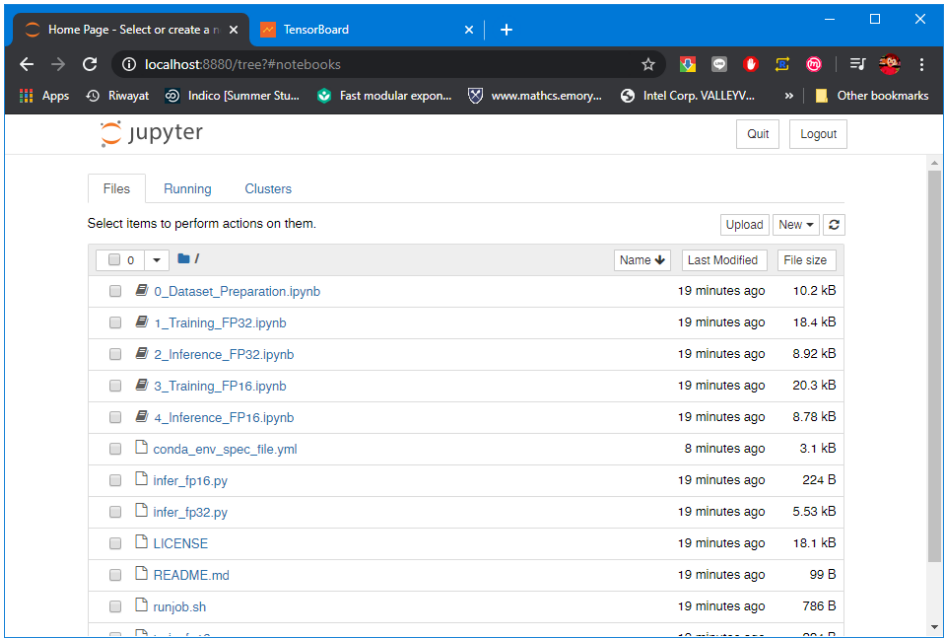
```
jupyter notebook --no-browser --port=8882
```

If the Jupyter Notebook Server is running, it will give outputs as follows. Please take a note of the token (in the example below, the token is a45bfac02a555738f2cf10858ec958546ca2e8d48d723f5b).

```
(CatsandDogsClassification) c228-103.maverick2(997)$ jupyter notebook --no-browser --
port=8882
[I 14:47:06.683 NotebookApp] Serving notebooks from local directory:
/work/06156/bagus/maverick2/Tensor-Cores-Image-Classification
[I 14:47:06.683 NotebookApp] The Jupyter Notebook is running at:
[I 14:47:06.683 NotebookApp]
http://localhost:8882/?token=a45bfac02a555738f2cf10858ec958546ca2e8d48d723f5b
[I 14:47:06.683 NotebookApp]  or
http://127.0.0.1:8882/?token=a45bfac02a555738f2cf10858ec958546ca2e8d48d723f5b
[I 14:47:06.683 NotebookApp] Use Control-C to stop this server and shut down all kernels
(twice to skip confirmation).
[C 14:47:06.691 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home1/06156/bagus/.local/share/jupyter/runtime/nbserver-237575-open.html
    Or copy and paste one of these URLs:
        http://localhost:8882/?token=a45bfac02a555738f2cf10858ec958546ca2e8d48d723f5b
     or http://127.0.0.1:8882/?token=a45bfac02a555738f2cf10858ec958546ca2e8d48d723f5b
```

You should be able to access the Jupyter Notebook on your computer by accessing http://localhost:8880 on your favourite browser. Please adjust the port number if you have changed them during the SSH login. Insert the token if requested.



9. Run TensorBoard

TensorBoard is a dashboard to display the statistics of training model in machine learning. TensorBoard is usually used with TensorFlow. Because we will use PyTorch in this assignment, we need to install TensorBoardX as an addition to TensorBoard.

