

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

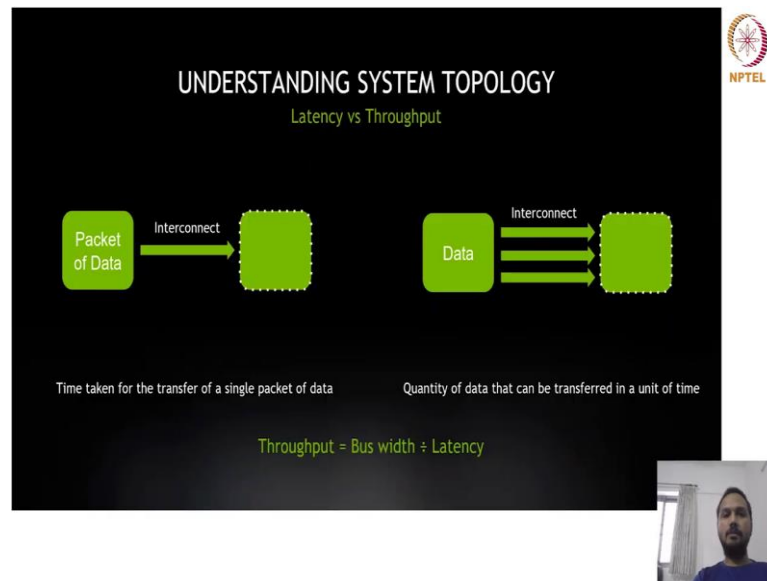
Lecture - 36
Fundamentals of Distributed AI Computing Session 2
Part 1

(Refer Slide Time: 00:14)



Welcome back everyone my name is Bharat and I am going to continue with our previous session. The session on the distributed deep learning the session is going to concentrate or continue with the concepts that we saw in the last lecture. So, again we will look at the concepts that we explained in a bit in terms of the query and then I will continue with hands on session where we are going to show you a demo of all the concepts that were explained.

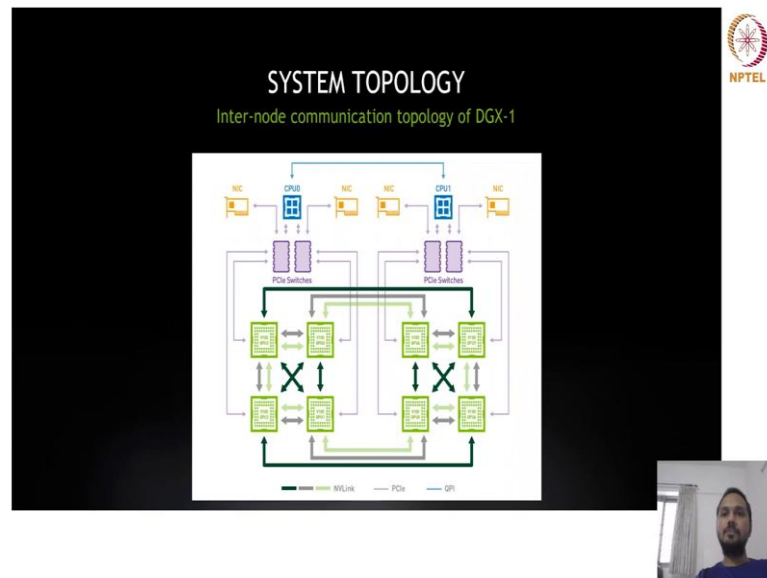
(Refer Slide Time: 01:04)



So, the first thing that we talked about during the session was also the necessity of understanding the system topology. We introduced you to two particular concepts latency and throughput. Latency is nothing but the time taken for the transfer of a single packet of data how much time does it take for me to reach from one destination to the other.

The second one is throughput is the quantity of data that can be transferred in one unit of time like can I transfer 1 KB of data 2 KB of data 3 KB and all and it is important from a point of view of scaling the scaling laws kind of define and they would be dependent on the system on which you are going to run the multi GPU or any kind of a distributed deep learning jobs.

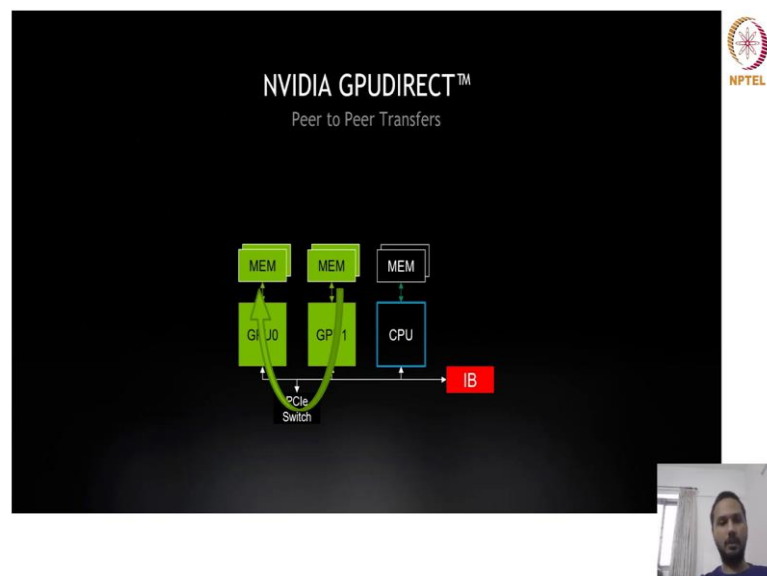
(Refer Slide Time: 02:07)



And we saw a particular system architecture which I explained in the last call which is one of the NVIDIA system called as NVIDIA DGX. NVIDIA DGX system consists of 2 CPUs CPU0 and CPU1 it consists of couple of PCs switches and it also has 8 GPUs this 8 GPUs are connected to each other via interconnects like NVLink.

If you want to go across nodes to different node you get connected via NIC cards or Infiniband which will connect to the other machines as well. We are going to do hands on in this particular session on a DGX machine itself.

