

**Applied Accelerated Artificial Intelligence**  
**Prof. Bharatkumar Sharma**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Palakkad**

**Lecture - 38**  
**Distributed Deep Learning using TensorFlow and Horovod**

(Refer Slide Time: 00:15)

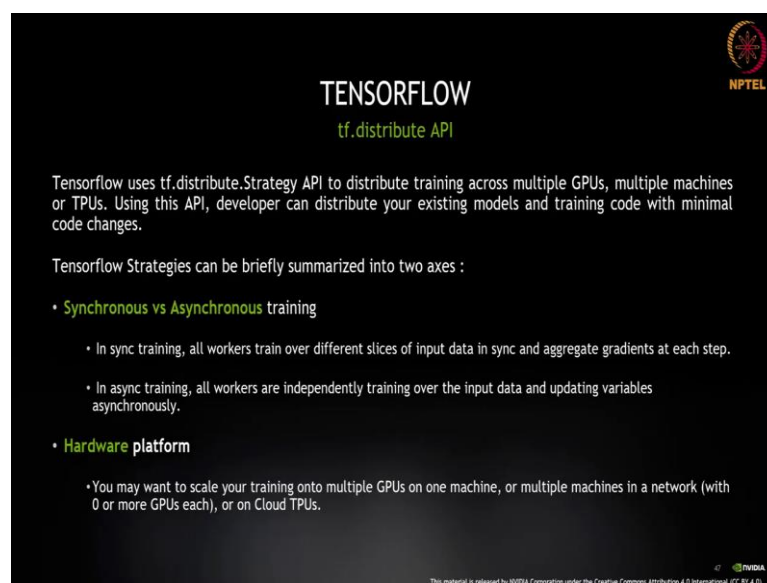


Welcome back everyone. This is the 3rd lecture on Distributed Deep Learning. So, far what we have covered is the necessity of using deep learning, we also covered the part of what is the impact of different kinds of system topology. And, we saw a demo of on a machine which is called a DGXV and which is consisting of 8 volta cards.

And, we saw the impact of latency and throughput and the effect of using peer to peer transfers using NVLink versus the non-NVLink based system and how the it can get impacted. We briefly covered the usage of NGC and how TensorFlow and or what kind of support distributed deep learning.

Today, we will go slightly deeper into two frameworks. One is TensorFlow, another is Horovod and we will look at a demo of it. Followed by in the end, we will look at what kind of a problems that you can face when you start using distributed deep learning and what convergence problems can come, how the accuracy get effected and how you can solve some of those problems as well. So, let us get started and with this we are going to start with distributed training and TensorFlow.

(Refer Slide Time: 02:01)



The slide is titled "TENSORFLOW" in large white letters, with "tf.distribute API" in green below it. In the top right corner is the NPTEL logo. The main text explains that TensorFlow uses the tf.distribute.Strategy API for distributed training across GPUs, machines, or TPUs. It then summarizes two axes of TensorFlow strategies: "Synchronous vs Asynchronous training" and "Hardware platform". The first axis has two bullet points: one for sync training (all workers train in sync and aggregate gradients) and one for async training (workers train independently and update variables asynchronously). The second axis has one bullet point: scaling training onto multiple GPUs on one machine, multiple machines in a network, or on Cloud TPUs. At the bottom, there is a small NVIDIA logo and a Creative Commons license notice.

**TENSORFLOW**  
tf.distribute API

Tensorflow uses tf.distribute.Strategy API to distribute training across multiple GPUs, multiple machines or TPUs. Using this API, developer can distribute your existing models and training code with minimal code changes.

Tensorflow Strategies can be briefly summarized into two axes :

- **Synchronous vs Asynchronous training**
  - In sync training, all workers train over different slices of input data in sync and aggregate gradients at each step.
  - In async training, all workers are independently training over the input data and updating variables asynchronously.
- **Hardware platform**
  - You may want to scale your training onto multiple GPUs on one machine, or multiple machines in a network (with 0 or more GPUs each), or on Cloud TPUs.

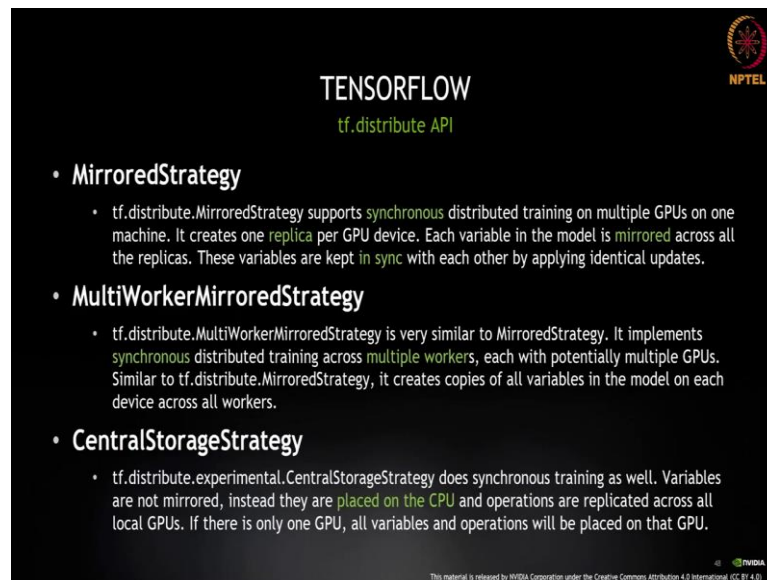
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

So, as we said all of the frameworks support distributed deep learning. This one is particularly referring to the TensorFlow APIs. If I look at from a point of view of this support, basically it has multiple implementation. And, these implementations are based on two types or two axes. One is synchronous versus asynchronous and the second one is based on the hardware type that you are going to run this particular strategy on.

So, in the synchronous versus asynchronous, we have touched this in the previous lecture. In the synchronous training all the workers train over different slices in a synchronous fashion and the gradient aggregation happens at every step. While, in asynchronous all the workers work independently training over the input data and updating the variables asynchronously.

Again, the second axes is hardware platform. TensorFlow supports multiple hardware. It supports GPU, it also supports multi-distributed training across CPUs and also on Cloud TPUs which is a Tensor Processing Units by Google.

(Refer Slide Time: 03:22)



A presentation slide titled "TENSORFLOW" with the subtitle "tf.distribute API". It lists three distribution strategies: MirroredStrategy, MultiWorkerMirroredStrategy, and CentralStorageStrategy, each with a brief description of its functionality. The slide includes the TensorFlow logo and an NPTEL logo in the top right corner. At the bottom, there is a small NVIDIA logo and a Creative Commons license notice.

**TENSORFLOW**  
tf.distribute API

- **MirroredStrategy**
  - tf.distribute.MirroredStrategy supports **synchronous** distributed training on multiple GPUs on one machine. It creates one **replica** per GPU device. Each variable in the model is **mirrored** across all the replicas. These variables are kept **in sync** with each other by applying identical updates.
- **MultiWorkerMirroredStrategy**
  - tf.distribute.MultiWorkerMirroredStrategy is very similar to MirroredStrategy. It implements **synchronous** distributed training across **multiple workers**, each with potentially multiple GPUs. Similar to tf.distribute.MirroredStrategy, it creates copies of all variables in the model on each device across all workers.
- **CentralStorageStrategy**
  - tf.distribute.experimental.CentralStorageStrategy does synchronous training as well. Variables are not mirrored, instead they are **placed on the CPU** and operations are replicated across all local GPUs. If there is only one GPU, all variables and operations will be placed on that GPU.

© 2018 NVIDIA  
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

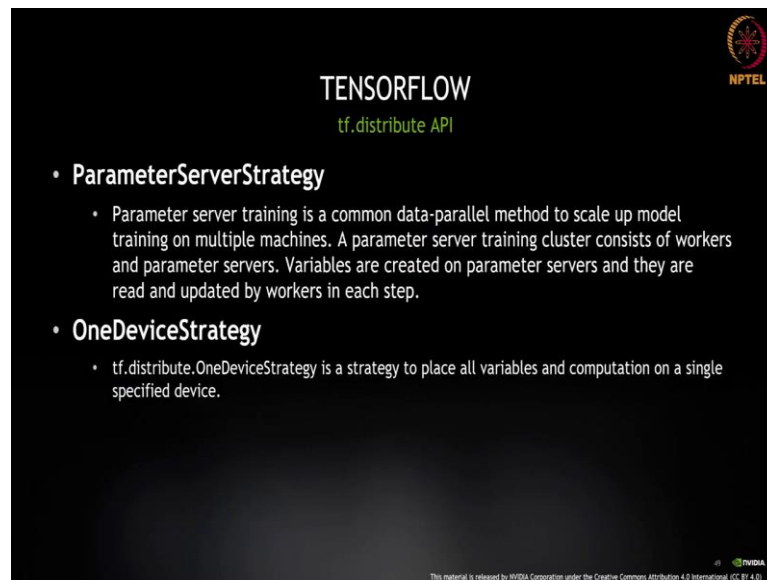
So, TensorFlow supports total of 5 strategies that we can use on the GPUs. There are more strategies once that once can be used for accelerator more like for Google TPUs and all. But, we are not going to touch upon them in this particular session, but let me go through the ones which are useful for the accelerators.

So, the first one is the MirroredStrategy, I have kind of highlighted some of the keywords. First, is it is synchronous in nature. It creates the replica per GPU. So, one of the key things its part of the data parallel and you can see it will create a replica per GPU. And, each variable in the model is mirrored and the variables are always kept in sync and that is what mirrored strategy works on.

MultiWorkerMirroredStrategy as the name says, the keyword there is multiple workers. So, it is very similar to the MirroredStrategy, but it creates multiple worker each with potentially multiple GPUs. It creates similarly to mirrored strategic copy of all the variables in the model on each device and across all the workers. There is another strategy which can be used which is CentralStorageStrategy, where the variables are not mirrored.

The updation of the variables or the main copy of it is kept or placed on the CPU and the operations are replicated across local GPUs. So, all so, if there was suppose only one GPU, in that case it will reside only on that particular one GPU.

(Refer Slide Time: 05:13)



A presentation slide titled "TENSORFLOW" with the subtitle "tf.distribute API". It lists two strategies: "ParameterServerStrategy" and "OneDeviceStrategy", each with a brief description. The slide includes the NPTEL logo in the top right and a footer with the NVIDIA logo and a Creative Commons license notice.

## TENSORFLOW

tf.distribute API

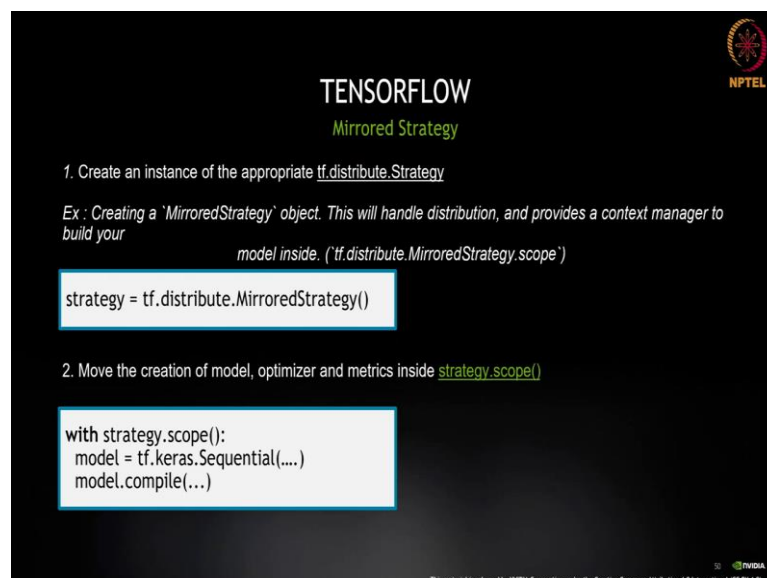
- **ParameterServerStrategy**
  - Parameter server training is a common data-parallel method to scale up model training on multiple machines. A parameter server training cluster consists of workers and parameter servers. Variables are created on parameter servers and they are read and updated by workers in each step.
- **OneDeviceStrategy**
  - tf.distribute.OneDeviceStrategy is a strategy to place all variables and computation on a single specified device.

© NVIDIA  
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

ParameterServerStrategy, we had covered to certain extent last time and it is common data parallel method to scale up. And, it kind of helps because all the workers are not talking to each other, while they fundamentally talk to a parameter server and it consist of multiple workers. So, the variables are created on the parameter servers and they are read and updated by the workers in each and every step.

Another one is OneDeviceStrategy, where you can define where to place all the variables, on which particular single device you would like to put them on.

(Refer Slide Time: 05:52)



A presentation slide titled "TENSORFLOW" with the subtitle "Mirrored Strategy". It provides a two-step guide to using the Mirrored Strategy, including code snippets for creating the strategy and using its scope. The slide includes the NPTEL logo in the top right and a footer with the NVIDIA logo and a Creative Commons license notice.

## TENSORFLOW

Mirrored Strategy

1. Create an instance of the appropriate `tf.distribute.Strategy`  
*Ex : Creating a 'MirroredStrategy' object. This will handle distribution, and provides a context manager to build your model inside. ('tf.distribute.MirroredStrategy.scope')*  

```
strategy = tf.distribute.MirroredStrategy()
```
2. Move the creation of model, optimizer and metrics inside `strategy.scope()`  

```
with strategy.scope():  
    model = tf.keras.Sequential(...)  
    model.compile(...)
```

© NVIDIA  
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

So, let us look at the steps which are required inside TensorFlow to convert your normal training to a distributed training. So, the first step in this particular example that we are showing is first of all we are using a mirrored strategy and here you can see the first thing is we are defining the strategy ourself.

The second step in this particular session, in this particular approach is to bundle your existing model within the scope of that strategy. So, you can see here that we are saying within the strategy dot scope, I am going to define my model. So, that model is defined within the scope of that particular strategy, that you have chosen.

If you are not using the distributed strategy, if you are doing it on a normal one GPU; this step is not required right. You would directly start creating your model layers. So, we can say it is quite simple to do this. And, the TensorFlow kind of makes the scaling very very easier by only a few lines of code. We will look at the code into more details later also.

(Refer Slide Time: 07:08)



But, before showing you the hands on part of it, let me also introduce you to another framework which is Horovod. And, then we will look at the demo of the Horovod and TensorFlow both of them.

(Refer Slide Time: 07:25)

A presentation slide titled "INTRODUCTION TO HOROVOD" with a dark blue background. The title is in white capital letters at the top center. In the top right corner, there is a small NPTEL logo. On the left side, there is a bulleted list of features in white text. On the right side, there is a Horovod logo consisting of a white hexagon with the word "HOROVOD" inside, surrounded by a ring of blue and white dots. Below the logo is the URL "horovod.ai" in green. At the bottom right, there is a small NVIDIA logo and a line of fine print in white.

## INTRODUCTION TO HOROVOD

- Horovod is a distributed deep learning training framework
- Supports TensorFlow, Keras, PyTorch, and Apache MXNet.
- Easy to Install and use
- Minimal code changes
- Proven Scalability
- Uses Advanced algorithms from HPC

[horovod.ai](https://horovod.ai)

NPTEL

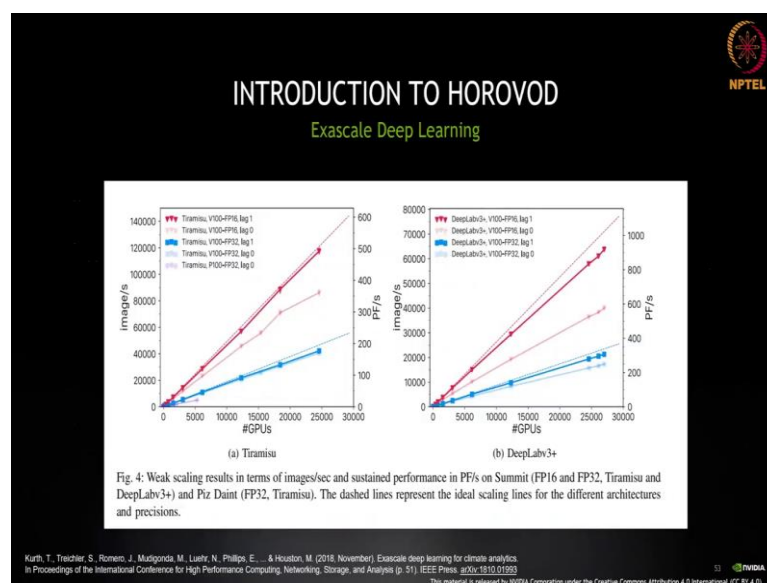
NVIDIA

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

So, Horovod basically is a library for distributed deep learning. It was initially a tool for internal use at Uber which was later open source for the community. Apart from the features which we have, it also have additional tools such as Horovod timeline which can be used to analyse the time taken for each operation and optimize accordingly. So, we can see what all features it has. It supports in the back end, it supports TensorFlow, it supports Keras, PyTorch, Apache's.

So, you can think of Horovod as a meta framework over existing frameworks like TensorFlow, PyTorch and all. It is very easy to install. There is almost minimal code changes that you have to do. And, Horovod has been used for multiple exercises and for even doing exascale kind of computing. It uses some advanced techniques to do really good scaling or it has different types of optimizers which are used in those scenarios.

(Refer Slide Time: 08:30)



So, Horovod has been used widely and is proven to be scalable across as I said 1000s of GPUs and nodes. This is from a paper Exascale Deep Learning for Climate Analytics and the model was trained on the summit supercomputer, with that let us now move to scaling a training script using Horovod.

But, the key point again that I would like to have here is that this is like a petaflop simulations and the number of GPUs across which it was run. And, you can see it is scaling pretty well.

(Refer Slide Time: 09:07)

### HOROVOD

Horovod with Tensorflow

The following steps needs to be done while using Horovod with Tensorflow

1. Initialize Horovod
2. Pin each GPU to a single process.
3. Wrap the optimiser in Horovod Distributed optimiser. The distributed optimiser delegates gradient computation to the original optimiser, averages gradients using all-reduce or all-gather, and then applies those averaged gradients.
4. Broadcast the initial variable states from rank 0 to all other processes.
5. Modify your code to save checkpoints only on worker 0 to prevent other workers from corrupting them.