Open **third** terminal in MobaXterm by pressing + icon on the terminal bar. Login into the Maverick2 Login Node as follows (you do not need to do port forward in this login). Replace `TACC_USERNAME` with your TACC Username.

```
ssh TACC_USERNAME@maverick2.tacc.utexas.edu
```

Then, login into Maverick2 Compute Node as follows (you do not need to do port forward in this login). Replace `TACC_USERNAME` with your TACC Username and `NODELIST` with the name of the compute node that you have been allocated.

```
ssh TACC_USERNAME@NODELIST.maverick2.tacc.utexas.edu
```

Activate the Anaconda Environment that you have created.

```
conda activate CatsandDogsClassification
```

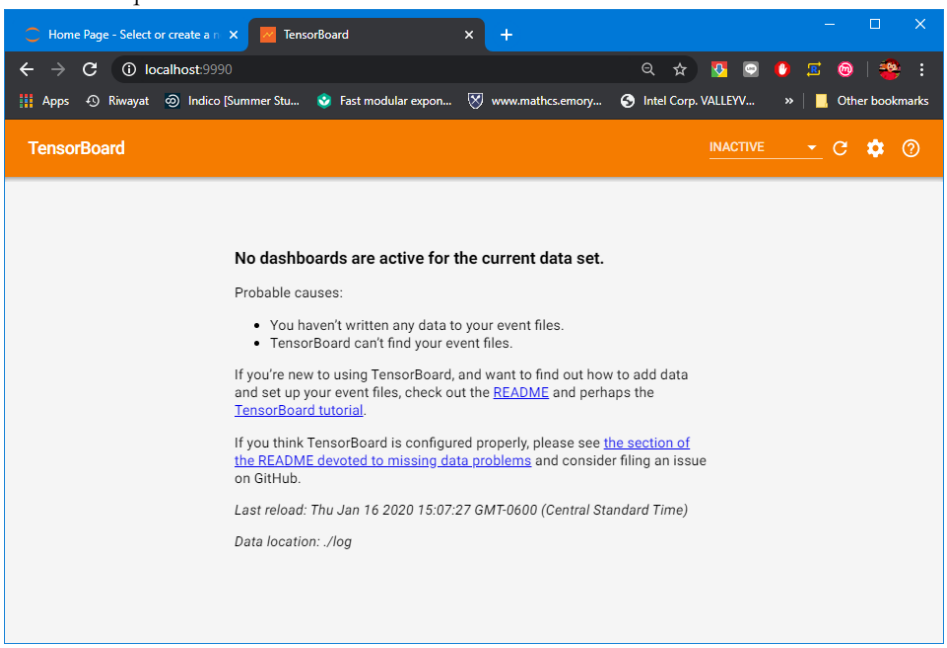
Go to your work directory.

```
cd $WORK/Tensor-Cores-Image-Classification/
```

Run the TensorBoard using this following command. Adjust the port number as needed if you have changed how you forward the port during SSH login.

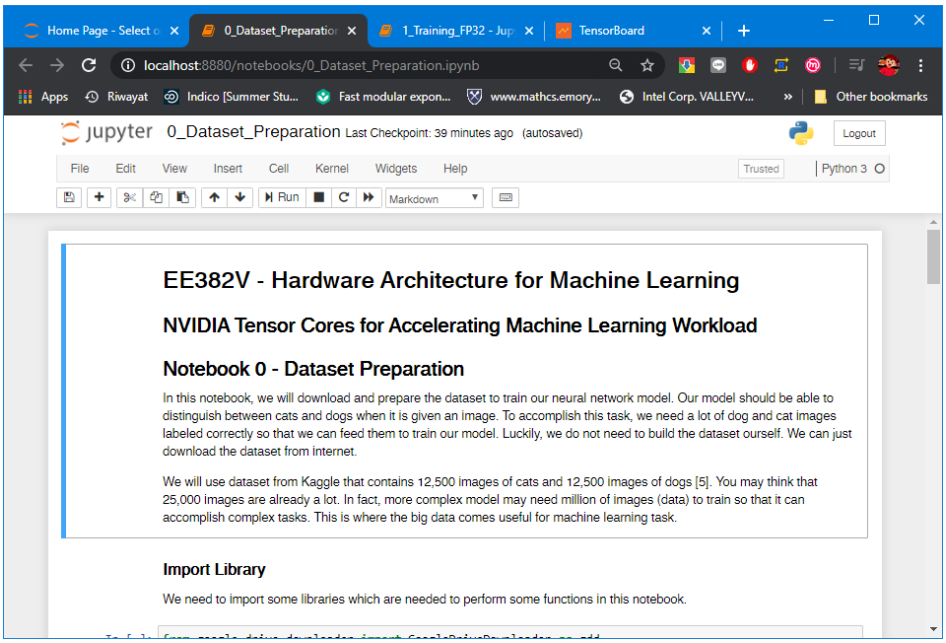
```
tensorboard --logdir ./log --port 9992
```