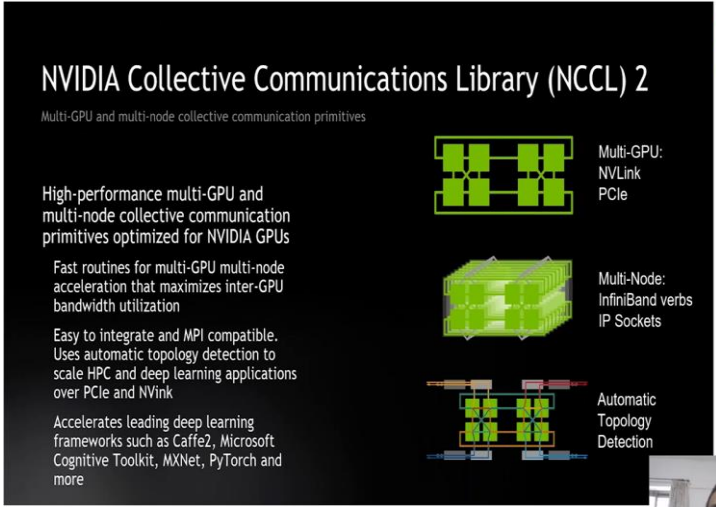
(Refer Slide Time: 02:54)



There is a particular concept that we explained yesterday which is referred to as GPU direct. In this case you can see there is GPU0 and GPU1 no matter what kind of a communication method you use whether the data parallel or model parallel whenever you want to communicate. If you are able to do direct communication between GPU0 and GPU1 referred to as peer to peer transfer directly without involving CPU.

Hence, reducing the latency is referred to as GPU direct. It is only possible under certain scenarios which we will see in the hands on session again.

(Refer Slide Time: 03:41)



The slide features a dark background with white and green text and diagrams. On the right side, there is a small circular logo with a red and white star-like pattern and the text 'NPTEL' below it. The main title 'NVIDIA Collective Communications Library (NCCL) 2' is in a large, bold, white font. Below it, in a smaller white font, is the subtitle 'Multi-GPU and multi-node collective communication primitives'. The slide is divided into three main sections. The first section on the left contains three bullet points in white text: 'High-performance multi-GPU and multi-node collective communication primitives optimized for NVIDIA GPUs', 'Fast routines for multi-GPU multi-node acceleration that maximizes inter-GPU bandwidth utilization', and 'Easy to integrate and MPI compatible. Uses automatic topology detection to scale HPC and deep learning applications over PCIe and NVlink'. The second section in the middle contains three diagrams. The top diagram shows a 2x2 grid of green squares connected by lines, with the text 'Multi-GPU: NVLink PCIe' to its right. The middle diagram shows a 3D stack of green cubes, with the text 'Multi-Node: InfiniBand verbs IP Sockets' to its right. The bottom diagram shows a network topology with green squares and lines, with the text 'Automatic Topology Detection' to its right. The third section on the right contains a small video inset showing a man with a beard and a blue shirt, looking at the camera.

NVIDIA Collective Communications Library (NCCL) 2
Multi-GPU and multi-node collective communication primitives

High-performance multi-GPU and multi-node collective communication primitives optimized for NVIDIA GPUs

Fast routines for multi-GPU multi-node acceleration that maximizes inter-GPU bandwidth utilization

Easy to integrate and MPI compatible. Uses automatic topology detection to scale HPC and deep learning applications over PCIe and NVlink

Accelerates leading deep learning frameworks such as Caffe2, Microsoft Cognitive Toolkit, MXNet, PyTorch and more

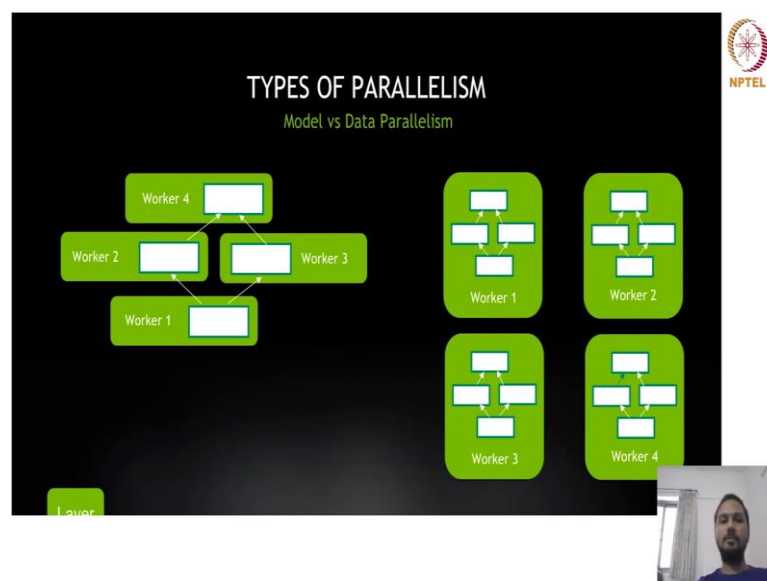
Multi-GPU: NVLink PCIe

Multi-Node: InfiniBand verbs IP Sockets

Automatic Topology Detection

We also talked about NVIDIA collective communication library. NVIDIA collective communication library is a library or a framework which has been built by NVIDIA which helps in doing as the name says collective communications. It is a library which is integrated into all of the frameworks like PyTorch TensorFlow and all to effectively do the multi-GPU when you enable multi-GPU distributed training NCCL is called behind the scenes and we will look at how to see or how to even be assured that this is happening.

(Refer Slide Time: 04:29)



It has various advantages which we have talked about yesterday like it is basically something which does different kinds of detection graph, it does graph analysis to find the shortest path, the path having lowest latency and highest bandwidth it does automatic topology detections and it hence it helps in scaling pretty well on different kinds of clusters.