54 NVIDIA

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

Horovod actually takes its motivation from something called as MPI. MPI stands for Message Passing Interface. It is one of the most prevalent distributed computing model in the high performance computing environment. Who are working in molecular dynamics, computational fluid dynamics and all those kinds of high-performance computing domain, if you want to scale or if you want to distribute your work across multiple nodes; the model which is used is message passing interface.

And, Horovod uses the MPI model where each multi-CPU thread would be launched, each of them corresponding to one GPU which is very different from the TensorFlow case, where one CPU thread was used to handle all the GPUs. This was the design choice by the Horovod team.

And, it is very very critical to understand this because this is one of the fundamental reason why Horovod can scale really well. The steps which are required in the code to make use of Horovod is; obviously, like MPI we need to do initialization. Initialization is key to finding out different aspects behind the scenes and Horovod kind of sets the context there. Then, you pin each GPU to a single process.

So, you would define multiple you at runtime you will define how many processes you want and then for each process you will pin a particular GPU. So, you can pin one or more GPUs to a single process. Like TensorFlow again you are going to wrap the optimizer also inside Horovod distributed optimizer.

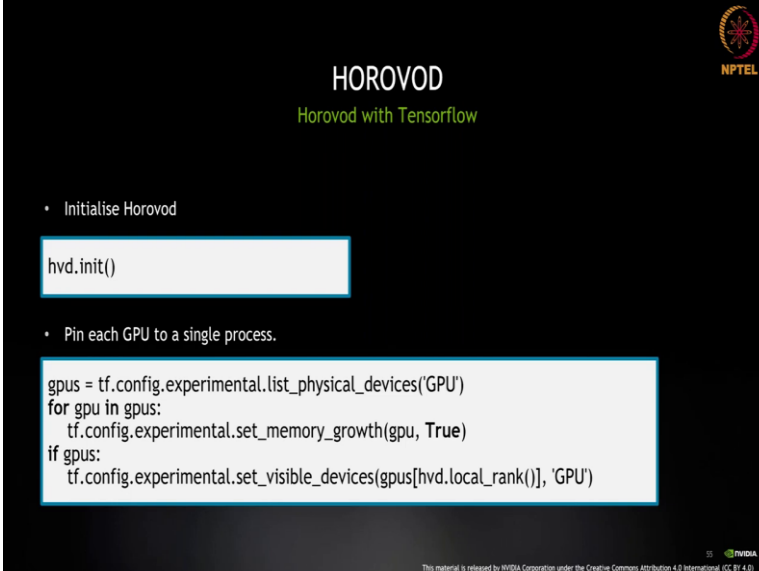
So, you are going to wrap the existing optimizer onto the Horovod distributed optimizer which will be responsible for calculating or delegating the gradient computation to the original optimizer. So, it does the gradient calculation to original optimizer whichever is there, but it takes the responsibility of averaging the gradient across multiple processes using MPI features, like all reduced, all gathered which are very well known in the MPI community.

And, it is proven to be working really well with different kinds of topologies using InfiniBand like structure and all. And, then applying those average gradients and once it is done with that. So, it is basically broadcasts the initial variable states from rank 0 to all the other processes. And, if you want you the code, you will have to change certain code to safety checkpoints and other things only by worker 0.



So, you do not want all of the workers or all the processes to save the checkpoint because the same state is maintained by everyone. So, there is no need of everyone writing to the same writing the same checkpointing values and corrupting them in short.

(Refer Slide Time: 12:19)

A presentation slide titled "HOROVOD" with the subtitle "Horovod with Tensorflow". The slide is dark-themed with a red NPTEL logo in the top right corner. It contains two bullet points: "Initialise Horovod" and "Pin each GPU to a single process.". The first bullet point is followed by a code block containing `hvd.init()`. The second bullet point is followed by a larger code block containing a Python script that lists physical devices, sets memory growth, and sets visible devices based on the local rank.

**HOROVOD**  
Horovod with Tensorflow

- Initialise Horovod

```
hvd.init()
```

- Pin each GPU to a single process.

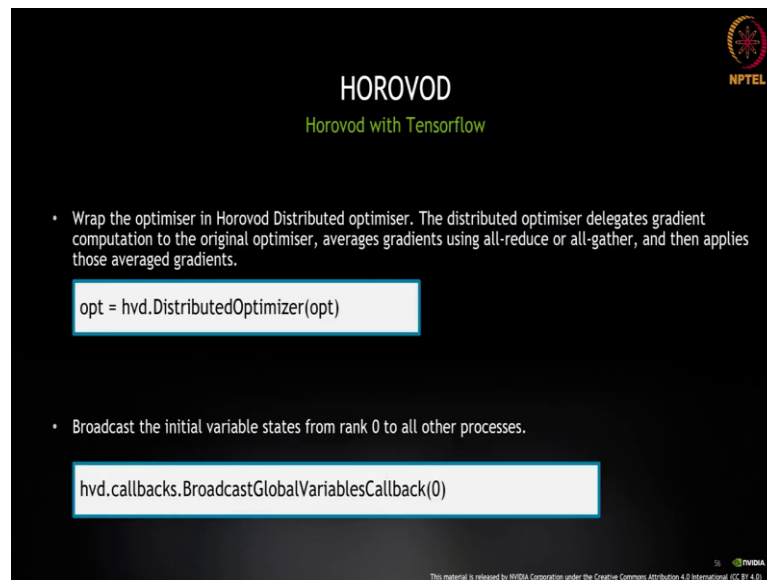
```
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')
```

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

So, from a code point of view this is how it will look like. We first import and then we will initialize the Horovod for the entire script. We then are going to pin the GPUs to a single process and you can see here the code which is we are first of all getting all the listed devices available. We get that particular list and for every GPU we are basically based on the rank of that particular process, like the process when you are launching said Horovod.

If you are launching two process, there will be process 1 will get a rank of 0, process 1 will get a rank of a process 2 will get a rank of 1. So, based on the rank which every Horovod process gets, you would do set visible device which means every process will get its own GPU. In TensorFlow, the strategy API would handle the updates, reading of the weights and making identical updates.

(Refer Slide Time: 13:37)



The slide features a dark background with the title 'HOROVOD' in white and 'Horovod with Tensorflow' in green below it. In the top right corner, there is a circular logo with a red and yellow design and the text 'NPTEL' underneath. The main content consists of two bullet points, each followed by a code snippet in a light blue box. The first bullet point describes wrapping an optimizer in Horovod's Distributed optimizer, and the code snippet is 'opt = hvd.DistributedOptimizer(opt)'. The second bullet point describes broadcasting initial variable states from rank 0, and the code snippet is 'hvd.callbacks.BroadcastGlobalVariablesCallback(0)'. At the bottom right, there is a small NVIDIA logo and a line of fine print.

## HOROVOD

Horovod with Tensorflow

- Wrap the optimiser in Horovod Distributed optimiser. The distributed optimiser delegates gradient computation to the original optimiser, averages gradients using all-reduce or all-gather, and then applies those averaged gradients.

```
opt = hvd.DistributedOptimizer(opt)
```

- Broadcast the initial variable states from rank 0 to all other processes.

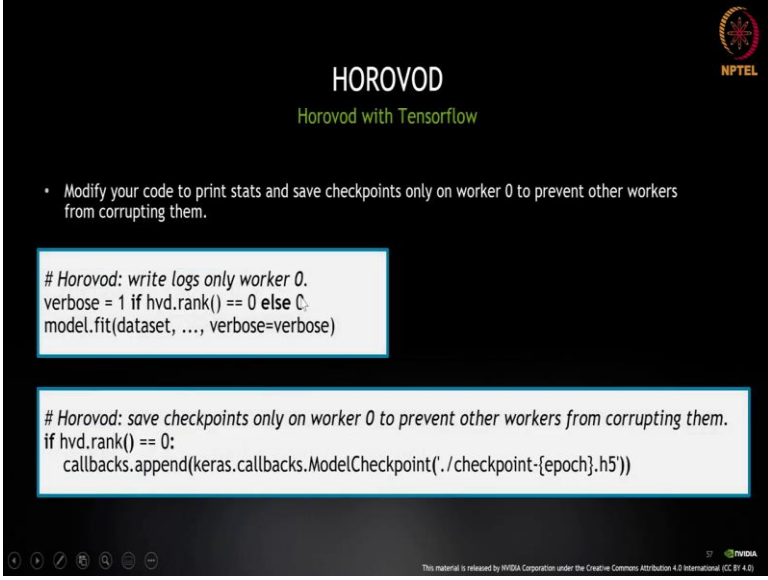
```
hvd.callbacks.BroadcastGlobalVariablesCallback(0)
```

10 NVIDIA  
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

But, in Horovod we can use the optimizer of a distributed optimizer method, updates are identical and are kept in sync. So, if you see here, we can choose our own optimizer and then wrap that optimizer inside the distributed optimizer. So, the current optimizer will continue to work, but the distributed optimizer as we said before, responsibility there for it is to make sure that it will basically take over and do the all reduce and all gathered part of it.

And, then do the averaging of the gradients and then you can also define callbacks from rank 0 to all the other processes.

(Refer Slide Time: 14:18)



A presentation slide titled "HOROVOD" with the subtitle "Horovod with Tensorflow". It features a bullet point about printing stats and saving checkpoints on worker 0. Two code snippets are shown in light blue boxes. The slide includes a navigation bar at the bottom and a footer with the NVIDIA logo and a Creative Commons license.

**HOROVOD**  
Horovod with Tensorflow

- Modify your code to print stats and save checkpoints only on worker 0 to prevent other workers from corrupting them.

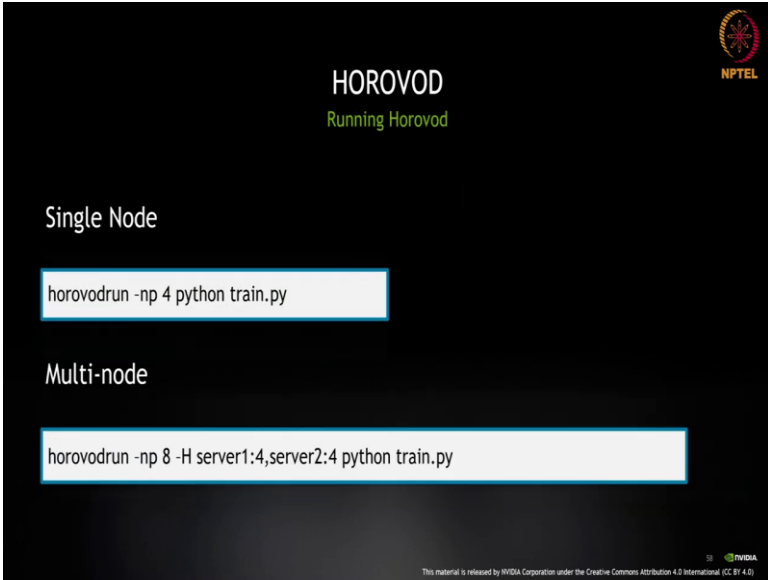
```
# Horovod: write logs only worker 0.  
verbose = 1 if hvd.rank() == 0 else 0  
model.fit(dataset, ..., verbose=verbose)
```

```
# Horovod: save checkpoints only on worker 0 to prevent other workers from corrupting them.  
if hvd.rank() == 0:  
    callbacks.append(keras.callbacks.ModelCheckpoint('./checkpoint-{epoch}.h5'))
```

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

The last, but not the least we already said to you that you can save checkpoints and we would ideally like only one worker and worker 0. So, that they are to prevent the other workers from corrupting them. So, you can say only if my rank is 0, then only I am going to basically save my checkpoints via basically a hook.

(Refer Slide Time: 14:49)



A presentation slide titled "HOROVOD" with the subtitle "Running Horovod". It shows two sections: "Single Node" and "Multi-node", each with a corresponding command in a light blue box. The slide includes a navigation bar at the bottom and a footer with the NVIDIA logo and a Creative Commons license.

**HOROVOD**  
Running Horovod

**Single Node**

```
horovodrun -np 4 python train.py
```

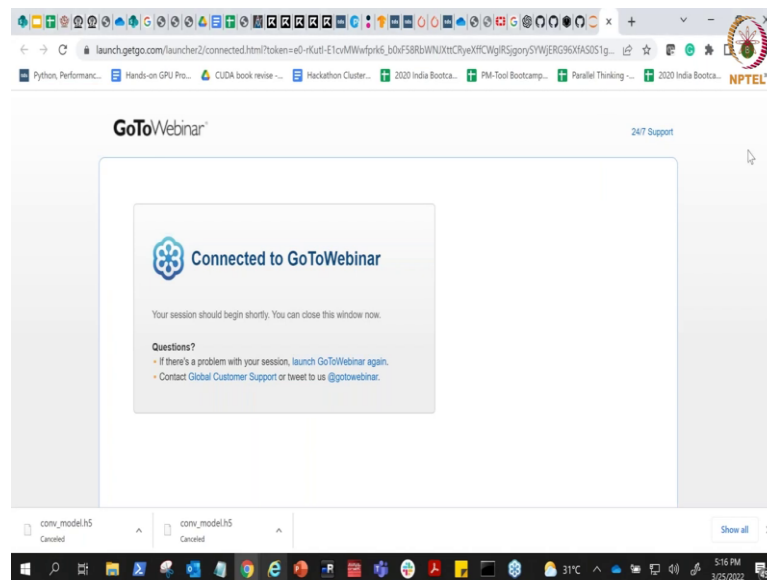
**Multi-node**

```
horovodrun -np 8 -H server1:4,server2:4 python train.py
```

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

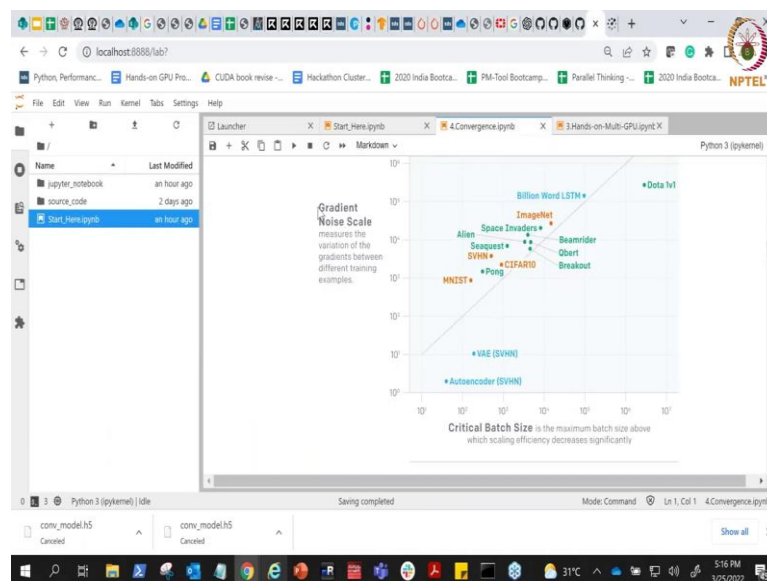
So, this is what kind of completes the flow of the Horovod part of it.

(Refer Slide Time: 14:57)



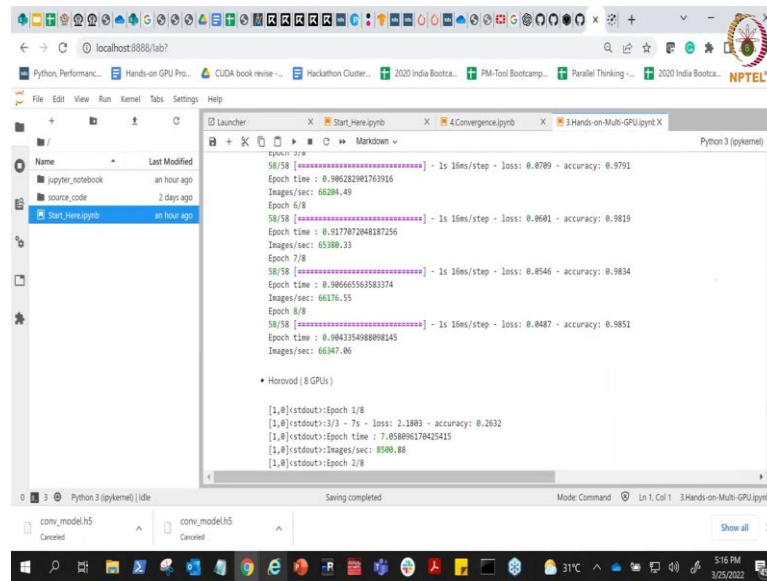
Let me take you to the live as into the code.

(Refer Slide Time: 14:59)



So, that you get an idea of how it looks like.