You should be able to access the TensorBoard on your computer by accessing http://localhost:9990 on your favourite browser. Please adjust the port number if you have changed them during the SSH login. Insert the token if requested.



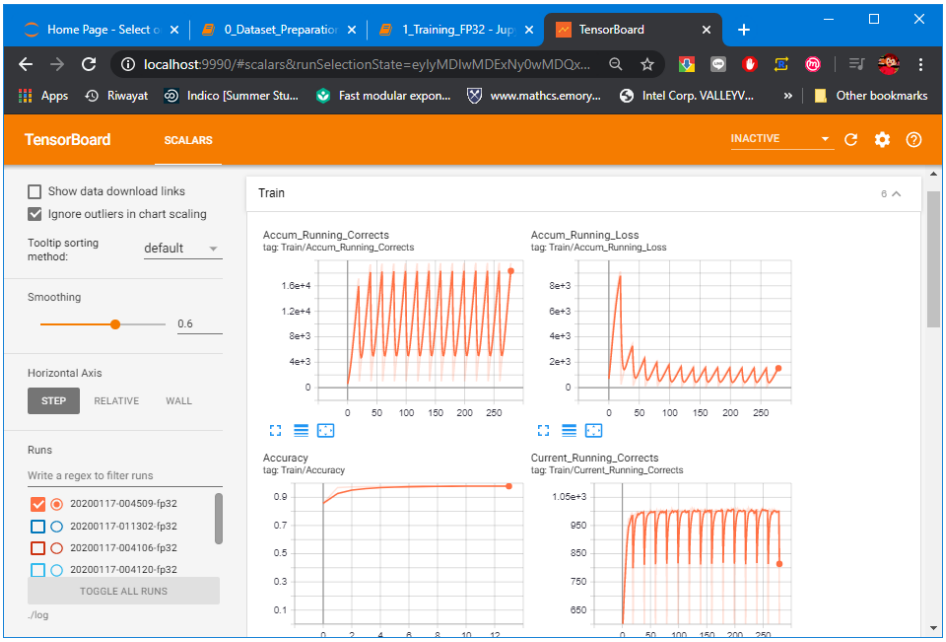
C. Dataset Preparation

In this section, we will download and prepare our dataset. Please open notebook `0_Dataset_Preparation.ipynb` on the Jupyter Notebook and follow the instructions on the notebook. After you have finished with the notebook, use `File -> Close and Halt` to close the notebook and kill the Python process associated with the notebook.



D. Training with Full Precision (FP32)

In this section, we will train our model using full precision. Please open notebook `1_Training_FP32.ipynb` on the Jupyter Notebook and follow the instructions on the notebook. While the model training is running, you can monitor the progress using TensorBoard.



As an addition, you can also monitor the GPU Memory Utilization by executing command `nvidia-smi` on the first terminal on MobaXterm (assuming the first terminal is idle).