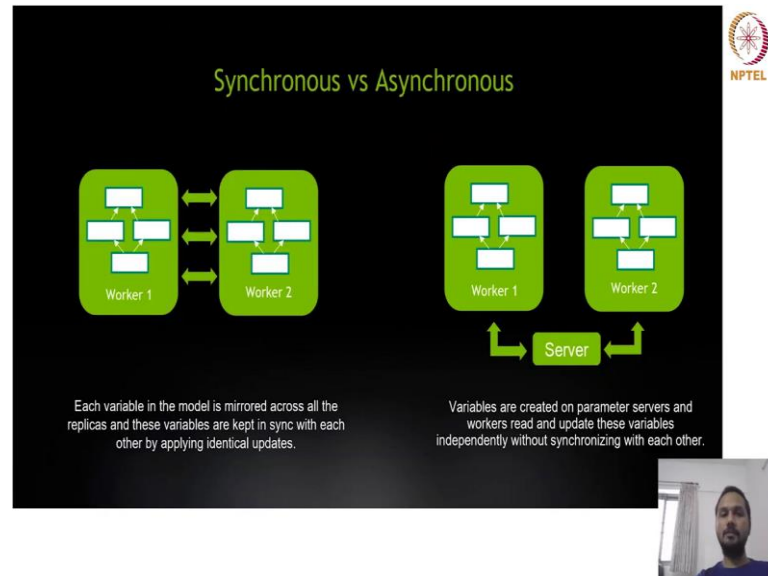
We talked about two different kinds of parallelism whenever you want to go from 1 GPU to multiple GPUs. The first kind of a parallelism is referred to as the model parallelism, the second one is data parallelism. In the model parallelism you would split the model or the layers across different workers each worker accessing a particular separate GPU worker the model parallelism is particularly more difficult and it is primarily used especially when you cannot fit the whole model into one particular gpu. GPUs now a day come with almost 80 GB of memory.

If you are talking about models nowadays you have models still few trillion parameters in that case, it is almost impossible to fit them into one GPU. So, you need to make use of model parallelism. Data parallelism is a way in which you take one model and replicate it across all the workers each having their separate GPU. What you are splitting basically is different batches of data input data which would be required for training.

These two methods can be combined together into hybrid model. In a hybrid model you combine both worker parallelism and data parallelism together to enable real world

problem solving especially when your model is large and the data set is also pretty large in that case you can combine both of them as well.

(Refer Slide Time: 06:38)



The next thing we also said is whether it is no matter what type of parallelism you can do it either in a synchronous fashion or asynchronous fashion. In a synchronous fashion point of view each variable in the model is mirrored across all the replicas and these variables are kept in sync with each other and by applying identical updates.

So, at every step when you are given the respect to like in case of data parallelism whenever you are basically updating the gradients, they are basically at every step they gets exchanged with each and every worker in a synchronous fashion and then they continue to the next step.

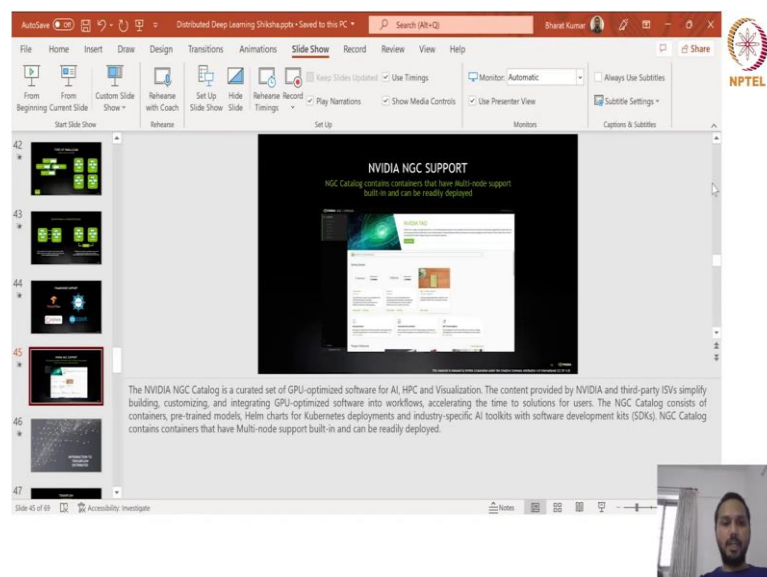
While in asynchronous communication, there might be different methods you do not need to be always at a synchronous step. The variables are can be created on a separate server which can be also referred to as a parameter server and all the workers basically can read and update these variables independently without synchronizing with each other and that is what we saw is a method of asynchronous programming.

(Refer Slide Time: 07:50)



We said in the last session that no matter what method it is or whether we want to use GPUs all of the existing frameworks support the same. You have TensorFlow, you have HOROVOD, PyTorch, mxnet all of them supporting Multi-GPU and they all have different methods for doing data parallelism model parallelism and a hybrid parallelism approach as well today we will try to touch on one or more of them.

(Refer Slide Time: 08:25)

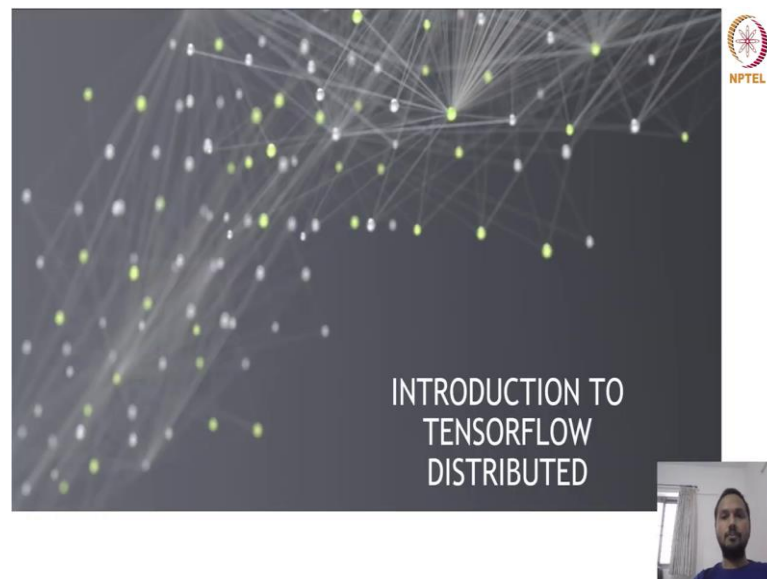


In the today's demo again in the session 1 of this particular series we touched upon the usage of container and how we can utilize or download the containers from NVIDIA

GPU cloud. In the today's session again we are going to use containers and this container would be running on a DGX machine and we have built this container and we will be using Docker for running this container and it has been pulled from the NGC repository.