(Refer Slide Time: 15:01)



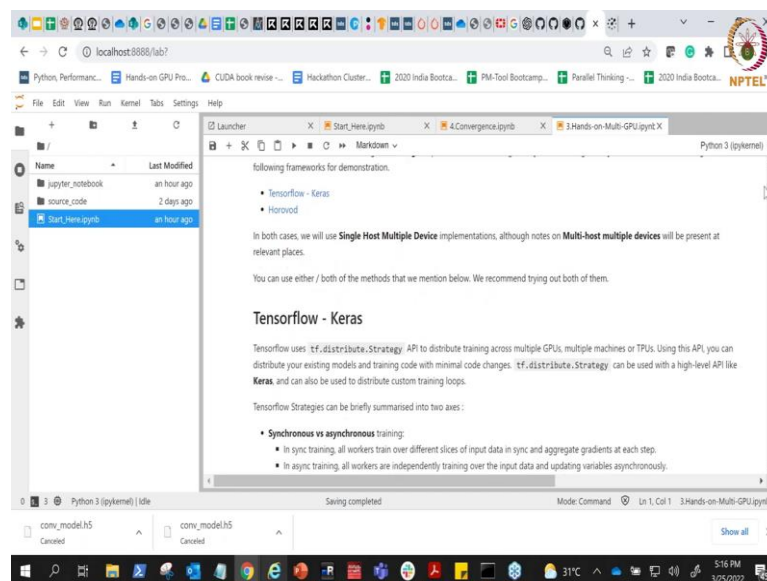
```
Epoch 58/58 [=====] - 1s 16ms/step - loss: 0.8709 - accuracy: 0.9791
Epoch time : 0.98628298769916
Images/sec: 66294.49
Epoch 6/8
Epoch 58/58 [=====] - 1s 16ms/step - loss: 0.8681 - accuracy: 0.9819
Epoch time : 0.9177872848187256
Images/sec: 65380.33
Epoch 7/8
Epoch 58/58 [=====] - 1s 16ms/step - loss: 0.8546 - accuracy: 0.9834
Epoch time : 0.90665565183374
Images/sec: 66376.55
Epoch 8/8
Epoch 58/58 [=====] - 1s 16ms/step - loss: 0.8487 - accuracy: 0.9851
Epoch time : 0.9843354988989145
Images/sec: 66347.86

Horovod (8 GPUs)

[1,0]<stdout>:Epoch 1/8
[1,0]<stdout>:1/3 - 7s - loss: 2.1883 - accuracy: 0.2632
[1,0]<stdout>:Epoch time : 7.858896178425415
[1,0]<stdout>:Images/sec: 8580.88
[1,0]<stdout>:Epoch 2/8
```

So, we are going to continue from our session which we did last time also.

(Refer Slide Time: 15:08)



```
following frameworks for demonstration.
• TensorFlow - Keras
• Horovod

In both cases, we will use Single Host Multiple Device implementations, although notes on Multi-host multiple devices will be present at relevant places.

You can use either / both of the methods that we mention below. We recommend trying out both of them.

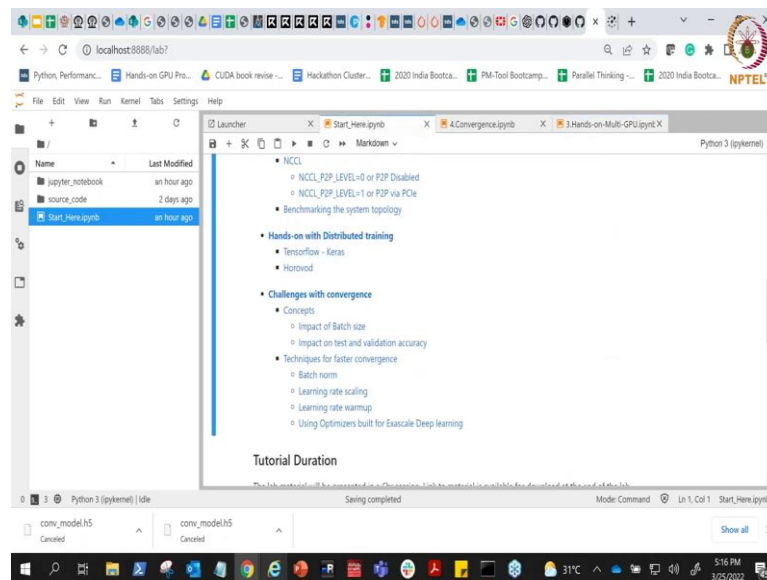
Tensorflow - Keras

Tensorflow uses tf.distribute.Strategy API to distribute training across multiple GPUs, multiple machines or TPUs. Using this API you can distribute your existing models and training code with minimal code changes. tf.distribute.Strategy can be used with a high-level API like Keras, and can also be used to distribute custom training loops.

Tensorflow Strategies can be briefly summarised into two axes :

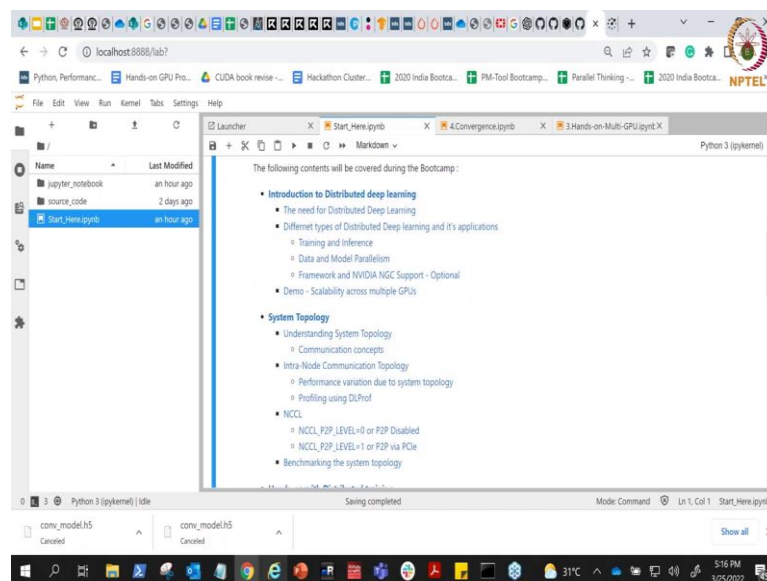
• Synchronous vs asynchronous training:
  • In sync training, all workers train over different slices of input data in sync and aggregate gradients at each step.
  • In async training, all workers are independently training over the input data and updating variables asynchronously.
```

(Refer Slide Time: 15:11)



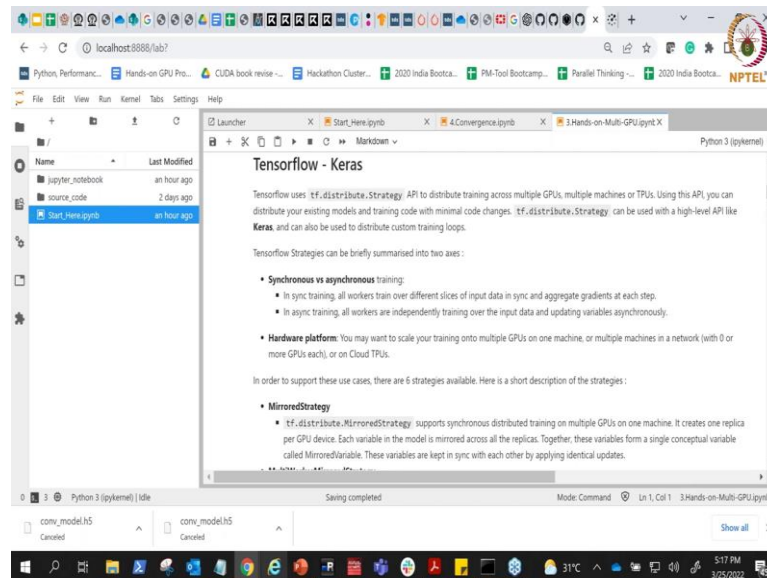
So, if you remember I launched the start here notebook.

(Refer Slide Time: 15:16)



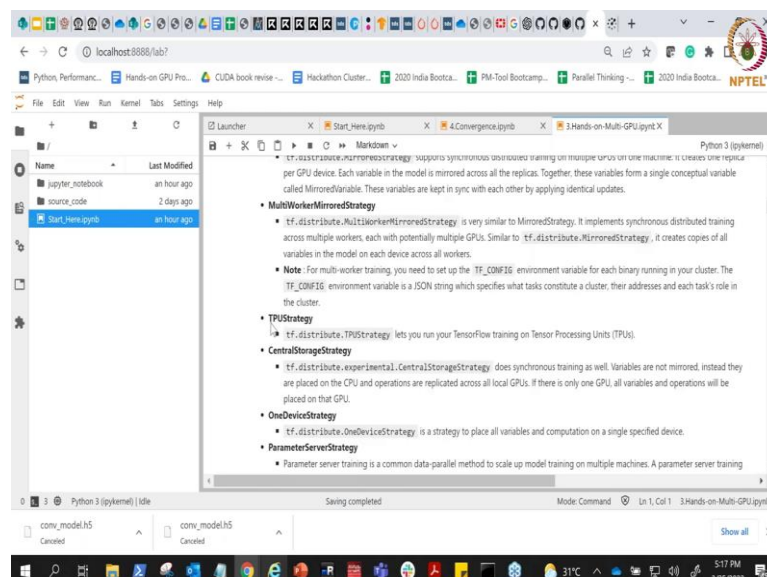
In the start here notebook, we basically went through the part of the concept or the necessity of distributed deep learning. Then, we saw the system topology and what was NCCL, how it was impacting the performance. And, today we are looking at the TensorFlow and Horovod implementation. So, again I am not going to repeat the theory part of it which I just explained in the method.

(Refer Slide Time: 15:39)



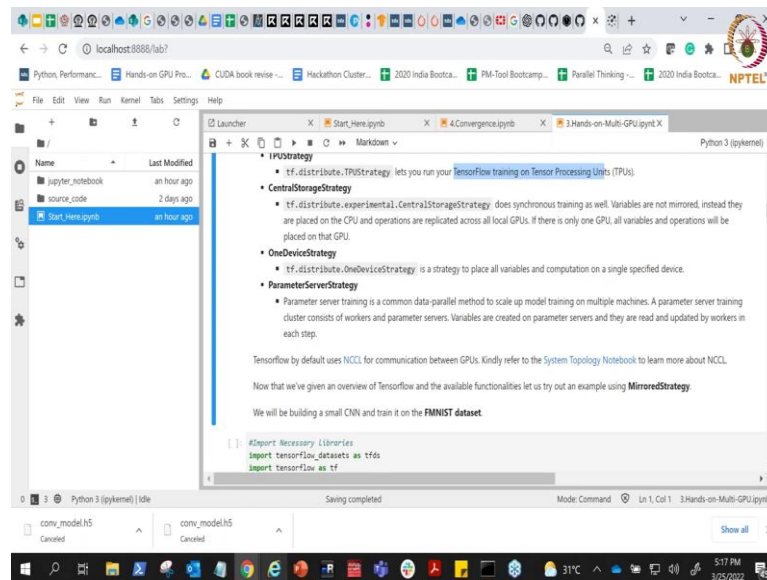
So, the all of these notebooks where they exist would be provided to you on the slack channel. The link would be provided. These are all open source notebooks. You can go ahead download them and try them in a multi GPU environment, you would require at least two GPUs to get a feeling of how you can distribute the work across multiple GPUs; within a single GPU environment these labs will be of almost of no use.

(Refer Slide Time: 16:05)

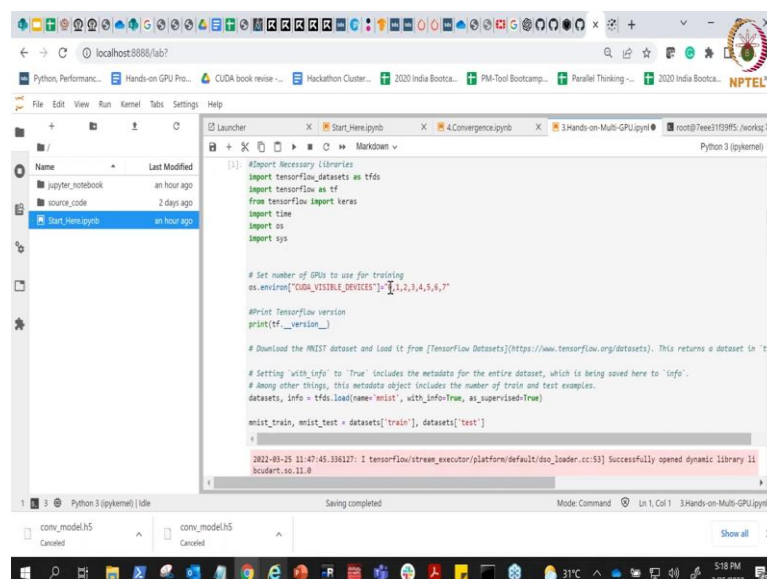


As we showed you earlier there are different kinds of strategies, like what we did not show you was there is a different strategy for TPU which is Google's TPU. And, you can run it and set it in case you are using any other kind of a hardware as well.

(Refer Slide Time: 16:21)



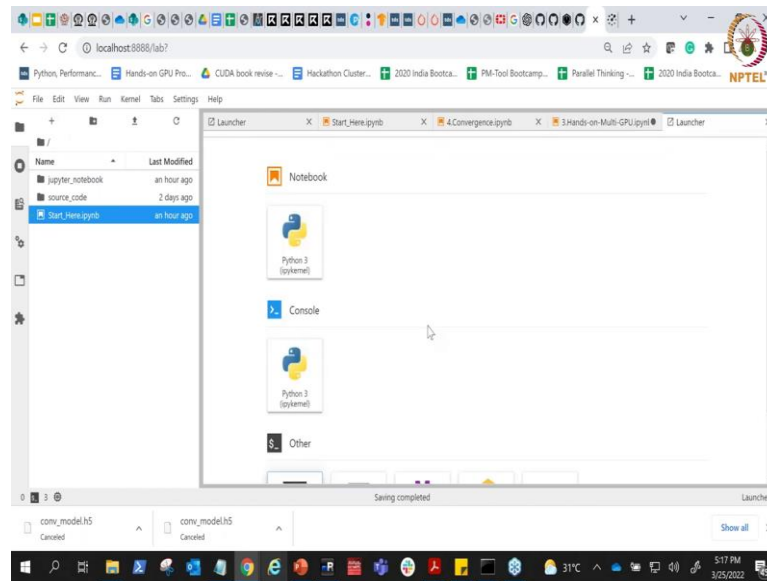
(Refer Slide Time: 16:23)



So, let me just start running the code one by one. So, like in the previously we are importing the tensorflow, then the keras. If you remember last time if I will show you the terminal ones.

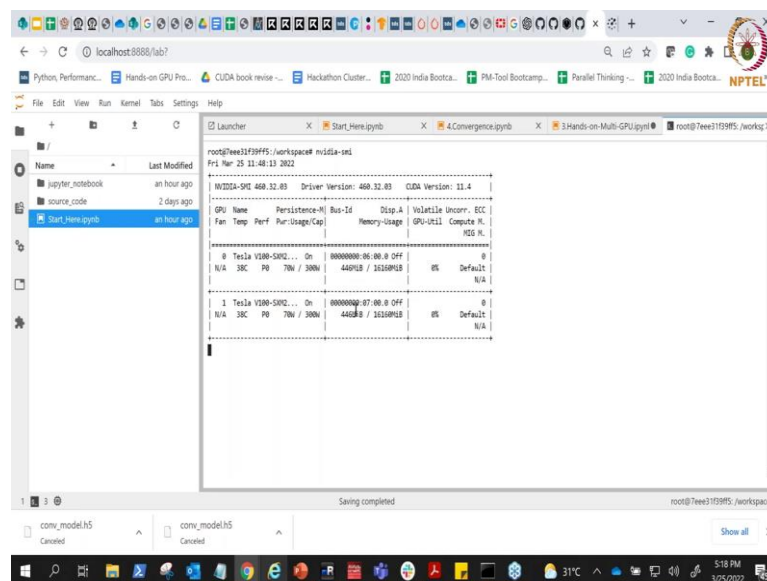


(Refer Slide Time: 16:39)



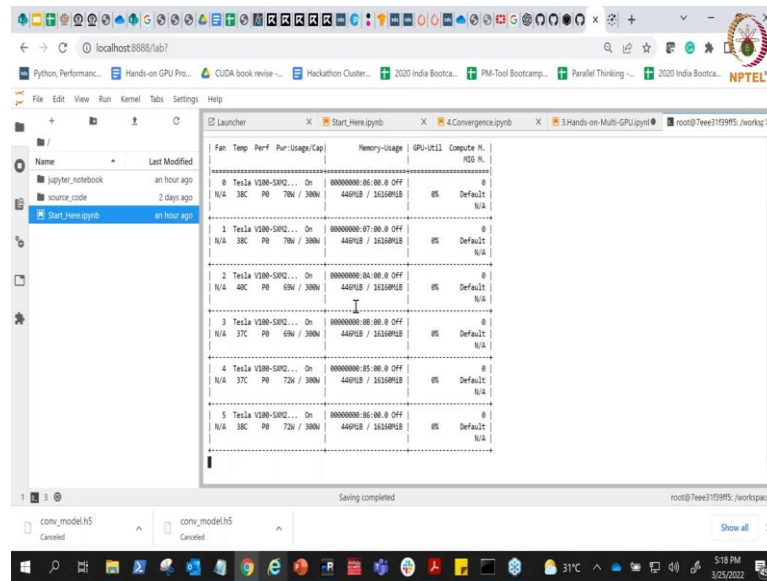
This particular yeah.

(Refer Slide Time: 16:49)

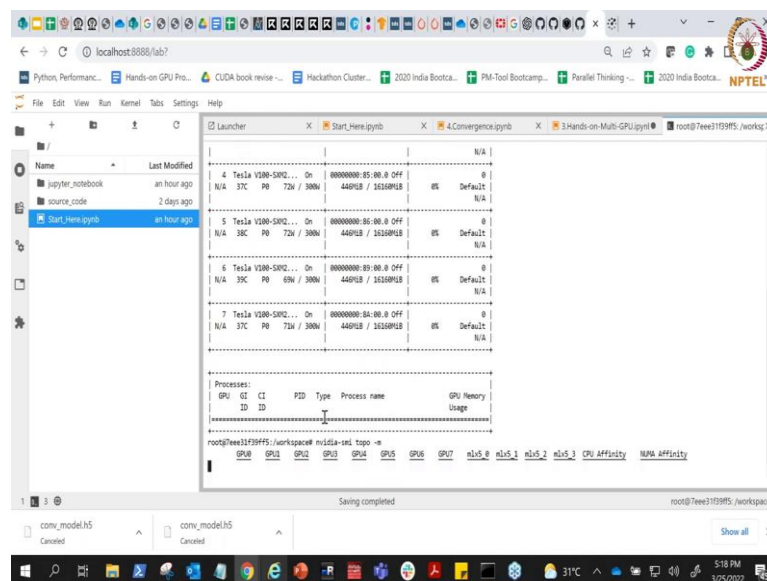


So, if you remember this particular machine, it has basically 8 GPUs.