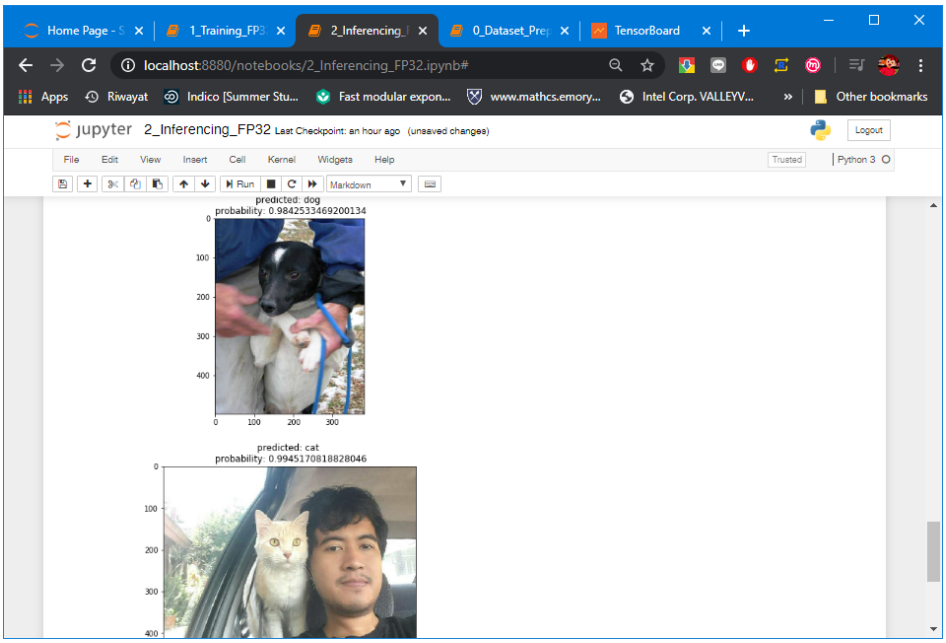
```
(CatsandDogsClassification) c228-103.maverick2(1062)$ nvidia-smi
Fri Jan 17 02:11:06 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.56       Driver Version: 418.56       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-PCIE...  On   | 00000000:AF:00.0 Off |                  Off |
| N/A   48C    P0    45W / 250W |     11MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-PCIE...  On   | 00000000:D8:00.0 Off |                  Off |
| N/A   34C    P0    26W / 250W |     11MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

After you have finished with the notebook, use `File -> Close and Halt` to close the notebook and kill the Python process associated with the notebook.

E. Inference with Full Precision (FP32)

In this section, we will make prediction from our trained model using full precision. Please open notebook `2_Inference_FP32.ipynb` on the Jupyter Notebook and follow the instructions on the notebook. If your model has been trained correctly, it can classify the test dataset (image) into cat or dog. After you have finished with the notebook, use `File -> Close and Halt` to close the notebook and kill the Python process associated with the notebook.



F. Training with Half Precision (FP16)

In this section, we will train our model using half precision. Please open notebook `3_Training_FP16.ipynb` on the Jupyter Notebook and follow the instructions on the notebook. As usual, while the model training is running, you can monitor the progress using TensorBoard. As an addition, you can also monitor the GPU Memory Utilization by executing command `nvidia-smi` on the first terminal on MobaXterm (assuming the first terminal is idle).

Compare your observation in half precision training with the full precision training (e.g., GPU memory utilization, time needed to train the model, accuracy, loss, model checkpoint size). You also need to compare which part of the code that should be modified to enable half precision training. After you have finished with the notebook, use `File -> Close and Halt` to close the notebook and kill the Python process associated with the notebook.

G. Inference with Half Precision (FP16)

In this section, we will make prediction from our trained model using half precision. Please open notebook `4_Inference_FP16.ipynb` on the Jupyter Notebook and follow the instructions on the notebook. Your task is to modify our prediction method to use FP16. If your model has been trained correctly, it can classify the test dataset (image) into cat or dog. After you have finished with the notebook, use `File -> Close and Halt` to close the notebook and kill the Python process associated with the notebook.

H. Profiling Training and Inference with Full Precision (FP32)

You might be wondering if we really use the Tensor Cores during FP16 training. In this section, we will try to profile both training and inference with full precision using nvprof and pyprof. We will not use Jupyter Notebook in this section.

1. Install A PyTorch Extension (Apex)

   We will need to install A PyTorch Extension (Apex) from NVIDIA as follows. It will take some time to compile the modules. If you are in Maverick2 Compute Node, don't forget to run this command first.

   ```
   module load cuda/10.1
   ```

   Then, running these commands to download, compile, and install Apex module.

   ```
   git clone https://github.com/NVIDIA/apex
   cd apex
   pip install cxxfilt
   pip install -v --no-cache-dir --global-option="--cpp_ext" --global-option="--cuda_ext" ./
   ```

2. Run The Profiler for Training with FP32

   There is a python script named `train_fp32.py`. This script is basically a stand-alone version of the `1_Training_FP32.ipynb` on Jupyter Notebook. We recommend to run the training no more than 3 epochs since running them with profiler significantly increase the runtime.

   ```
   nvprof -f -o train_fp32.sql --profile-from-start off -- python train_fp32.py
   ```

   Then, parse the profiling result into Python dictionary as follows.

   ```
   python -m apex.pyprof.parse train_fp32.sql > train_fp32.dict
   ```

Finally, you can display the dictionary on terminal.