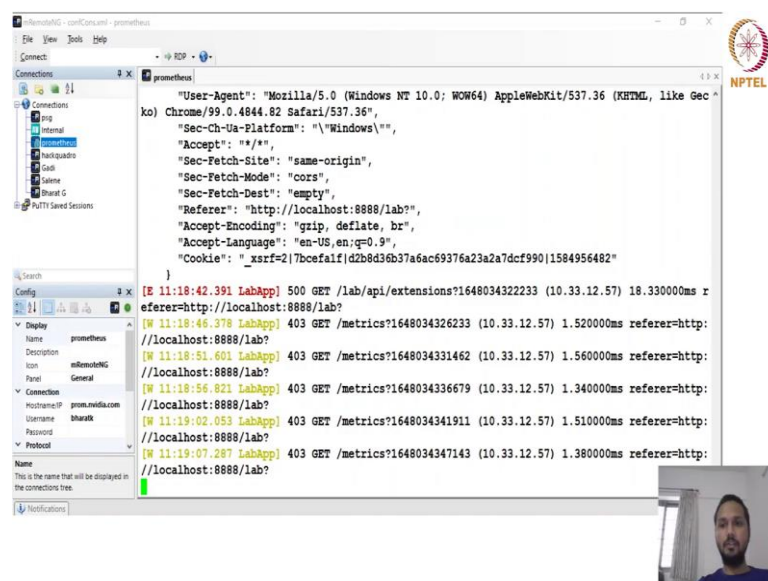
So, in case you have missed that session we recommend you to definitely look at the NGC session which was delivered in session 1.

(Refer Slide Time: 09:09)



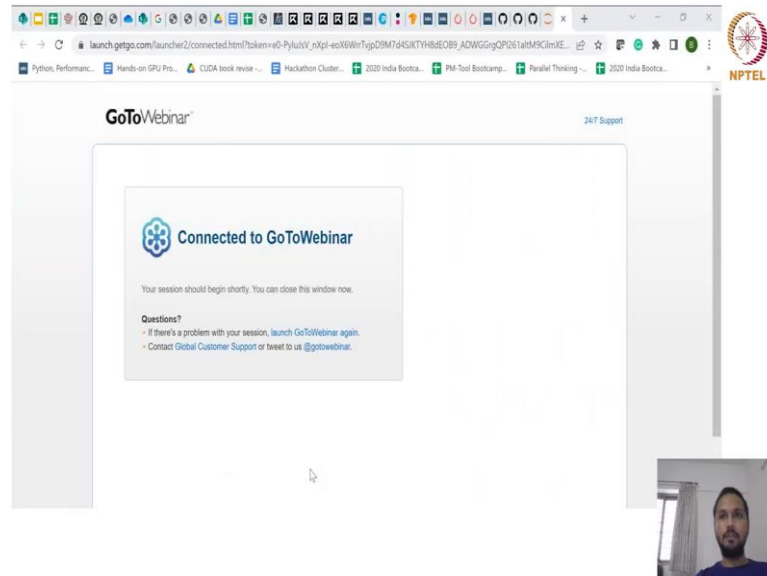
So, let me start by showing you a particular demo.

(Refer Slide Time: 09:16)

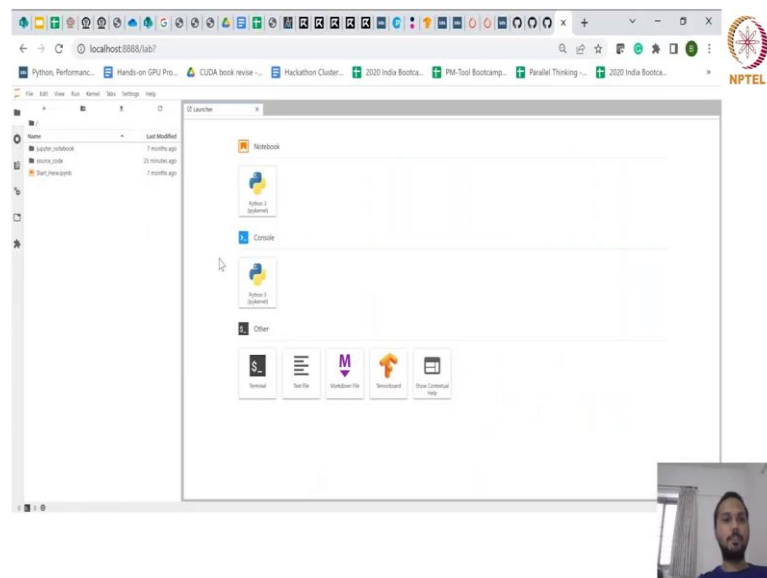


So, as you can see here I have already set up a particular server, in the server I am basically running a Docker container and I have already forwarded it to my machine.

(Refer Slide Time: 09:27)

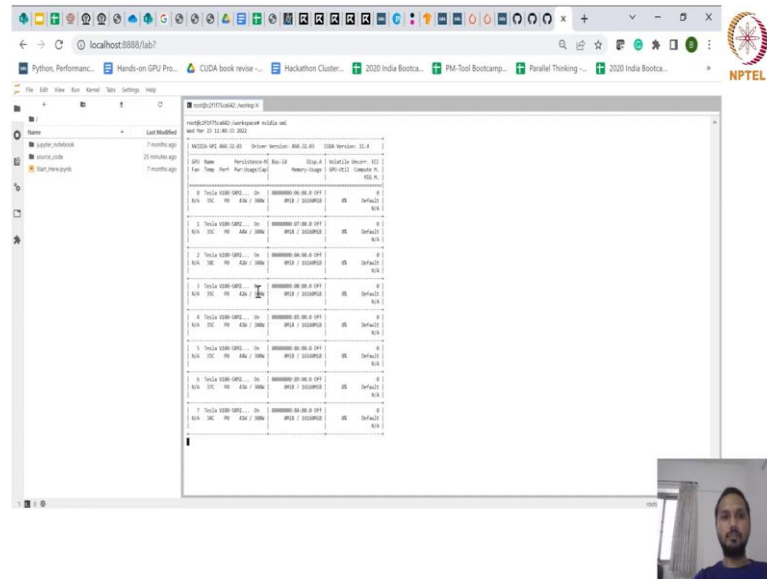


(Refer Slide Time: 09:28)