(Refer Slide Time: 16:58)

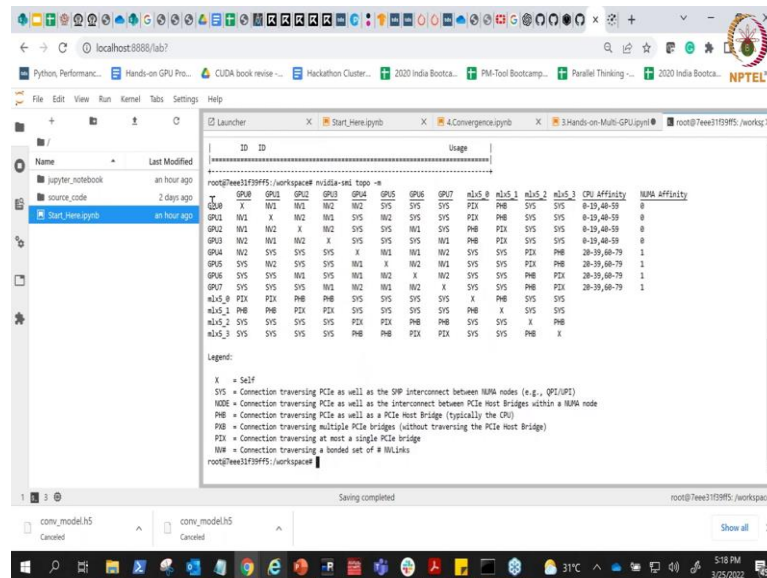


(Refer Slide Time: 16:59)



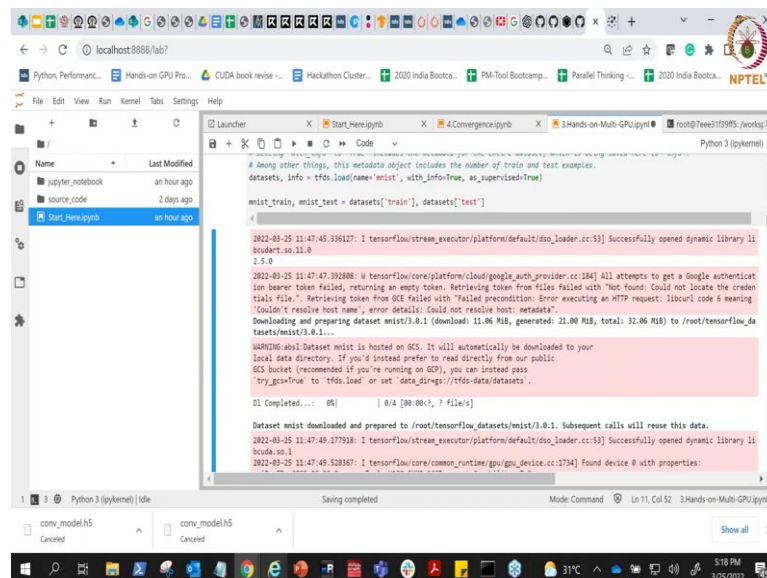
And, you can see here 1 0 1 2 3 4 5 6 7 and we also talk about how to see ok. So, there is a lag which is happening `nvidia-smi topo -m` will basically show you the system topology as well.

(Refer Slide Time: 17:20)



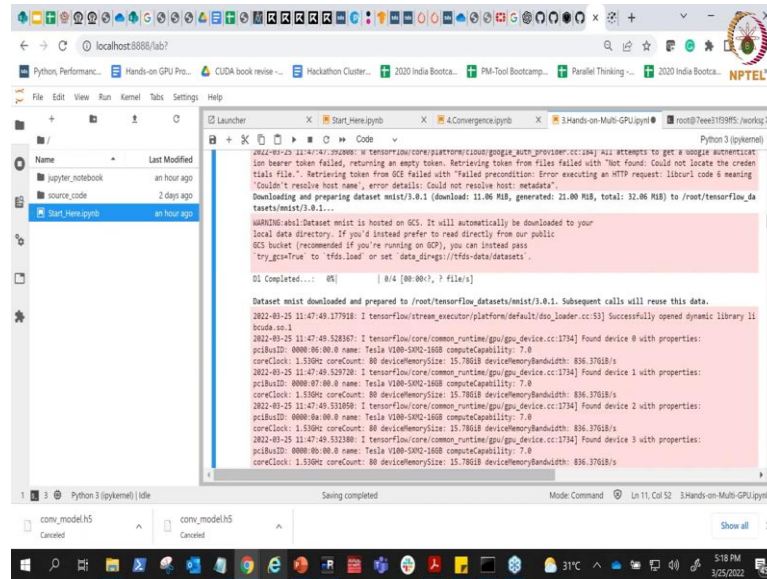
So, we are in a machine where we have 8 GPUs. So, here you can see that I have set the CUDA\_VISIBLE\_DEVICES to all of the available GPUs.

(Refer Slide Time: 17:37)



And, we are again running the fashion mnist data set.

(Refer Slide Time: 17:40)

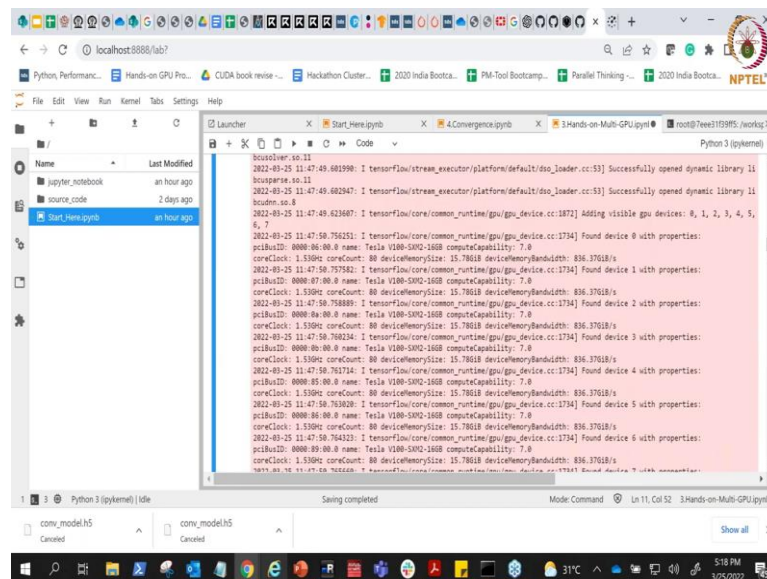


```
2022-09-25 11:47:47.292005: I tensorflow/core/platforms/google_auth_provider.cc:104 All attempts to get a Google authentication bearer token failed, returning an empty token. Retrieving token from files failed with "Not found: Could not locate the credentials file.". Retrieving token from GCE failed with "Failed precondition: Error executing an HTTP request: libcurl code 6 meaning 'Could not resolve host name', error details: Could not resolve host: metadata.".
Downloading and preparing dataset mnist/3.0.1 (download: 11.06 MiB, generated: 21.00 MiB, total: 32.06 MiB) to /root/tensorflow_datasets/mnist/3.0.1...
WARNING: This dataset is hosted on GCS. It will automatically be downloaded to your local data directory. If you'd instead prefer to read directly from our public GCS bucket (recommended if you're running on GCP), you can instead pass try_gcs=True to tfds.load or set data_dirgs://tfds-data/datasets/.
01 Completed... 0% [ 0/4 [00:00<?, ? file/s]]

Dataset mnist downloaded and prepared to /root/tensorflow_datasets/mnist/3.0.1. Subsequent calls will reuse this data.
2022-09-25 11:47:49.177918: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcudart.so.1
2022-09-25 11:47:49.528357: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 0 with properties:
pciBusID: 0000:84:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:49.529720: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 1 with properties:
pciBusID: 0000:87:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:49.531050: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 2 with properties:
pciBusID: 0000:8a:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:49.532380: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 3 with properties:
pciBusID: 0000:8b:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
```

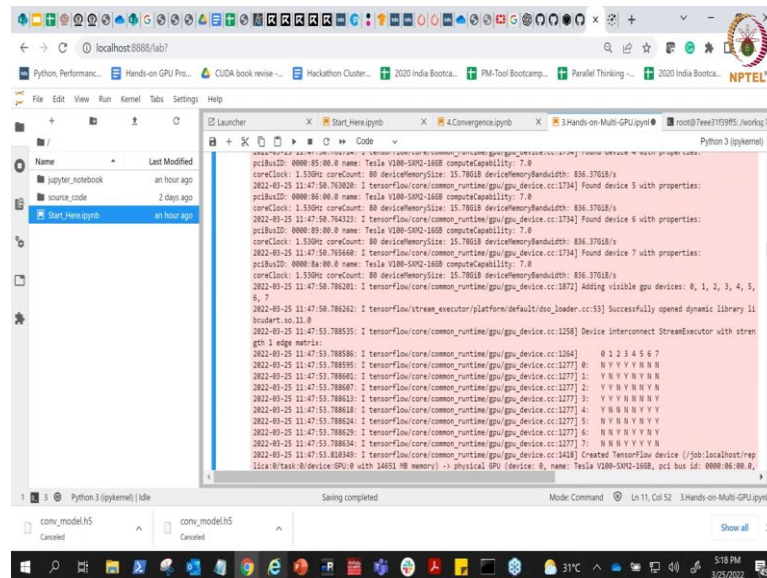
So, we do not need to worry about that part.

(Refer Slide Time: 17:41)



```
2022-09-25 11:47:49.681990: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcudart.so.1
2022-09-25 11:47:49.682947: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcudnn.so.8
2022-09-25 11:47:49.623607: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1872] Adding visible gpu devices: 0, 1, 2, 3, 4, 5, 6, 7
2022-09-25 11:47:50.756251: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 0 with properties:
pciBusID: 0000:84:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:50.757582: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 1 with properties:
pciBusID: 0000:87:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:50.758880: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 2 with properties:
pciBusID: 0000:8a:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:50.760204: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 3 with properties:
pciBusID: 0000:8b:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:50.761744: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 4 with properties:
pciBusID: 0000:85:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:50.763020: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 5 with properties:
pciBusID: 0000:86:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:50.764323: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 6 with properties:
pciBusID: 0000:88:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.53094 coreCount: 80 deviceMemorySize: 15.76818 deviceMemoryBandwidth: 836.37618/s
2022-09-25 11:47:50.765600: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 7 with properties:
```

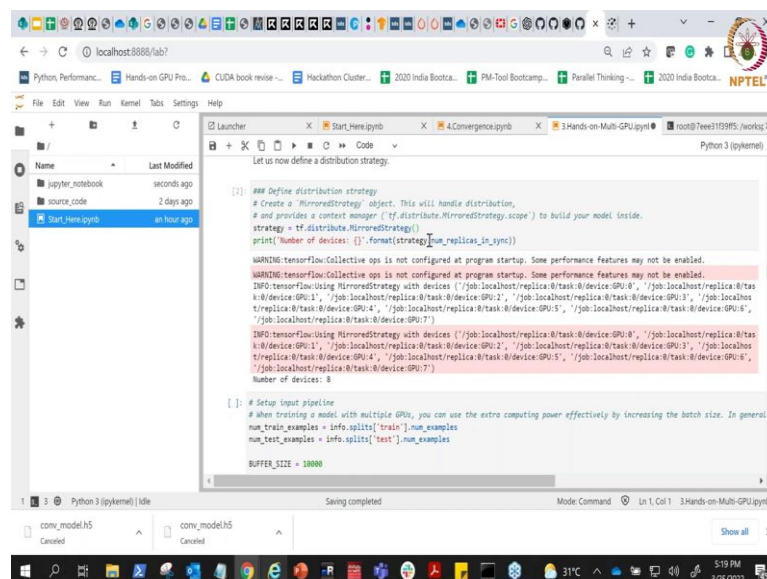
(Refer Slide Time: 17:42)



```
pciBusID: 0000:85:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.530GHz coreCount: 80 deviceMemorySize: 15.768GiB deviceMemoryBandwidth: 836.37GiB/s
2022-09-25 11:47:50.763620: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 5 with properties:
pciBusID: 0000:86:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.530GHz coreCount: 80 deviceMemorySize: 15.768GiB deviceMemoryBandwidth: 836.37GiB/s
2022-09-25 11:47:50.764323: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 6 with properties:
pciBusID: 0000:89:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.530GHz coreCount: 80 deviceMemorySize: 15.768GiB deviceMemoryBandwidth: 836.37GiB/s
2022-09-25 11:47:50.765660: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734] Found device 7 with properties:
pciBusID: 0000:8a:00.0 name: Tesla V100-SXM2-16GB computeCapability: 7.0
coreClock: 1.530GHz coreCount: 80 deviceMemorySize: 15.768GiB deviceMemoryBandwidth: 836.37GiB/s
2022-09-25 11:47:50.766261: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1872] Adding visible gpu devices: 0, 1, 2, 3, 4, 5, 6, 7
2022-09-25 11:47:50.786262: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libnvidia-rtx.so.460.39.0
2022-09-25 11:47:53.788535: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1258] Device interconnect StreamExecutor with strength 1 edge matrix:
2022-09-25 11:47:53.788586: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1264] 0 1 2 3 4 5 6 7
2022-09-25 11:47:53.788595: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1277] 0: N Y Y Y N N N N
2022-09-25 11:47:53.788601: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1277] 1: Y N Y Y N N Y N
2022-09-25 11:47:53.788613: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1277] 2: Y Y N N Y N Y N
2022-09-25 11:47:53.788618: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1277] 3: Y Y N N Y N Y N
2022-09-25 11:47:53.788624: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1277] 4: Y N N N Y Y Y Y
2022-09-25 11:47:53.788629: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1277] 5: N Y N N Y Y Y Y
2022-09-25 11:47:53.788634: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1277] 6: N Y Y Y N Y Y Y
2022-09-25 11:47:53.818349: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1418] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 14651 MB memory) -> physical GPU (device: 0, name: Tesla V100-SXM2-16GB, pci bus id: 0000:85:00.0)
```

So, it just downloads the fashion mnist data set.

(Refer Slide Time: 17:46)



```
Let us now define a distribution strategy.

[2]: ## Define distribution strategy
# Create a 'MirroredStrategy' object. This will handle distribution,
# and provides a context manager ('tf.distribute.MirroredStrategy.scope') to build your model inside.
strategy = tf.distribute.MirroredStrategy()
print('Number of devices: {}'.format(strategy.num_replicas_in_sync))

WARNING:tensorflow:Collective ops is not configured at program startup. Some performance features may not be enabled.
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/task:0/device:GPU:0', '/job:localhost/replica:0/task:0/device:GPU:1', '/job:localhost/replica:0/task:0/device:GPU:2', '/job:localhost/replica:0/task:0/device:GPU:3', '/job:localhost/replica:0/task:0/device:GPU:4', '/job:localhost/replica:0/task:0/device:GPU:5', '/job:localhost/replica:0/task:0/device:GPU:6', '/job:localhost/replica:0/task:0/device:GPU:7')
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/task:0/device:GPU:0', '/job:localhost/replica:0/task:0/device:GPU:1', '/job:localhost/replica:0/task:0/device:GPU:2', '/job:localhost/replica:0/task:0/device:GPU:3', '/job:localhost/replica:0/task:0/device:GPU:4', '/job:localhost/replica:0/task:0/device:GPU:5', '/job:localhost/replica:0/task:0/device:GPU:6', '/job:localhost/replica:0/task:0/device:GPU:7')
Number of devices: 8

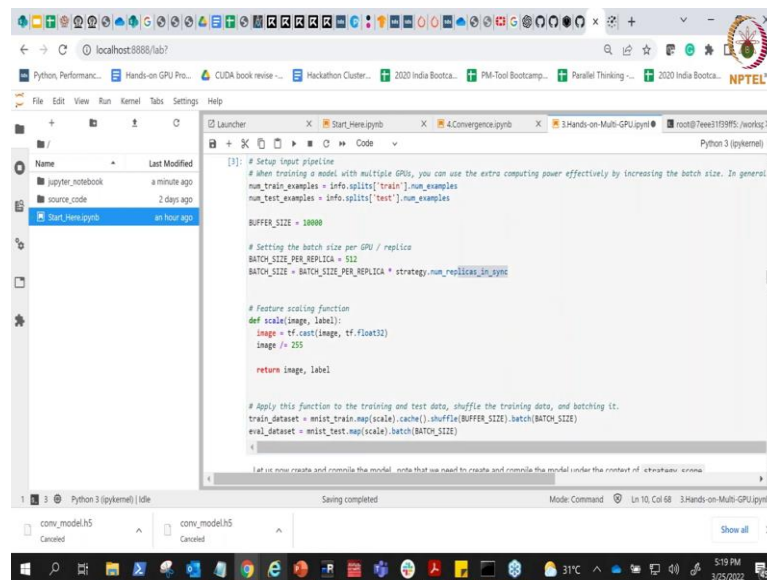
[3]: # Setup input pipeline
# When training a model with multiple GPUs, you can use the extra computing power effectively by increasing the batch size. In general
num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000
```

So, the first step that we discussed in the in the TensorFlow was to define the strategy. Here, as you can see here we are seeing tf dot distributed MirroredStrategies. We are adopting the MirroredStrategy here. And, you can see what it will do is that once you run it, it will try to you can use number of replicas in sync. You would what all GPUs devices, how many of them are visible to it can be seen. By this you can see here it can see almost a all of the devices which is the 8 devices which are available.