```
python -m apex.pyprof.prof -w 100 -c kernel,op,sil,tc,flops,bytes,device,stream,block,grid train_fp32.dict
```

You can also export it to CSV if you want to open them in the spreadsheet (recommended).

```
python -m apex.pyprof.prof --csv -c kernel,mod,op,dir,sil,tc,flops,bytes,device,stream,block,grid train_fp32.dict > train_fp32.csv
```

If you receive this `AssertionError` as shown below.

```
Traceback (most recent call last):
  File "/work/06156/bagus/anaconda3/envs/CatsandDogsClassification/lib/python3.7/runpy.py", line 193, in _run_module_as_main
    "__main__", mod_spec)
  File "/work/06156/bagus/anaconda3/envs/CatsandDogsClassification/lib/python3.7/runpy.py", line 85, in _run_code
    exec(code, run_globals)
  File "/work/06156/bagus/anaconda3/envs/CatsandDogsClassification/lib/python3.7/site-packages/apex/pyprof/prof/__main__.py", line 10, in <module>
    main()
  File "/work/06156/bagus/anaconda3/envs/CatsandDogsClassification/lib/python3.7/site-packages/apex/pyprof/prof/prof.py", line 224, in main
    bytes = xx.bytes()
  File "/work/06156/bagus/anaconda3/envs/CatsandDogsClassification/lib/python3.7/site-packages/apex/pyprof/prof/reduction.py", line 107, in bytes
    return self.elems() * Utility.typeToBytes(self.type)
  File "/work/06156/bagus/anaconda3/envs/CatsandDogsClassification/lib/python3.7/site-packages/apex/pyprof/prof/utility.py", line 20, in typeToBytes
    assert False
AssertionError
```

Please open `utility.py` file highlighted in red above (your path will be different). Please make changes as shown below.

Before

```
@staticmethod
def typeToBytes(t):
    if (t in ["uint8", "int8", "byte", "char"]):
        return 1
    elif (t in ["float16", "half", "int16", "short"]):
        return 2
    elif (t in ["float32", "float", "int32", "int"]):
        return 4
    elif (t in ["int64", "long", "float64", "double"]):
        return 8
    assert False

@staticmethod
def typeToString(t):
    if (t in ["uint8", "byte", "char"]):
        return "uint8"
    elif (t in ["int8",]):
        return "int8"
    elif (t in ["int16", "short",]):
        return "int16"
    elif (t in ["float16", "half"]):
        return "fp16"
    elif (t in ["float32", "float"]):
        return "fp32"
    elif (t in ["int32", "int",]):
        return "int32"
    elif (t in ["int64", "long"]):
        return "int64"
    elif (t in ["float64", "double",]):
        return "fp64"
    assert False
```

After

```
@staticmethod
def typeToBytes(t):
    if (t in ["uint8", "int8", "byte", "char", "bool"]):
        return 1
    elif (t in ["float16", "half", "int16", "short"]):
        return 2
    elif (t in ["float32", "float", "int32", "int"]):
        return 4
    elif (t in ["int64", "long", "float64", "double"]):
        return 8
    assert False

@staticmethod
def typeToString(t):
    if (t in ["uint8", "byte", "char"]):
        return "uint8"
    elif (t in ["int8",]):
        return "int8"
    elif (t in ["int16", "short",]):
        return "int16"
    elif (t in ["float16", "half"]):
        return "fp16"
    elif (t in ["float32", "float"]):
        return "fp32"
    elif (t in ["int32", "int",]):
        return "int32"
    elif (t in ["int64", "long"]):
        return "int64"
    elif (t in ["float64", "double",]):
        return "fp64"
    elif (t in ["bool"]):
        return "bool"
    assert False
```

If completed successfully, you should see table shown below. (Note: table below is just an example).