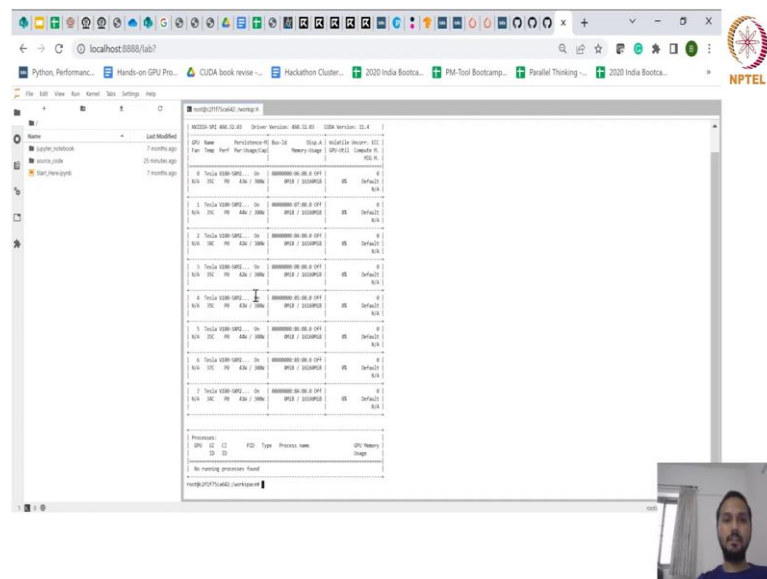
So, all of the materials that you are seeing here they are all been made open source and we will be putting the link to the material by the end of tomorrow's session into the slack channel and you would be able to download this hands on material as well. If you have access to any of the multi-GPU machine you would be able to replicate it and run it in inside a container environment.

(Refer Slide Time: 09:59)



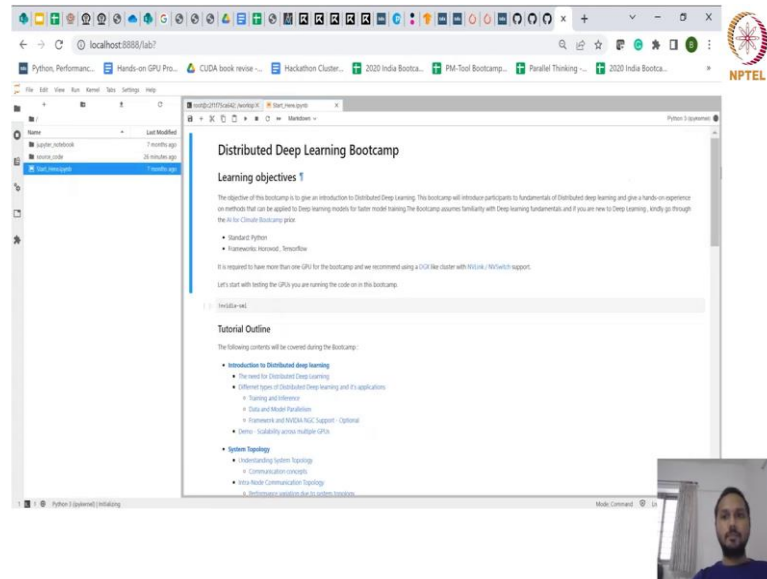
The first thing I would like to show you is the machine itself.

(Refer Slide Time: 10:06)



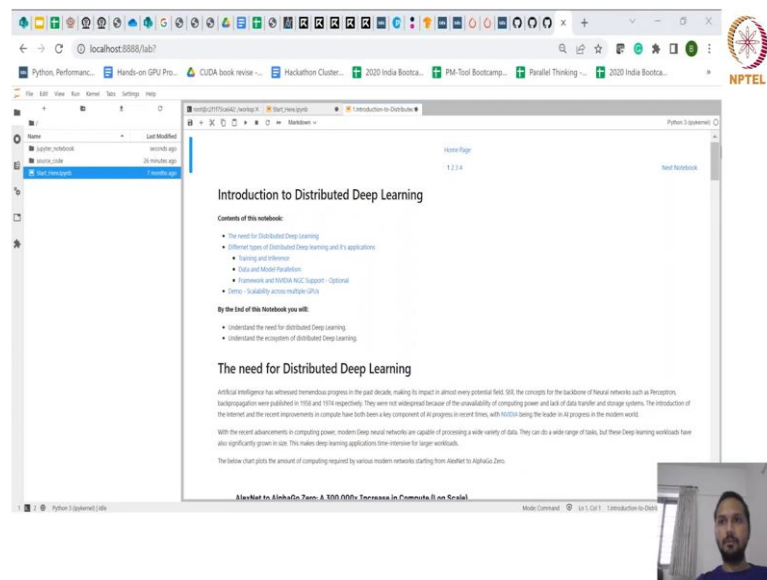
So, if I do nvidia-smi command which was showed you previously also you can see that this machine basically has 8 GPUs the 8 GPUs are of type Tesla V100 and they are all currently 0 percent utilized at this time.

(Refer Slide Time: 10:22)



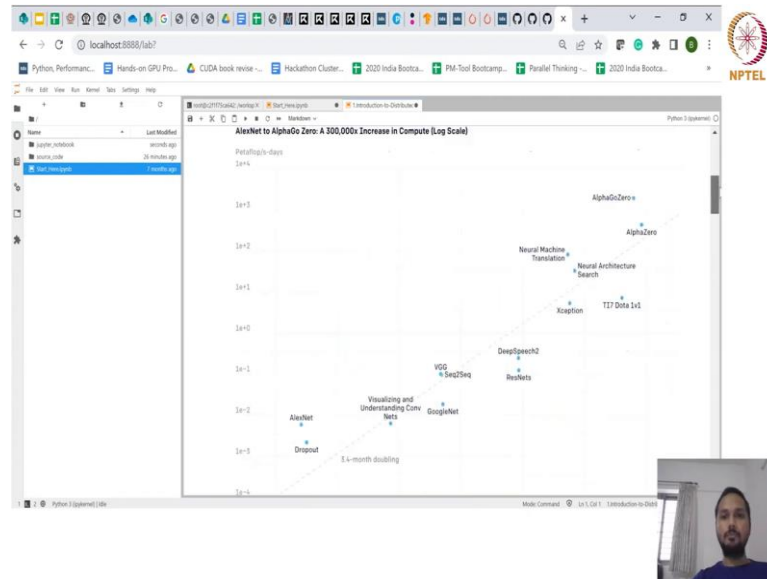
So, let us get started and I am going to start with certain theory once more of distributed deep learning.