(Refer Slide Time: 18:22)



```
[3]: # Setup input pipeline
# when training a model with multiple GPUs, you can use the extra computing power effectively by increasing the batch size. In general
num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

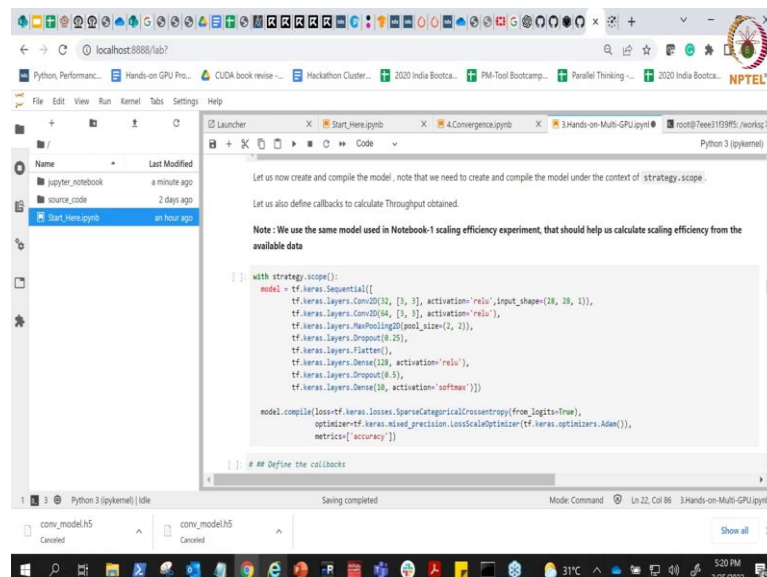
# Setting the batch size per GPU / replica
BATCH_SIZE_PER_REPLICA = 32
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

# Feature scaling function
def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255
    return image, label

# Apply this function to the training and test data, shuffle the training data, and batching it.
train_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

After this, let us run the code again. So, this particular code is basically a code to just do some additional task of doing defining the batch size. For each and every replica you are going to have your batch size and then you are going to basically just give it. So, there is nothing so, great about this code.

(Refer Slide Time: 18:43)



```
Let us now create and compile the model, note that we need to create and compile the model under the context of 'strategy.scope'.
Let us also define callbacks to calculate throughput obtained.

Note: We use the same model used in Notebook-1 scaling efficiency experiment, that should help us calculate scaling efficiency from the
available data

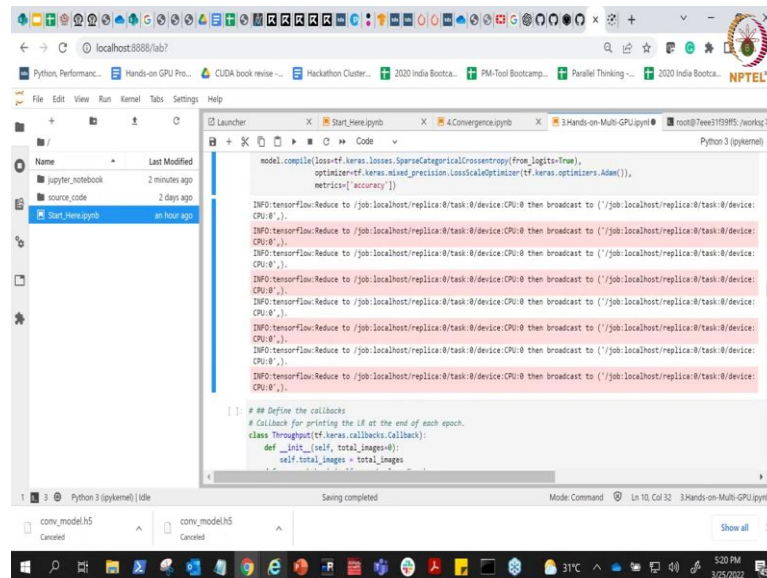
with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, [3, 3], activation='relu', input_shape=(28, 28, 1)),
        tf.keras.layers.Conv2D(64, [3, 3], activation='relu'),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Dropout(0.25),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(10, activation='softmax')])

    model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=['accuracy'])

## Define the callbacks
```

You can just look at the code and just, here we are trying to shuffle the data for the training and the batch set.

(Refer Slide Time: 18:53)



```
model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              optimizer=tf.keras.mixed_precision.LossScaleOptimizer(tf.keras.optimizers.Adam()),
              metrics=['accuracy'])

INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to (/job:localhost/replica:0/task:0/device:CPU:0).

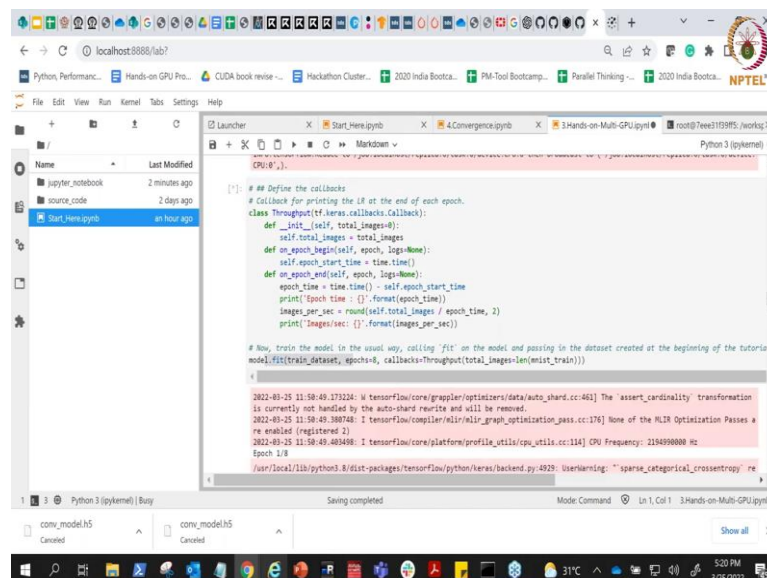
[] ## Define the callbacks
# Callback for printing the LR at the end of each epoch.
class Throughput(tf.keras.callbacks.Callback):
    def __init__(self, total_images=0):
        self.total_images = total_images

    def on_epoch_end(self, epoch, logs=None):
        self.epoch_start_time = time.time()
        def on_epoch_end(self, epoch, logs=None):
            epoch_time = time.time() - self.epoch_start_time
            print('Epoch time: {}'.format(epoch_time))
            images_per_sec = round(self.total_images / epoch_time, 2)
            print('Images/sec: {}'.format(images_per_sec))

# Now, train the model in the usual way, calling 'fit' on the model and passing in the dataset created at the beginning of the tutorial
model.fit(train_dataset, epochs=8, callbacks=[Throughput(total_images=len(train_train))])
```

After you are done with this, you would see that we have to define the scope and you can see here within the scope here we are creating the model. So, the model is a traditional convolution neural network model. And, the main idea here is to get it inside the scope and then within you can define your hyper parameters like what symmetric and what is the optimizer and all those things here. So, here we are using the Adam optimizer in this particular case.

(Refer Slide Time: 19:25)



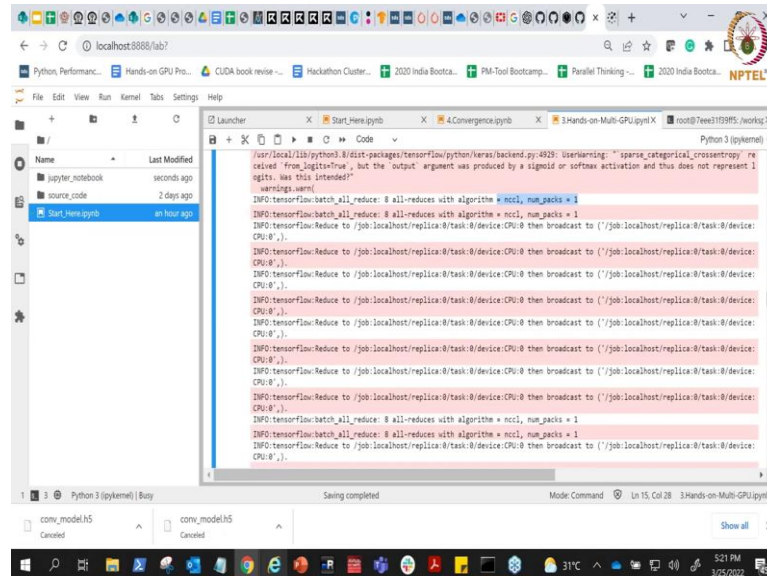
```
## Define the callbacks
# Callback for printing the LR at the end of each epoch.
class Throughput(tf.keras.callbacks.Callback):
    def __init__(self, total_images=0):
        self.total_images = total_images

    def on_epoch_end(self, epoch, logs=None):
        self.epoch_start_time = time.time()
        def on_epoch_end(self, epoch, logs=None):
            epoch_time = time.time() - self.epoch_start_time
            print('Epoch time: {}'.format(epoch_time))
            images_per_sec = round(self.total_images / epoch_time, 2)
            print('Images/sec: {}'.format(images_per_sec))

# Now, train the model in the usual way, calling 'fit' on the model and passing in the dataset created at the beginning of the tutorial
model.fit(train_dataset, epochs=8, callbacks=[Throughput(total_images=len(train_train))])
```

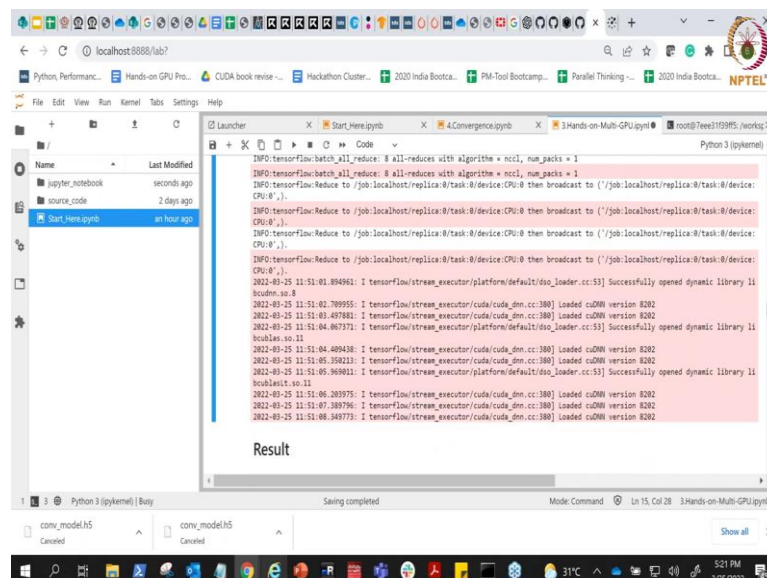
Once you are done with that, you can basically call the model dot fit function and it is ideally supposed to run across all of the GPUs.

(Refer Slide Time: 19:34)



So, again you can see here it is using nccl. So, it will still use nccl as we have shown you last time.

(Refer Slide Time: 19:43)





(Refer Slide Time: 19:53)

The screenshot shows a Windows desktop environment. At the top, there is a taskbar with various application icons. Below it, a web browser window is open at the URL `localhost:8886/lab/`. The browser displays a Jupyter Notebook interface. The notebook has several tabs open, with the active tab being `Start_Here.ipynb`. The notebook content shows the output of a TensorFlow training script, including messages about dynamic library opening and training progress. The output is as follows:

```

2022-09-25 11:51:04.007371: I tensorflow/stream_executor/platform/default/dyn_loader.cc:53] Successfully opened dynamic library ...
Epoch 1/8
2022-09-25 11:51:04.488438: I tensorflow/stream_executor/cuda/cuda_dnn.cc:380] Loaded cudaNN version 8202
2022-09-25 11:51:05.350211: I tensorflow/stream_executor/cuda/cuda_dnn.cc:380] Loaded cudaNN version 8202
2022-09-25 11:51:05.950001: I tensorflow/stream_executor/platform/default/dyn_loader.cc:53] Successfully opened dynamic library ...
localhost:8886 11
2022-09-25 11:51:06.203975: I tensorflow/stream_executor/cuda/cuda_dnn.cc:380] Loaded cudaNN version 8202
2022-09-25 11:51:07.389796: I tensorflow/stream_executor/cuda/cuda_dnn.cc:380] Loaded cudaNN version 8202
2022-09-25 11:51:08.348773: I tensorflow/stream_executor/cuda/cuda_dnn.cc:380] Loaded cudaNN version 8202
2022-09-25 11:51:09.314017: I tensorflow/stream_executor/cuda/cuda_dnn.cc:380] Loaded cudaNN version 8202
15/15 [=====] - 254 72ms/step - loss: 1.1400 - accuracy: 0.6525
Epoch time : 15.48602368387534s
Images/sec: 2353.82
Epoch 2/8
15/15 [=====] - 0s 18ms/step - loss: 0.3960 - accuracy: 0.8796
Epoch time : 0.2964292236328125s
Images/sec: 282136.43
Epoch 3/8
15/15 [=====] - 0s 17ms/step - loss: 0.2554 - accuracy: 0.9238
Epoch time : 0.2785012722615381s
Images/sec: 221438.87
Epoch 4/8
15/15 [=====] - 0s 17ms/step - loss: 0.1832 - accuracy: 0.9468
Epoch time : 0.27874517444959s
Images/sec: 225256.36
Epoch 5/8
15/15 [=====] - 0s 19ms/step - loss: 0.1384 - accuracy: 0.9599
Epoch time : 0.395408239364624s

```

At the bottom of the Jupyter Notebook interface, there is a terminal window showing the command `python3 3_Hands-on-Multi-GPU.ipynb` being executed. The terminal output shows the command being run and the resulting output.

And, it will start the job and.

(Refer Slide Time: 20:01)

The screenshot shows a Jupyter Notebook interface with a terminal window open. The terminal displays the output of a command to show NVLink connections between GPUs. The output is a table with columns for GPU ID, ID, and Usage. The table lists 16 GPUs (GPU0 to GPU15) and their connections. Below the table is a legend explaining the symbols used.

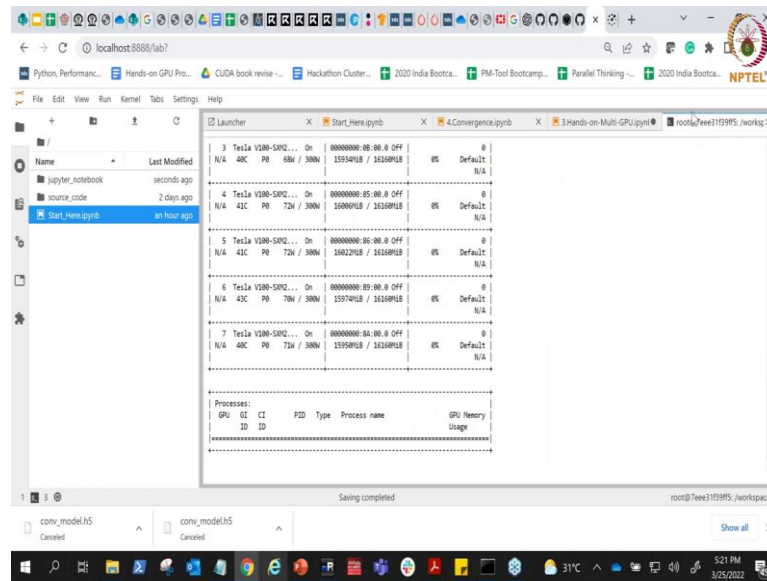
GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	mls.0	mls.1	mls.2	mls.3	CPU Affinity	NVL Affinity
GPU0	X	NL1	NL1	NL2	NL2	SYS	SYS	P2X	PMB	SYS	SYS	0-13,48-59	0
GPU1	NL1	X	NL2	SYS	NL2	SYS	SYS	P2X	PMB	SYS	SYS	0-13,48-59	0
GPU2	NL1	NL2	X	NL2	SYS	NL1	SYS	PMB	P2X	SYS	SYS	0-13,48-59	0
GPU3	NL2	NL1	NL2	X	SYS	SYS	SYS	PMB	SYS	SYS	SYS	0-13,48-59	0
GPU4	NL2	SYS	SYS	X	NL1	NL1	NL2	SYS	P2X	PMB	SYS	20-39,60-79	1
GPU5	SYS	NL2	SYS	SYS	NL1	X	NL2	NL1	SYS	P2X	PMB	20-39,60-79	1
GPU6	SYS	SYS	NL1	SYS	NL2	NL2	X	NL2	SYS	PMB	P2X	20-39,60-79	1
GPU7	SYS	SYS	SYS	NL2	NL2	NL1	NL2	X	SYS	SYS	PMB	20-39,60-79	1
mls.0	P2X	PMB	SYS	SYS	SYS	SYS	SYS	X	PMB	SYS	SYS		
mls.1	PMB	P2X	P2X	SYS	SYS	SYS	PMB	X	SYS	SYS			
mls.2	SYS	SYS	P2X	P2X	P2X	PMB	PMB	SYS	SYS	X	PMB		
mls.3	SYS	SYS	SYS	PMB	PMB	P2X	P2X	SYS	PMB	X			

Legend:

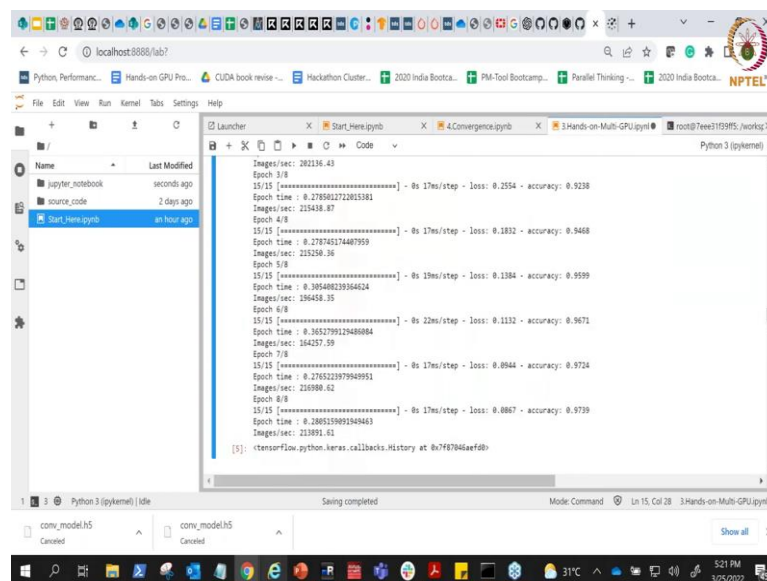
- X = Self
- NL1 = Connection traversing PCIe as well as the SMP interconnect between NVLink nodes (e.g., CPU/GPU)
- NL2 = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NVLink node
- PMB = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
- P2X = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
- P2X = Connection traversing at most a single PCIe bridge
- NM = Connection traversing a bonded set of # NVLinks

root@Tree139BFS: /workspace

(Refer Slide Time: 20:06)

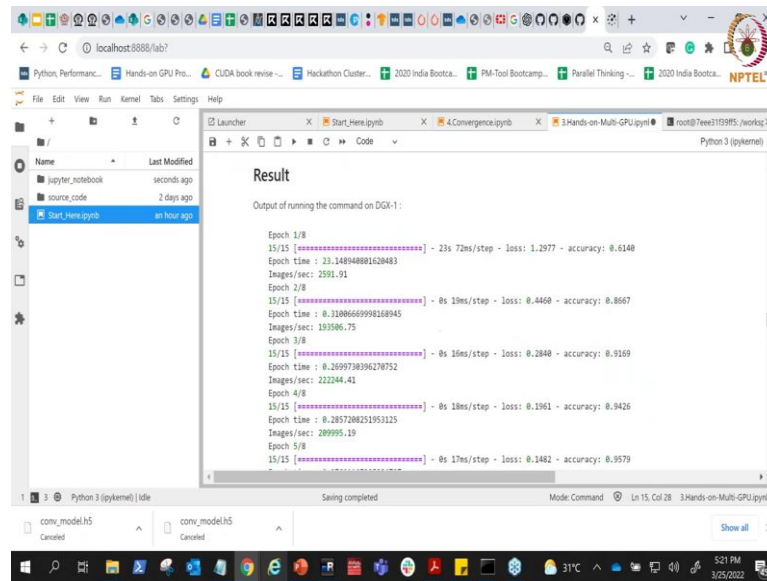


(Refer Slide Time: 20:13)



So, you can see here it has already finished and the images per second is almost 213891.

(Refer Slide Time: 20:23)

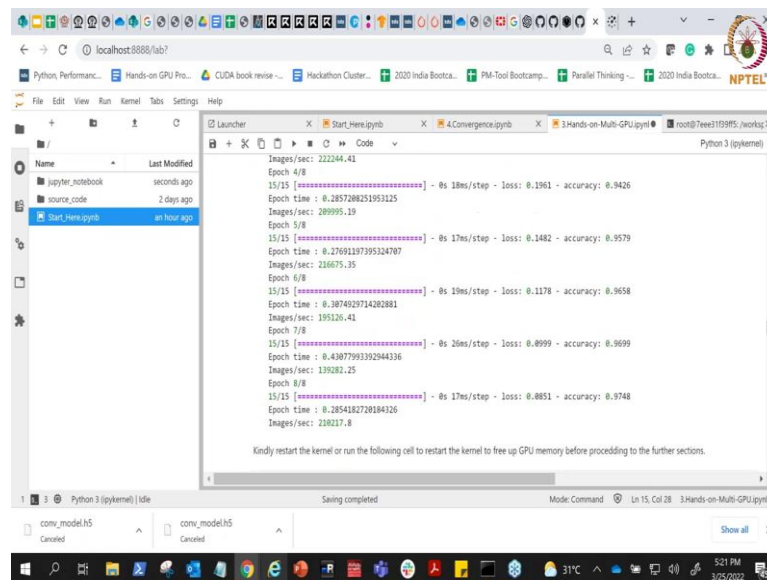


The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code editor contains a cell with the following output:

```
Output of running the command on DGX-1:  
  
Epoch 1/8  
15/15 [#####] - 23s 72ms/step - loss: 1.2977 - accuracy: 0.6148  
Epoch time : 23.148948891628483  
Images/sec: 2591.91  
Epoch 2/8  
15/15 [#####] - 0s 19ms/step - loss: 0.4468 - accuracy: 0.8667  
Epoch time : 0.3108669998168945  
Images/sec: 193506.75  
Epoch 3/8  
15/15 [#####] - 0s 16ms/step - loss: 0.2848 - accuracy: 0.9269  
Epoch time : 0.260973836278752  
Images/sec: 222244.41  
Epoch 4/8  
15/15 [#####] - 0s 18ms/step - loss: 0.1961 - accuracy: 0.9426  
Epoch time : 0.285728251953125  
Images/sec: 289995.19  
Epoch 5/8  
15/15 [#####] - 0s 17ms/step - loss: 0.1482 - accuracy: 0.9579
```

So, it was quite easy to run it on a TensorFlow environment, practically a code does not change so much.

(Refer Slide Time: 20:29)

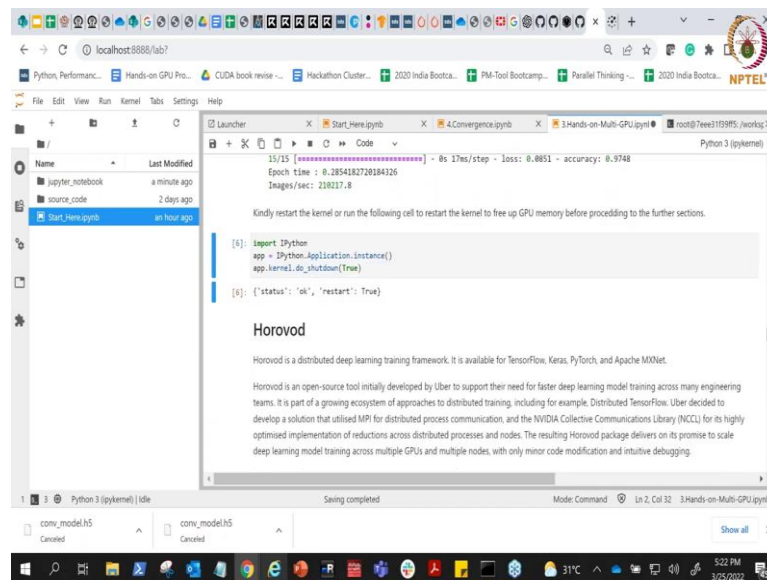


The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code editor contains a cell with the following output:

```
Images/sec: 222244.41  
Epoch 4/8  
15/15 [#####] - 0s 18ms/step - loss: 0.1961 - accuracy: 0.9426  
Epoch time : 0.285728251953125  
Images/sec: 289995.19  
Epoch 5/8  
15/15 [#####] - 0s 17ms/step - loss: 0.1482 - accuracy: 0.9579  
Epoch time : 0.27691197395324707  
Images/sec: 216675.35  
Epoch 6/8  
15/15 [#####] - 0s 19ms/step - loss: 0.1178 - accuracy: 0.9658  
Epoch time : 0.3874929714262881  
Images/sec: 195126.41  
Epoch 7/8  
15/15 [#####] - 0s 20ms/step - loss: 0.8999 - accuracy: 0.9699  
Epoch time : 0.43877993392944336  
Images/sec: 139282.25  
Epoch 8/8  
15/15 [#####] - 0s 17ms/step - loss: 0.8851 - accuracy: 0.9748  
Epoch time : 0.2854182728184326  
Images/sec: 228237.8  
  
Kindly restart the kernel or run the following cell to restart the kernel to free up GPU memory before proceeding to the further sections.
```

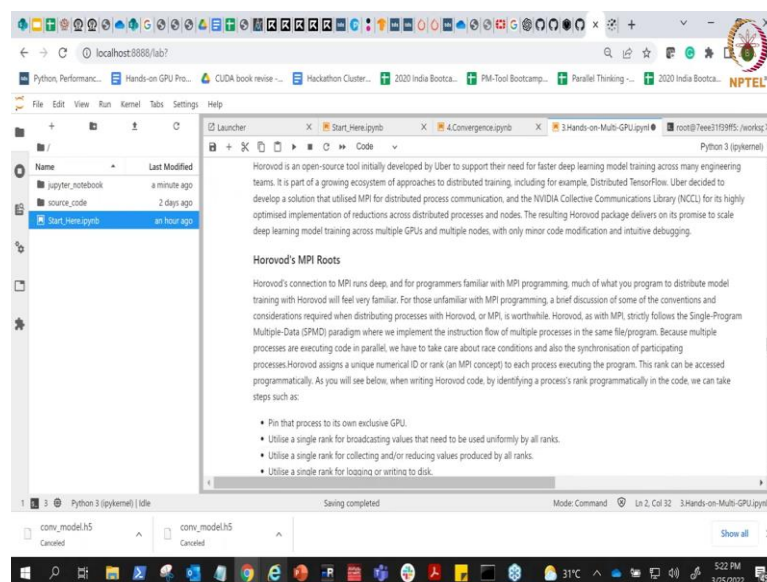
I am going to just run this particular cell.

(Refer Slide Time: 20:31)



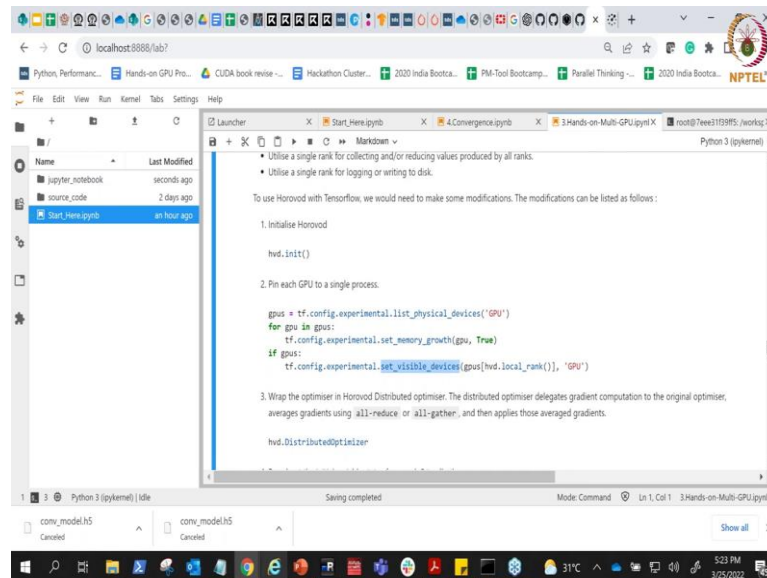
Because, TensorFlow kind of does not read the allocation and it will not allow us to run the Horovod simulation. So, I am just shutting down the kernel, just to make sure we have freed up all these space.

(Refer Slide Time: 20:47)



So, so this was a very quick demo of how to use TensorFlow. Now, we are going to move towards Horovod. So, Horovod again I have told you that it takes its motivation from MPI which has been the most scalable distributed framework, that we have seen in the high performance computing domains.

(Refer Slide Time: 21:09)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The file explorer shows a directory structure with files like 'jupyter\_notebook', 'source\_code', and 'Start\_Here.py'. The code editor displays a Python script for configuring Horovod. The script includes comments and code for initializing Horovod, pinning GPUs to processes, and wrapping the optimizer in Horovod's Distributed Optimizer.

```
• Utilise a single rank for collecting and/or reducing values produced by all ranks.
• Utilise a single rank for logging or writing to disk.

To use Horovod with Tensorflow, we would need to make some modifications. The modifications can be listed as follows:

1. Initialise Horovod
hvd.init()

2. Pin each GPU to a single process.

gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

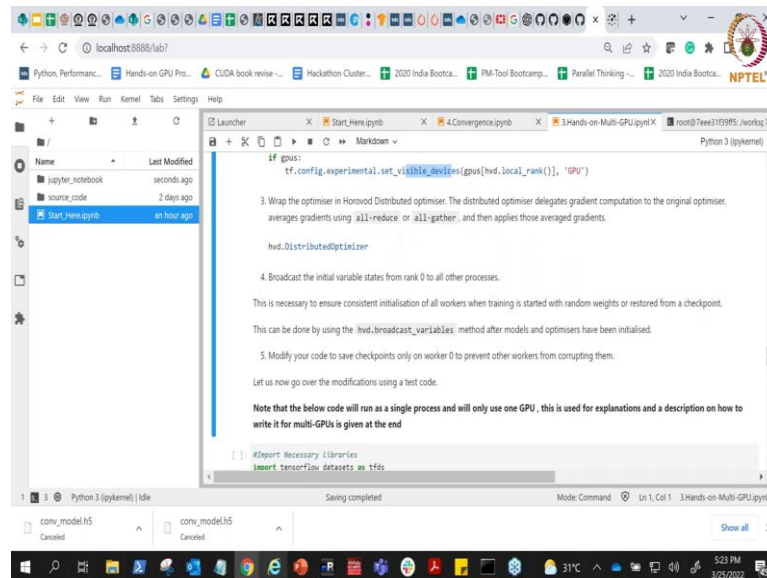
3. Wrap the optimiser in Horovod Distributed optimiser. The distributed optimiser delegates gradient computation to the original optimiser,
averages gradients using 'all-reduce' or 'all-gather', and then applies those averaged gradients.

hvd.DistributedOptimizer
```

But, it basically has to make sure that it kind of adheres to certain principles like pin the process to its own exclusive GPU. So, every process is going to pin itself to a particular GPU. It will utilize a single rank for broadcasting values that needs to be used uniformly. So, a single rank needs to be used for broadcasting the values. Utilize a single rank for collecting the values and reducing, because without this there would be a problem in terms of the accuracy.

So, utilize a single rank for also logging and writing to the disk. So, generally this is done by your a primary process or the rank 0. And, I will show you what do I mean by that right. So, you can see here you are first doing the initialization. After the initialization basically, the second thing which we told is you have to set the set visible device.

(Refer Slide Time: 22:07)



The screenshot shows a Jupyter Notebook with a file explorer on the left and a code editor on the right. The code editor displays the following content:

```
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

3. Wrap the optimiser in Horovod Distributed optimiser. The distributed optimiser delegates gradient computation to the original optimiser,
   averages gradients using 'all-reduce' or 'all-gather', and then applies those averaged gradients.

hvd.DistributedOptimizer

4. Broadcast the initial variable states from rank 0 to all other processes.

This is necessary to ensure consistent initialisation of all workers when training is started with random weights or restored from a checkpoint.
This can be done by using the hvd.broadcast_variables method after models and optimisers have been initialised.

5. Modify your code to save checkpoints only on worker 0 to prevent other workers from corrupting them.

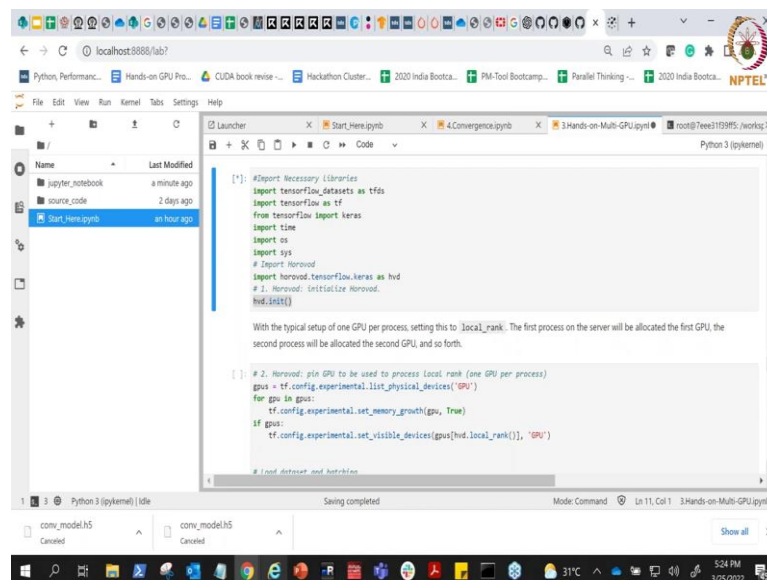
Let us now go over the modifications using a test code.

Note that the below code will run as a single process and will only use one GPU, this is used for explanations and a description on how to
write it for multi-GPUs is given at the end

[ ]: # Import Necessary Libraries
import tensorflow_datasets as tfds
```

And, it is based on the rank. So, the rank 0 will set the g will use the GPU 0, rank 1 will use GPU 1, that is how it uses. And, then you will define your own distributed, you will wrap the optimizer onto distributed optimizer.

(Refer Slide Time: 22:26)



The screenshot shows a Jupyter Notebook with a file explorer on the left and a code editor on the right. The code editor displays the following content:

```
[ ]: # Import Necessary Libraries
import tensorflow_datasets as tfds
import tensorflow as tf
from tensorflow import keras
import time
import os
import sys
# Import Horovod
import horovod.tensorflow.keras as hvd
# 1. Horovod: initialize Horovod.
hvd.init()

With the typical setup of one GPU per process, setting this to hvd.local_rank(). The first process on the server will be allocated the first GPU, the
second process will be allocated the second GPU, and so forth.

[ ]: # 2. Horovod: pin GPU to be used to process local_rank (one GPU per process)
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

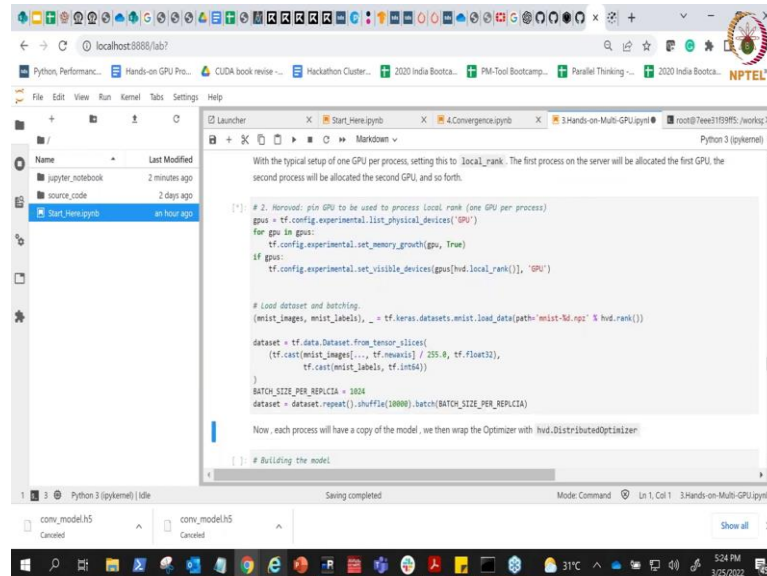
# Load dataset and horovod
```

So, let us look at the actual code here. So, again you can see we are using Horovod, but we are using within horovods still tensorflow. As I said that Horovod is basically a higher level framework. It is it can work on different backends like PyTorch and other methods also. Here, we are still using horovod, but over tensorflow. But, if you write



code in Horovod, it can use any of the back ends. So, let us use initialize, the first thing that we are doing here is to initialize the Horovod. So, till the time it is running anyhow.

(Refer Slide Time: 23:14)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code editor contains the following Python code:

```
With the typical setup of one GPU per process, setting this to 'local_rank'. The first process on the server will be allocated the first GPU, the second process will be allocated the second GPU, and so forth.