```
Kernel                 Op              Sil(ns)     TC FLOPs         Bytes        Dev Str Block         Grid
cudnn::gemm::comput conv2d               2912 -            0             0    0   7 128,1,1       99,1,1
volta_fp16_scudnn_f conv2d           21462046 -  241692573696    1952467328    0   7 128,1,1       100352,1,1
elementwise_kernel   __add__             3168 -            1             8    0   7 128,1,1       1,1,1
batch_norm_collect_ batch_norm       17792793 -   6576668672    6576668672    0   7 512,1,1       64,1,1
batch_norm_transfor batch_norm        4651069 -   6576668672    6576668672    0   7 512,1,1       64,1,1024,1
elementwise_kernel   relu             5622389 -    822083584    3288334336    0   7 512,1,1       1605632,1,1
max_pool_forward_nc max_pool2d        6126225 -            0             0    0   7 256,1,1       802816,1,1
cudnn::gemm::comput conv2d               1760 -            0             0    0   7 128,1,1       25,1,1
volta_fp16_s884cudn conv2d            2280367 1   26306674688     822091776    0   7 256,1,1       12544,1,1
elementwise_kernel   __add__             1792 -            1             8    0   7 128,1,1       1,1,1
batch_norm_collect_ batch_norm        3903683 -   1644167168    1644167168    0   7 512,1,1       64,1,1
batch_norm_transfor batch_norm        1211926 -   1644167168    1644167168    0   7 512,1,1       64,1,1024,1
elementwise_kernel   relu             1401174 -    205520896     822083584    0   7 512,1,1       401408,1,1
...
```

Metric	Description
Kernel	Shows kernel name. Use the CSV version to obtain full kernel name.
Op	Shows the operation inside the neural network.
Sil	Shows the silicon time in nano second; how long the kernel executes.
TC	Shows whether this kernel executes using Tensor Cores or not. ("1" means yes)
FLOPs	Shows performance in Floating Point Operation per Seconds
Bytes	Shows number of bytes transferred in and out of GPU Memory
Dev	Shows device ID. We only use one GPU with GPU ID 0.
Str	Shows CUDA Stream ID.
Block	Shows kernel launch parameter; How many thread in one block.
Grid	Shows kernel launch parameter; How many block in one grid.

If you want to add more data column, please visit [11].

3. Measure The Tensor Cores Usage During Training with FP32

In previous step, we can observe whether the Tensor Cores are being used or not for a particular operation (used or not). In this step, we can find out how much the utilization of Tensor Cores if they are being used. Since we need to collect metrics from performance counter, running this step will take longer compared to previous step. Use the command below to do profiling with metric `tensor_precision_fu_utilization`. This returns the utilization level of the multiprocessor function units executing Tensor Cores instructions on a scale of 0 to 10. Any kernel showing a non-zero value is using Tensor Cores.

```
nvprof -m tensor_precision_fu_utilization -- python train_fp32.py
```

You can use this command to ask the profiler to generate CSV file instead of displaying it on terminal (recommended).

```
nvprof --csv --log-file train_fp32_util.csv -m
tensor_precision_fu_utilization -- python train_fp32.py
```

If the profiler completed successfully, you will see the table like the one below.