(Refer Slide Time: 10:29)



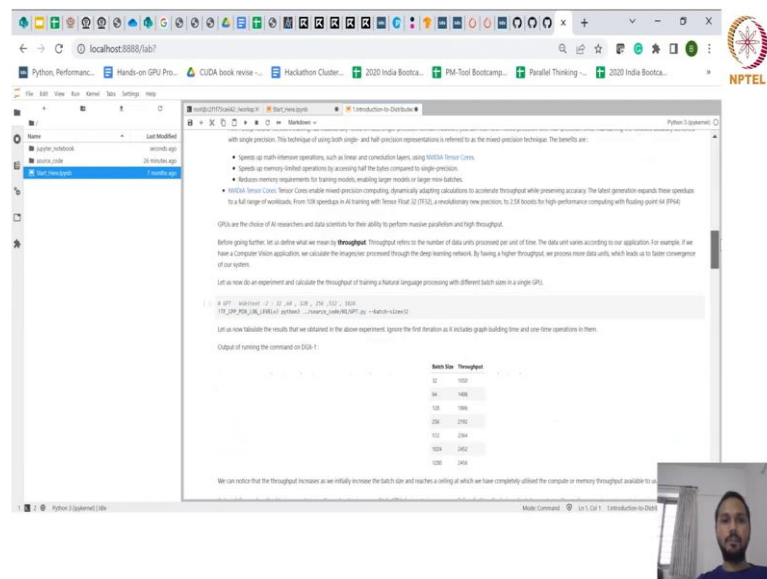
And then we will move towards the part of just give me a second I need to somehow yeah.

(Refer Slide Time: 10:41)

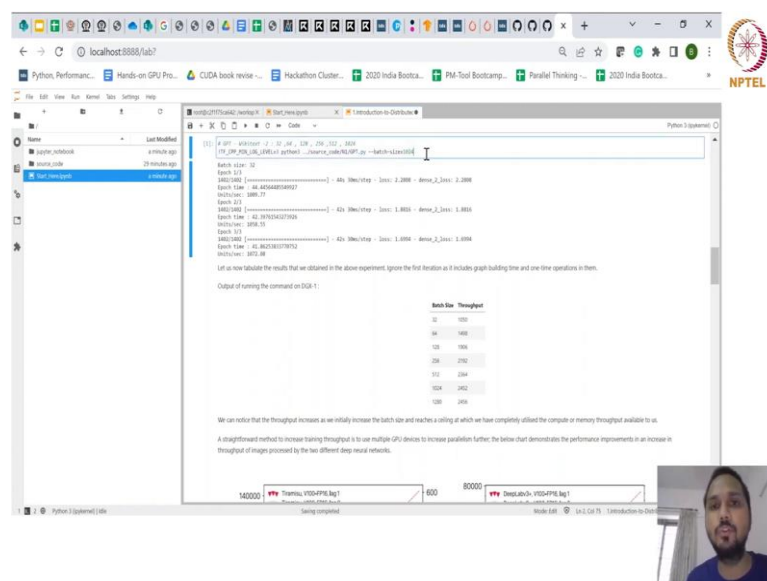


So, we already talked about some of these fundamentals.

(Refer Slide Time: 10:42)



(Refer Slide Time: 10:44)



So, we will quickly skip that, but what I am going to do is I will give you an overview of some of the hyper parameters that you can do to improve the performance of your existing models. So, here I am basically running or calculating the throughput of training of a natural language processing model which is referred to as GPT here and we are going to try a hyper parameter called as batch size which will kind of define in terms of how much performance we can get by tuning this hyper parameters in self themselves.

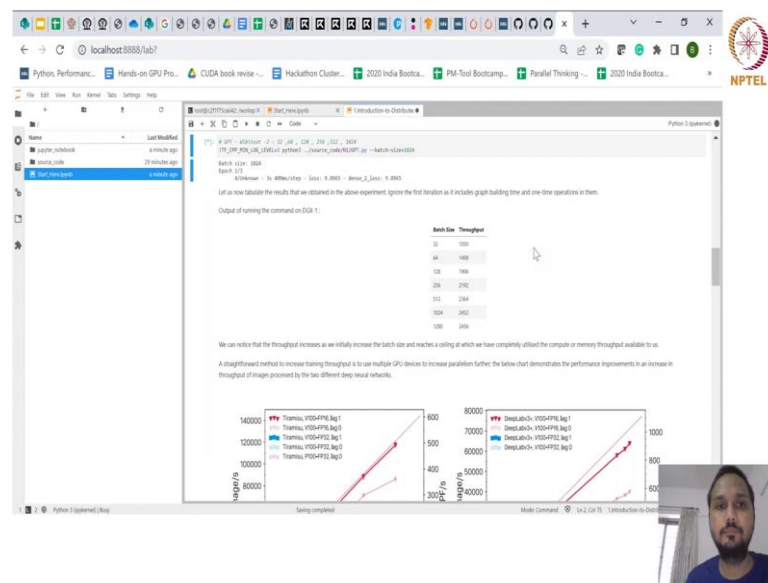
So, it will take some time to finish by the time it finishes you would see something like this. If I change the if I have a batch size of 32 the throughput for me is around 1050. The same if I change my batch size to 64, I will see that my throughput basically increases to 1498 and if I change my batch size to 128 which means I am able to give it a larger throughput in that particular case my throughput will increase from 1906 to 2192 and it keeps on increasing till the time it saturates at 1280 batch size.

What does it mean is something that you can try to get more performance on a single GPU by also trying out certain hyper parameters like the batch size. Generally, GPUs as we said earlier are basically massively parallel architectures and if you give them much more higher batch sizes or much more parallelism then in that case they will give you more performance within the single GPU environment itself. So, let us just wait for to see the 32 and then we will directly jump to 1024 to show you.