[ ]: # 2. Horovod: pin GPU to be used to process local rank (one GPU per process)
    gpus = tf.config.experimental.list_physical_devices('GPU')
    for gpu in gpus:
        tf.config.experimental.set_memory_growth(gpu, True)
    if gpus:
        tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

# Load dataset and batching
(mnist_images, mnist_labels), _ = tf.keras.datasets.mnist.load_data(path='mnist-td.rgs' % hvd.rank())

dataset = tf.data.Dataset.from_tensor_slices(
    (tf.cast(mnist_images..., tf.float32) / 255.0, tf.float32),
    tf.cast(mnist_labels, tf.int64))

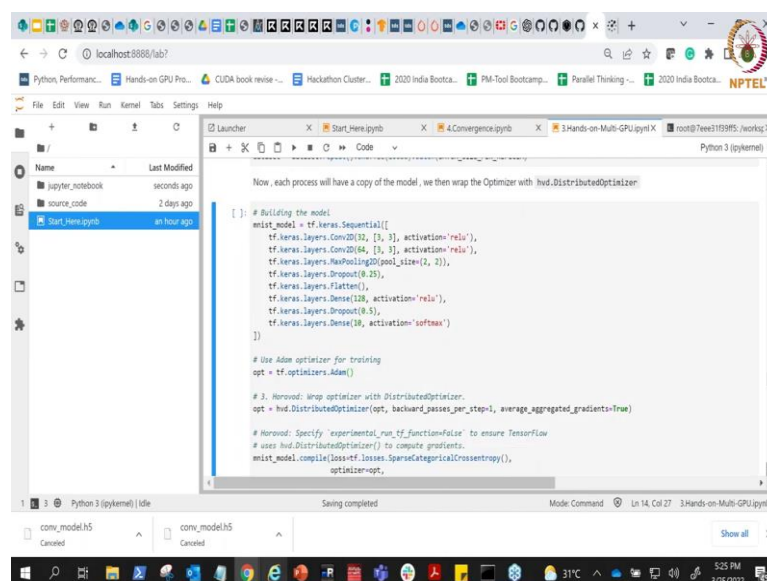
BATCH_SIZE_PER_REPLICA = 100
dataset = dataset.repeat().shuffle(10000).batch(BATCH_SIZE_PER_REPLICA)

Now, each process will have a copy of the model, we then wrap the Optimizer with hvd.DistributedOptimizer

[ ]: # Building the model
```

The second thing which we told was to set the visible device. This is very very critical, otherwise you will all end up using one single GPU. And, we are doing the same thing, we are creating the data sets and creating a replica of it to. So, let it run, it is taking an certain amount of time.

(Refer Slide Time: 23:35)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code editor contains the following Python code:

```
Now, each process will have a copy of the model, we then wrap the Optimizer with hvd.DistributedOptimizer

[ ]: # Building the model
    mnist_model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, [3, 3], activation='relu'),
        tf.keras.layers.Conv2D(64, [3, 3], activation='relu'),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Dropout(0.25),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

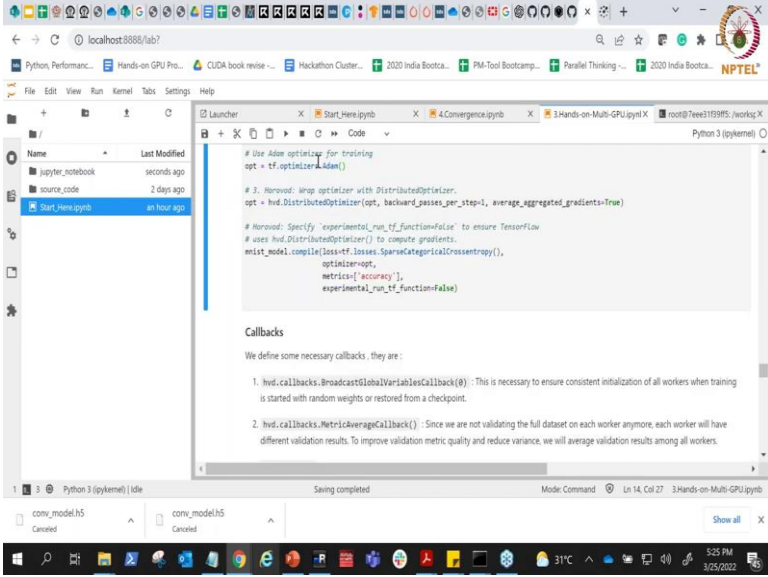
# Use Adam optimizer for training
opt = tf.optimizers.Adam()

# 3. Horovod: Wrap optimizer with DistributedOptimizer.
opt = hvd.DistributedOptimizer(opt, backward_passes_per_step=1, average_aggregated_gradients=True)

# Horovod: Specify 'experimental_run_tf_function=False' to ensure TensorFlow
# uses hvd.DistributedOptimizer to compute gradients
mnist_model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
                    optimizer=opt,
```

After you are done with this, as you can see you are creating your normal model which is in this particular case we are building the model with couple of layers convolution, MaxPooling, Dropout. And, then we are also defining a normal optimizer, here in this case we are using the Adam optimizer for doing this.

(Refer Slide Time: 24:04)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code editor displays the following Python code:

```
# Use Adam optimizer for training
opt = tf.optimizer.Adam()

# 3. Horovod: Wrap optimizer with DistributedOptimizer
opt = hvd.DistributedOptimizer(opt, backward_passes_per_step=1, average_aggregated_gradients=True)

# Horovod: Specify 'experimental_run_tf_function=False' to ensure TensorFlow
# uses hvd.DistributedOptimizer() to compute gradients.
mist_model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
                  optimizer=opt,
                  metrics=['accuracy'],
                  experimental_run_tf_function=False)
```

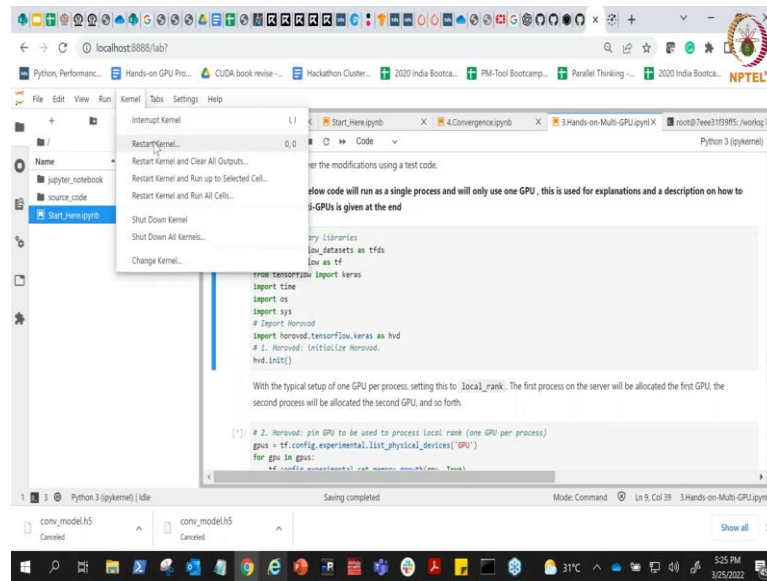
Below the code, there is a section titled "Callbacks" with the text: "We define some necessary callbacks, they are:" followed by a list of two callbacks:

1. `hvd.callbacks.BroadcastGlobalVariablesCallback(0)` : This is necessary to ensure consistent initialization of all workers when training is started with random weights or restored from a checkpoint.
2. `hvd.callbacks.MetricAverageCallback()` : Since we are not validating the full dataset on each worker anymore, each worker will have different validation results. To improve validation metric quality and reduce variance, we will average validation results among all workers.

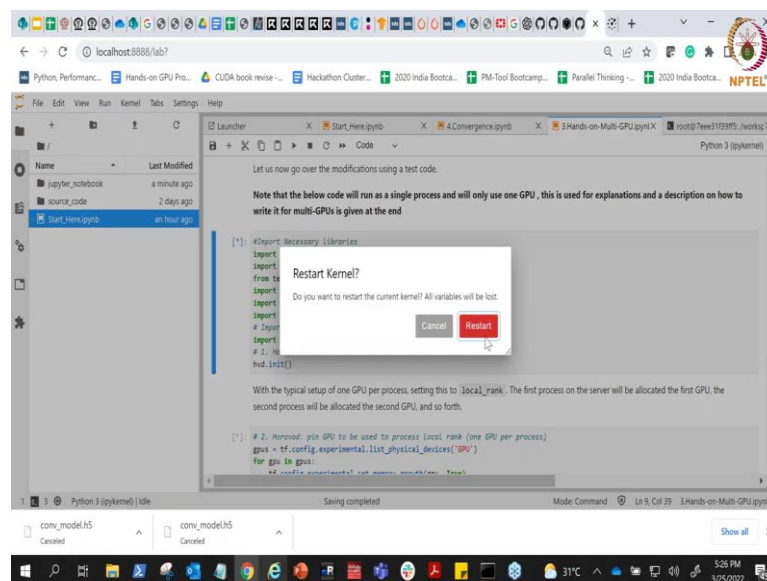
After we are done with the Adam optimizer, we are doing Horovod dot DistributedOptimizer. So, we are wrapping the existing optimizer within the DistributedOptimizer and it takes certain flags like how many backward passes you need per step or average aggregated gradients. Do you want average gradients or not and we have set this to True obviously. And, then we compile the overall code. Let me check, it is still not finished.



(Refer Slide Time: 24:40)

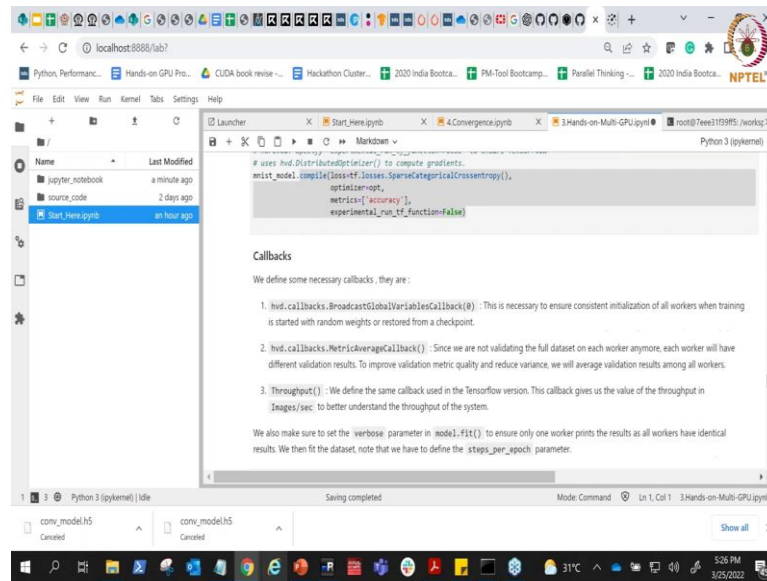


(Refer Slide Time: 24:42)

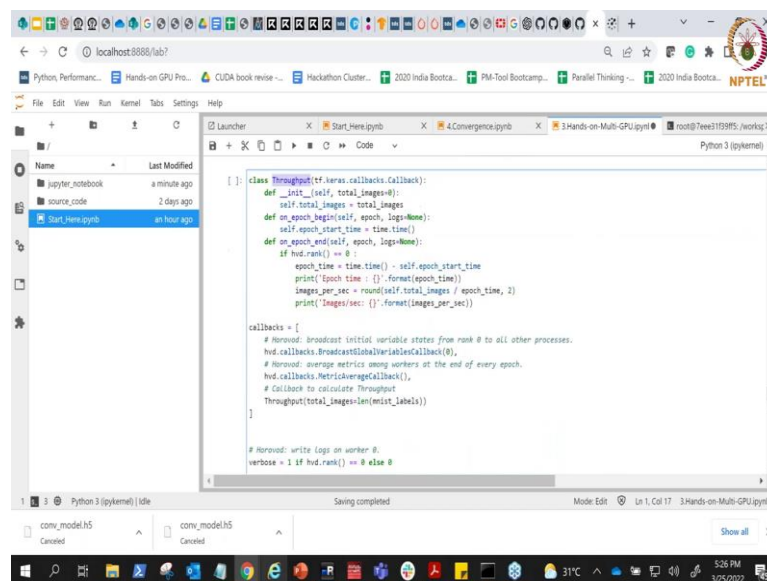


Let me just restart the kernel, after you are done with this.

(Refer Slide Time: 24:54)

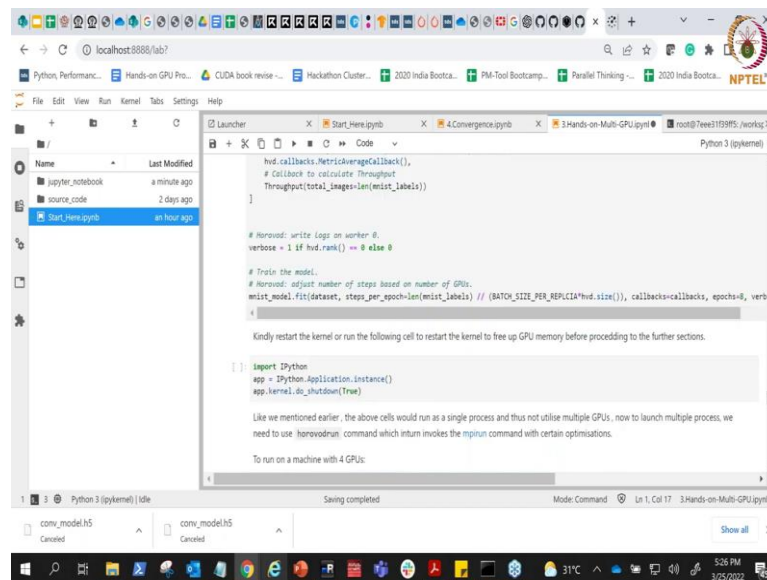


(Refer Slide Time: 24:55)



After that I think we have just written some wrapper to figure out how what is how many images per second, we are able to calculate using the throughput.

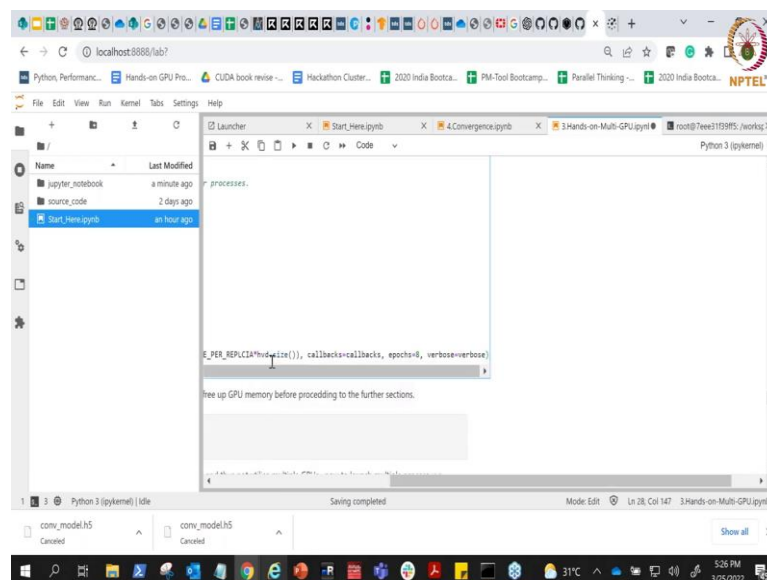
(Refer Slide Time: 25:05)



```
hvd.callbacks.MetricAverageCallback(),  
# Callback to calculate throughput  
Throughput(total_images=len(mnist_labels))  
]  
  
# Horovod: write logs on worker 0.  
verbose = 1 if hvd.rank() == 0 else 0  
  
# Train the model.  
# Horovod: adjust number of steps based on number of GPUs.  
mnist_model.fit(dataset, steps_per_epoch=len(mnist_labels) // (BATCH_SIZE * hvd.size()), callbacks=callbacks, epochs=8, verb  
  
Kindly restart the kernel or run the following cell to restart the kernel to free up GPU memory before proceeding to the further sections.  
  
import IPython  
app = IPython.Application.instance()  
app.kernel.do_shutdown(True)  
  
Like we mentioned earlier, the above cells would run as a single process and thus not utilize multiple GPUs, now to launch multiple process, we  
need to use horovodrun command which invokes the mpirun command with certain optimisations.  
  
To run on a machine with 4 GPUs:
```

And, the callback that we are going to provide it with. So, this is the callback that we are creating to calculate the how many images per second, we are able to work on.

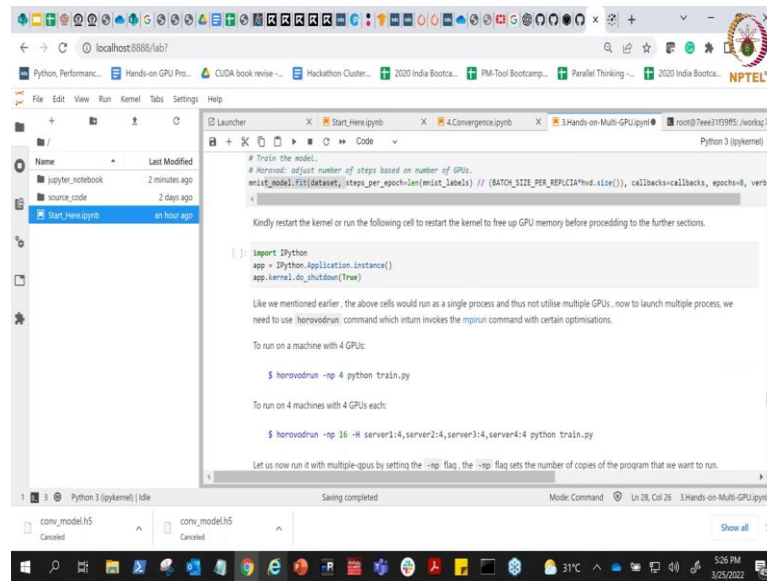
(Refer Slide Time: 25:19)



```
processes.  
  