```
Invocations                               Metric Name                                 Metric Description         Min         Max         Avg
Device "Tesla V100-PCIE-16GB (0)"
    Kernel: _ZN2at6native18elementwise_...
       3186                tensor_precision_fu_utilization   Tensor-Precision Function Unit Utilization    Idle (0)    Idle (0)    Idle (0)
    Kernel: _ZN2at6native18elementwise_...
         60                tensor_precision_fu_utilization   Tensor-Precision Function Unit Utilization    Idle (0)    Idle (0)    Idle (0)
    Kernel: volta_fp16_scudnn_fp16_128x...
         75                tensor_precision_fu_utilization   Tensor-Precision Function Unit Utilization    Idle (0)    Idle (0)    Idle (0)
    Kernel: volta_fp16_s884cudnn_fp16_2...
        225                tensor_precision_fu_utilization   Tensor-Precision Function Unit Utilization     Mid (4)     Mid (5)     Mid (4)
    Kernel: Volta_hmma_implicit_gemm_fp...
        600                tensor_precision_fu_utilization   Tensor-Precision Function Unit Utilization    High (7)    High (8)    High (7)
```

You can add more metrics to profile. The more metrics you add, the longer the time needed to execute the application with profiler running. To show all available metrics, run this command.

```
nvprof --query-metrics
```

4. Run The Profiler for Inference with FP32
   Repeat step 2 but using `infer_fp32.py`. The stand-alone version of inference is a bit different than the one in Jupyter Notebook (i.e., `2_Inference_FP32.ipynb`). Instead of displaying the image and the prediction result, the stand-alone version will output a CSV file that contains the image file name, prediction, and probability. You can see the inference result by opening CSV file `infer_fp32_result.csv`. You can also adjust the number of images that you want to infer.

5. Measure The Tensor Cores Usage During Inference with FP32
   Repeat step 3 but using `infer_fp32.py`.

I. Profiling Training and Inference with Half Precision (FP16)

In this section, you have to repeat step 2 until step 5 from Section H using `train_fp16.py` and `infer_fp16.py` for training and inference, respectively. Both of file are still empty. Your task is to modify `train_fp32.py` and `infer_fp32.py` so that both training and inference can be done in half precision and thus we can take advantage of Tensor Cores.

J. Analysis and Discussion

Here is some questions to help you analyze and discuss the result of this assignment. You are free to improvise and come up with your analysis and discussion.

1. How to modify the FP32 training and inference to use the Tensor Cores?
2. How is the comparison between the time needed to train the model in FP32 without Tensor Cores and the time needed to train the model in FP16 with Tensor Cores?
3. How is the comparison between the GPU Memory needed to train the model in FP32 without Tensor Cores and the GPU Memory needed to train the model in FP16 with Tensor Cores? Does training using FP16 allow you to increase the batch size?
4. How is the comparison between the accuracy and loss of the model in FP32 and the model in FP16? Does training in FP16 affect the model accuracy and loss?
5. What is the advantage of using FP16 in terms of memory bandwidth required and arithmetic intensity? (hint: compare the FLOPs and Bytes of data transferred in or out the GPU memory from profiling for each kernel).
6. How do you translate the advantage of using FP16 and Tensor Cores in terms of power efficiency (e.g., performance per watt)?
7. NVIDIA stated that Tensor Cores brings 8x speed-up for FP16 computation compared to FP32 computation without Tensor Cores. Why can we see speed-up for around 8x by using FP16 and Tensor Cores for training?
8. NVIDIA Tesla T4 is a GPU that is specifically designed for inference and it has Tensor Cores. How do you observe the advantages of using Tensor Cores and FP16 for inference (e.g., performance-wise, power-wise)?
9. Why does the Tensor Cores cannot be used for all kernels although we have modified the training and inference into FP16? Is there any specific requirements to use Tensor Cores? What kind of kernels that can run on Tensor Cores?

K. References

[1] https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/
[2] https://portal.tacc.utexas.edu/user-guides/maverick2
[3] https://mobaxterm.mobatek.net/download.html
[4] https://www.anaconda.com/distribution/
[5] https://www.kaggle.com/c/dogs-vs-cats/data
[6] http://www.image-net.org
[7] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016.
[8] https://medium.com/predict/using-pytorch-for-kaggles-famous-dogs-vs-cats-challenge-part-1-preprocessing-and-training-407017e1a10c
[9] https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/
[10] https://github.com/pytorch/examples/blob/42e5b996718797e45c46a25c55b031e6768f8440/imagenet/main.py#L89-L101
[11] https://github.com/NVIDIA/apex/tree/master/apex/pyprof