So, you can see here the units per second is kind of consistent at 1058.55 which is the throughput of my current training. So, I am running a model called a GPT which is a national language processing model it is quite a large model actually if you do the same on a sequential processing unit you might end up maybe taking day to finish the same job.

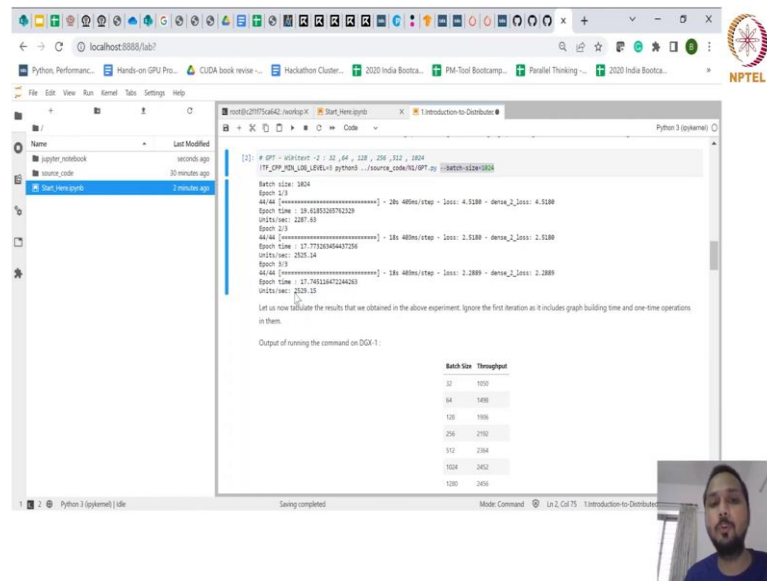
So, as you can see it kind of saturated at 1072. So, let me just change the batch size and make it as 1024 for training the model.

(Refer Slide Time: 13:39)



And let us see what happens. Can one of the organizers confirm if they are able to see my screen and they are able to see the Jupiter Notebook running?

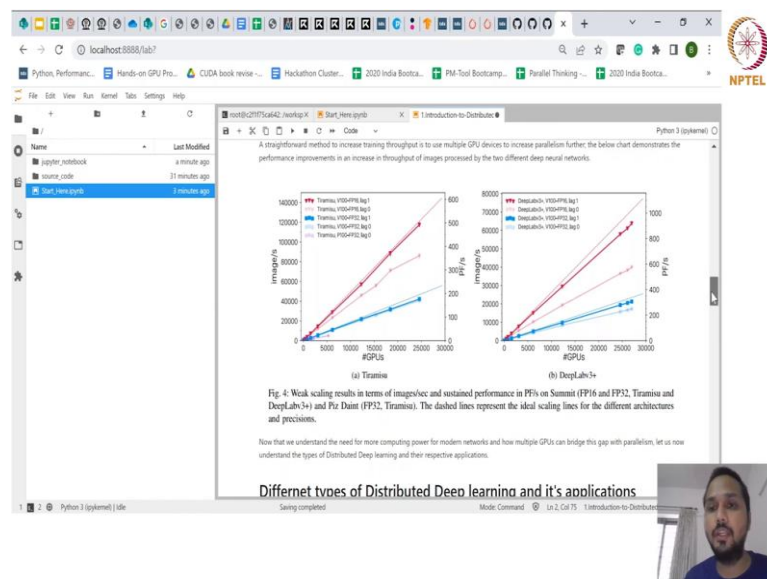
(Refer Slide Time: 14:07)



Student: Yes sir, we can see it.

Thank you right. You can see here previously when we are run it, it was around 1078 when I increase the batch size you can see here that the overall time the throughput doubled it went to 2529 which itself is a very huge difference, but just by changing a hyper parameter called as batch size.

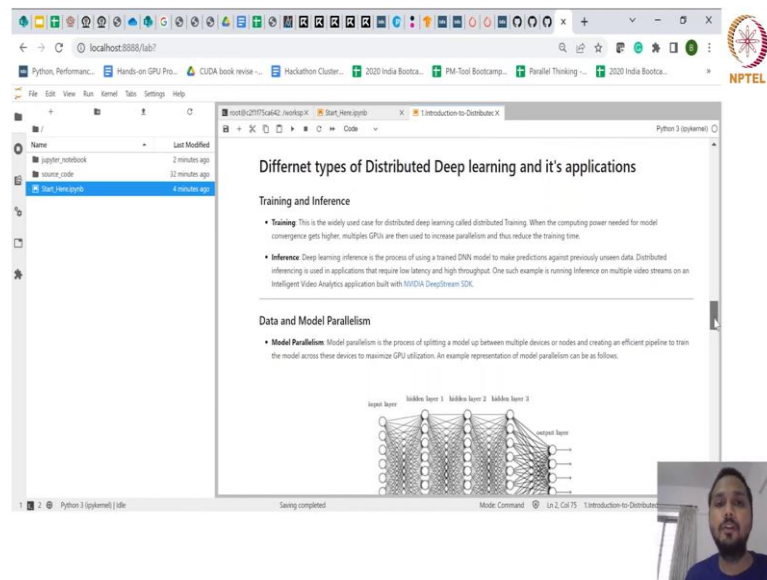
(Refer Slide Time: 15:05)



But as we had discussed last time that the models are becoming larger and larger and we are able to not able to train it on a single GPU because of the memory constraints or time

constraints and we would like to utilize many many GPUs and you can see here an example of a weak scaling in terms of images per second across so many GPUs. We are reaching almost utilizing 30,000 GPUs for this particular example here for running a particular a training.

(Refer Slide Time: 15:42)



Now, we talked about distributed deep learning, but distributed deep learning particularly is not just limited to training phase of artificial intelligence the same can be done at the inferencing stage also. Just to recap in an artificial deep learning space you have two stages the first one is the training phase where you are trying to learn the parameters and then finally, once you have the model trained you are going to deploy it and inference.

You can use distributed deep learning for both training as well as for inferencing like for inferencing we have another systems like NVIDIA deep stream which can help you in scaling your inferencing part of it. Today we are going to focus just on the training part of it using two frameworks and cover some of the fundamentals.