BATCH_SIZE * hvd.size()), callbacks=callbacks, epochs=8, verbose=verbose)  
  
Free up GPU memory before proceeding to the further sections.
```

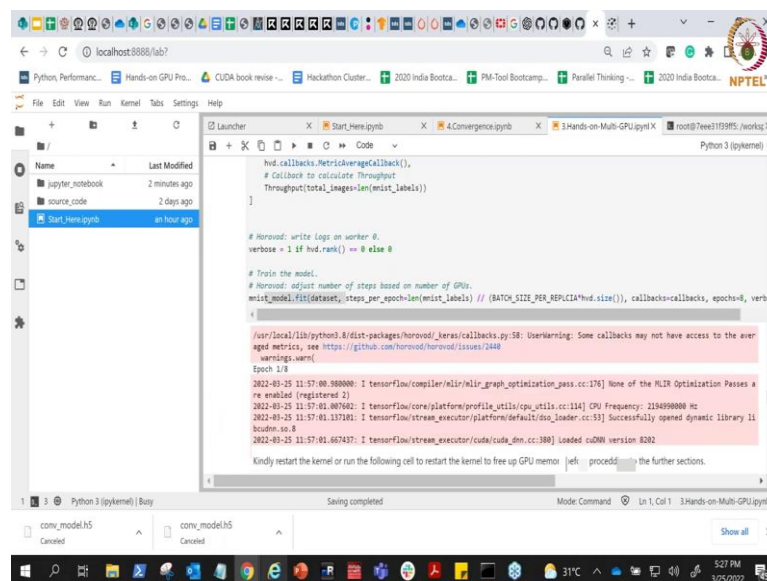
And in the end yes again, you would do model dot fit. So, you say model dot fit and it you basically provide it with the same thing. So, you basically are saying this.

(Refer Slide Time: 25:31)



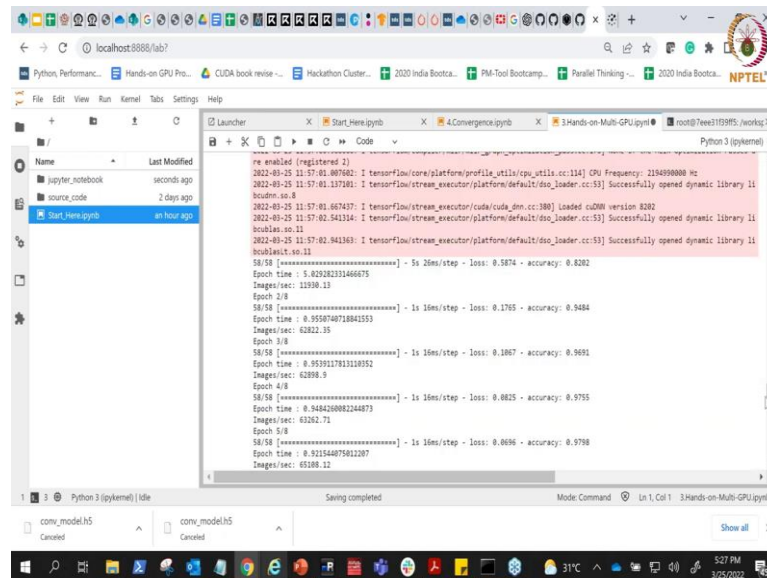
So, let us see if anything has happened.

(Refer Slide Time: 25:43)



It seems to having some problem to run output ok. So, it started working now great. I do not know what is happening.

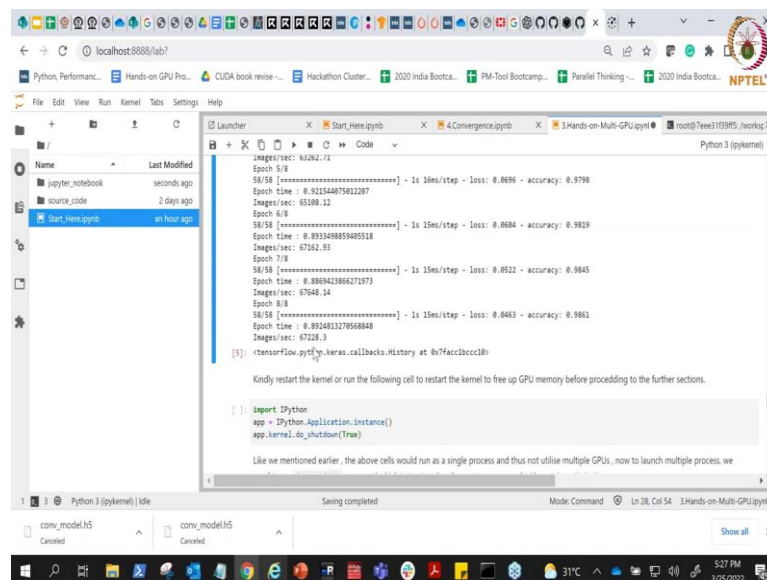
(Refer Slide Time: 25:54)



```
re enabled (registered 2)
2022-09-25 11:57:01.867602: I tensorflow/core/platform/profile_utils/cpu_utils.cc:114] CPU Frequency: 2154990000 Hz
2022-09-25 11:57:01.137101: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library ll
boudm.so.8
2022-09-25 11:57:01.667477: I tensorflow/stream_executor/cuda/cuda_dnn.cc:390] Loaded cuDNN version 8202
2022-09-25 11:57:02.541314: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library ll
boudm.so.11
2022-09-25 11:57:02.941361: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library ll
boudm.so.11
58/58 [=====] - 1s 26ms/step - loss: 0.5874 - accuracy: 0.8202
Epoch time : 5.8225292331466675
Images/sec: 13108.13
Epoch 2/8
58/58 [=====] - 1s 16ms/step - loss: 0.1765 - accuracy: 0.9484
Epoch time : 0.9550704718841553
Images/sec: 62822.35
Epoch 3/8
58/58 [=====] - 1s 16ms/step - loss: 0.1067 - accuracy: 0.9691
Epoch time : 0.953911781110352
Images/sec: 62898.9
Epoch 4/8
58/58 [=====] - 1s 16ms/step - loss: 0.0825 - accuracy: 0.9755
Epoch time : 0.948426082244873
Images/sec: 63262.71
Epoch 5/8
58/58 [=====] - 1s 16ms/step - loss: 0.0696 - accuracy: 0.9798
Epoch time : 0.921544075612207
Images/sec: 65108.12
```

So, there is some refresh problem actually, it is not the problem of a this thing. So, I hope the code was kind of clear with respect to a model dot fit here. This is going to run basically a one optimizer, here is one process only.

(Refer Slide Time: 26:10)



```
Images/sec: 63262.71
Epoch 5/8
58/58 [=====] - 1s 16ms/step - loss: 0.0696 - accuracy: 0.9798
Epoch time : 0.921544075612207
Images/sec: 65108.12
Epoch 6/8
58/58 [=====] - 1s 15ms/step - loss: 0.0604 - accuracy: 0.9819
Epoch time : 0.893349859405518
Images/sec: 67362.93
Epoch 7/8
58/58 [=====] - 1s 15ms/step - loss: 0.0522 - accuracy: 0.9845
Epoch time : 0.880423866271973
Images/sec: 67546.14
Epoch 8/8
58/58 [=====] - 1s 15ms/step - loss: 0.0463 - accuracy: 0.9861
Epoch time : 0.8934612179568048
Images/sec: 67228.3
[5]: <tensorflow.python.keras.callbacks.History at 0x7fac1bccc1b>

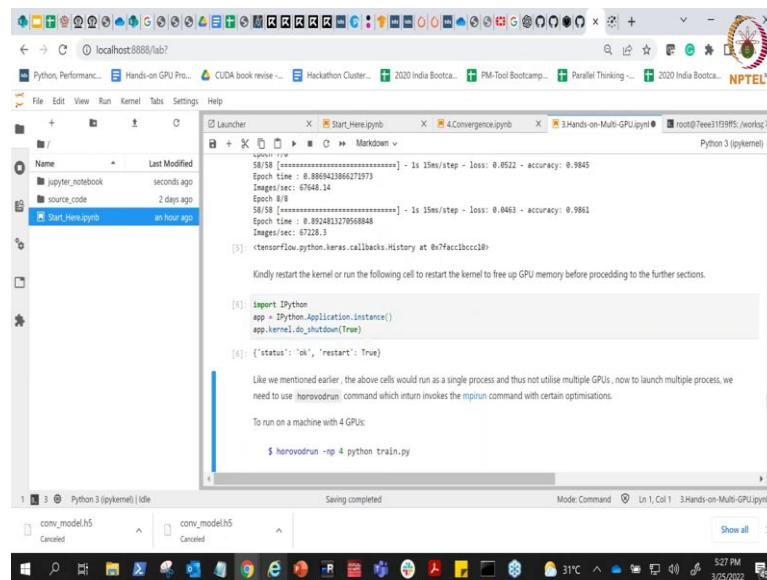
Kindly restart the kernel or run the following cell to restart the kernel to free up GPU memory before proceeding to the further sections.

import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)

Like we mentioned earlier, the above cells would run as a single process and thus not utilize multiple GPUs, now to launch multiple process, we
```

And, you can see it is like 67000 something right.

(Refer Slide Time: 26:15)



The screenshot shows the JupyterLab interface. The left sidebar displays a file explorer with 'jupyter\_notebook', 'source\_code', and 'Start\_Here.py'. The main area shows a Jupyter notebook with the following code:

```
[1]: import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

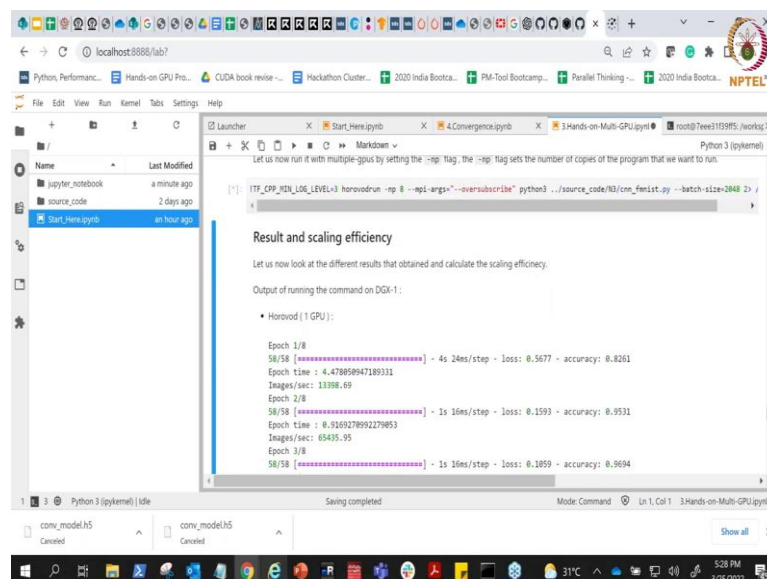
```
[2]: {'status': 'ok', 'restart': True}
```

Below the code, there is a text block explaining the need to use the `horovodrun` command to launch multiple processes. It includes the command:

```
$ horovodrun -np 4 python train.py
```

So, let us stop this as well. So, we have just restarted it.

(Refer Slide Time: 26:27)



The screenshot shows the JupyterLab interface. The left sidebar displays a file explorer with 'jupyter\_notebook', 'source\_code', and 'Start\_Here.py'. The main area shows a Jupyter notebook with the following code:

```
[1]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="-oversubscribe" python3 ./source_code/13/cnn_finetune.py --batch-size=2048 2> /
```

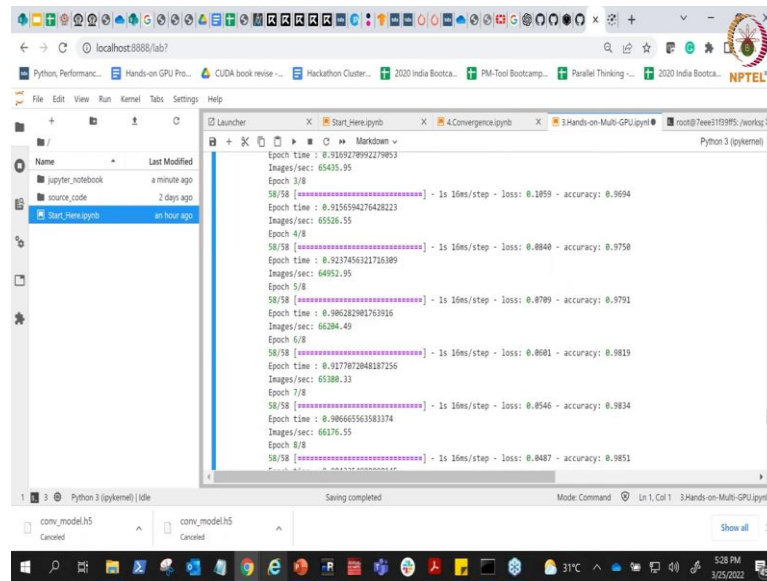
Below the code, there is a text block titled "Result and scaling efficiency" which explains the output of running the command on DGX-1. It includes the command:

```
$ horovodrun -np 4 python train.py
```

And, we are going to now run the horovod with multiple number of processes. You can see here, I am running the same code, but this code I can define the number of processes. So, if I have 2 GPUs, I can set `-np 2`, if I have 4 GPUs, I can set `-np 4`, if I have 8 GPUs, I can set `-np 8`.

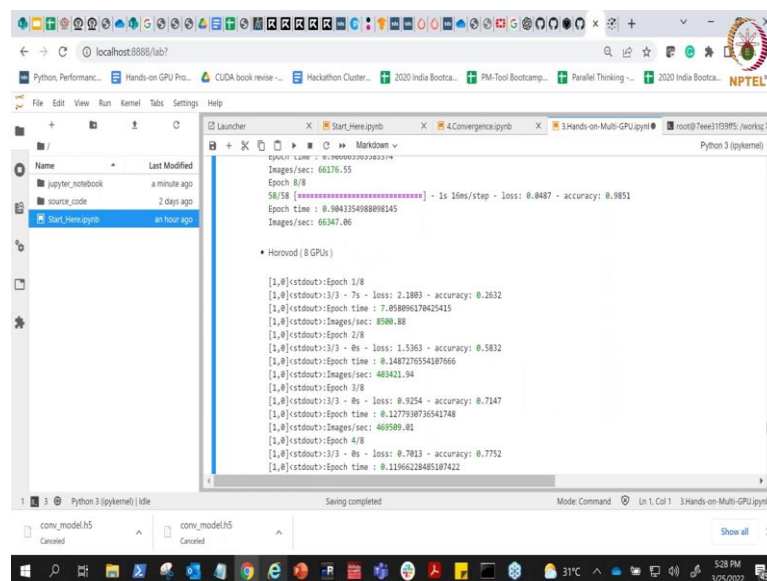


(Refer Slide Time: 26:56)



```
Epoch time : 0.910927899279853
Images/sec : 65435.95
Epoch 3/8
58/58 [=====] - 1s 16ms/step - loss: 0.1859 - accuracy: 0.9694
Epoch time : 0.915659427642823
Images/sec : 65526.55
Epoch 4/8
58/58 [=====] - 1s 16ms/step - loss: 0.0840 - accuracy: 0.9750
Epoch time : 0.9237456321716389
Images/sec : 64952.95
Epoch 5/8
58/58 [=====] - 1s 16ms/step - loss: 0.0709 - accuracy: 0.9791
Epoch time : 0.906282902763916
Images/sec : 66304.49
Epoch 6/8
58/58 [=====] - 1s 16ms/step - loss: 0.0601 - accuracy: 0.9819
Epoch time : 0.917872048187256
Images/sec : 65388.33
Epoch 7/8
58/58 [=====] - 1s 16ms/step - loss: 0.0546 - accuracy: 0.9834
Epoch time : 0.9066556583374
Images/sec : 66376.55
Epoch 8/8
58/58 [=====] - 1s 16ms/step - loss: 0.0487 - accuracy: 0.9851
```

(Refer Slide Time: 26:56)



```
Epoch time : 0.9090000000000001
Images/sec : 66376.55
Epoch 8/8
58/58 [=====] - 1s 16ms/step - loss: 0.0487 - accuracy: 0.9851
Epoch time : 0.9043354988898145
Images/sec : 66347.06

Horovod ( 8 GPUs )

[1,0]<stdout>:Epoch 1/8
[1,0]<stdout>:3/3 - 7s - loss: 2.1803 - accuracy: 0.2632
[1,0]<stdout>:Epoch time : 7.058096170425415
[1,0]<stdout>:Images/sec : 8580.88
[1,0]<stdout>:Epoch 2/8
[1,0]<stdout>:3/3 - 0s - loss: 1.5363 - accuracy: 0.5832
[1,0]<stdout>:Epoch time : 0.1487276554807666
[1,0]<stdout>:Images/sec : 403421.94
[1,0]<stdout>:Epoch 3/8
[1,0]<stdout>:3/3 - 0s - loss: 0.9254 - accuracy: 0.7347
[1,0]<stdout>:Epoch time : 0.1277930736541748
[1,0]<stdout>:Images/sec : 469509.01
[1,0]<stdout>:Epoch 4/8
[1,0]<stdout>:3/3 - 0s - loss: 0.7813 - accuracy: 0.7752
[1,0]<stdout>:Epoch time : 0.11966228485107422
```

So, based on the number of GPUs that you have, you can basically run this and, ideally it is supposed to give you the scaling efficiency that you are expecting out of it. So, there is some problem running this thing on this machine. But, hopefully it will get through in a bit. So, let it run, till the time we will go back to the theory part and hopefully we have got an idea of how to use both Horovod and TensorFlow.