(Refer Slide Time: 16:38)

Model Parallelism

Model parallelism is the process of splitting a model up between multiple devices or nodes and creating an efficient pipeline to train the model across these devices to maximize GPU utilization. An example representation of model parallelism can be as follows.

Data Parallelism In modern deep learning, when the dataset is too big to fit into the memory, we could only do stochastic gradient descent for batches. The shortcoming of stochastic gradient descent is that the estimate of the gradients might not accurately represent the true gradients of using the full dataset. Therefore, it may take much longer to converge. A natural way to have a more accurate estimate of the gradients is to use larger batch sizes or even use the full dataset. To allow this, the gradients of small batches are calculated on each GPU. The final estimate of the gradients is the weighted average of the gradients calculated from all the small batches.

We talked about the model parallelism part and we also talked about the data parallelism part.

(Refer Slide Time: 16:41)

Synchronous data parallelism In synchronous data parallelism, all workers train over different slices of input data in sync and aggregate gradients at each step.

Asynchronous data parallelism In asynchronous data parallelism, all workers are independently training over the input data and updating variables asynchronously.

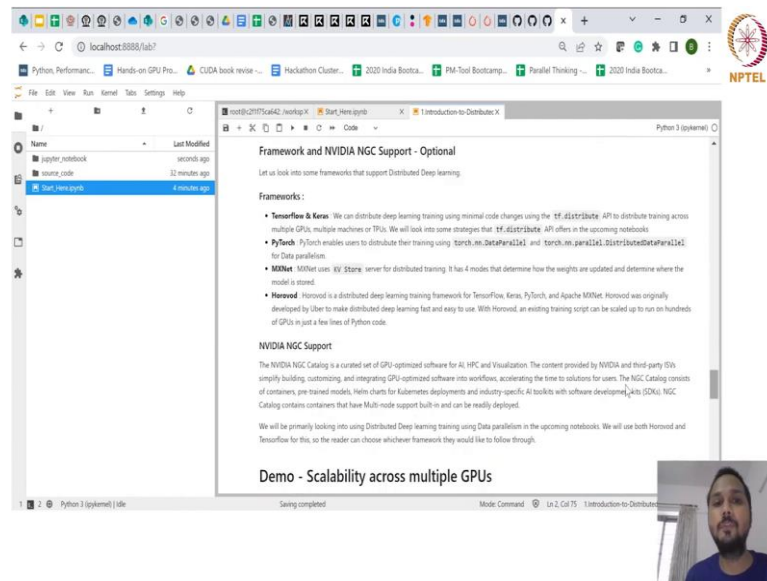
Optional: Typically, `async` training is supported via all-reduce and `async` through parameter server architecture.

Example representation of Synchronous and Asynchronous data parallelism:

Hybrid Parallelism Hybrid parallelism is used when we would like to make use of both Data and Model parallelism. An example would be when we need to train a large model that cannot fit into one GPU but can fit into a node, we could use Model parallelism inside a node and using Data parallelism across nodes.

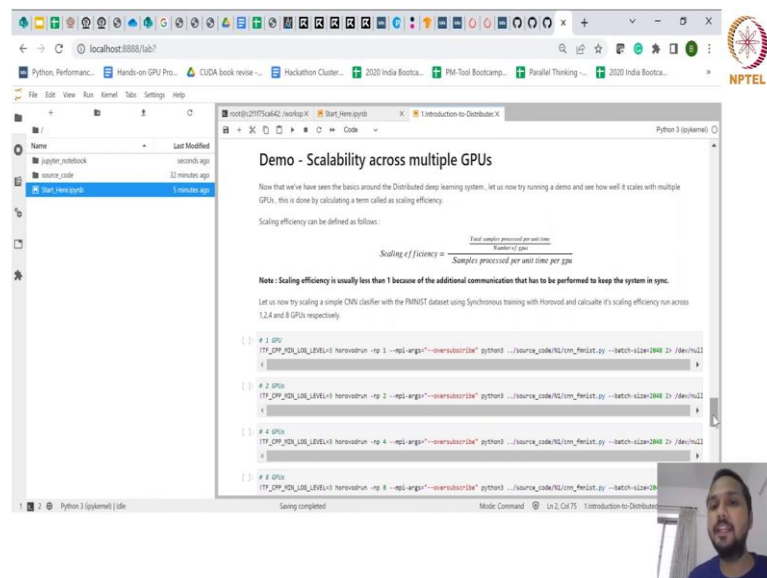
We talked about synchronous and asynchronous communication as well.

(Refer Slide Time: 16:45)



So, all of the frameworks as I said kind of support this and we are going to look into those details later on.

(Refer Slide Time: 16:52)

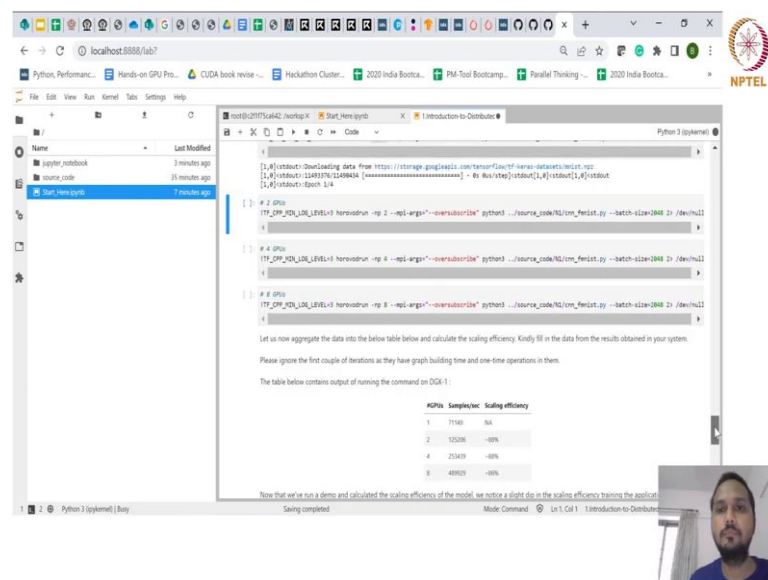


But let us define what do we mean by a when I start using more number of GPUs am I efficient or not efficient or the definition of scalability or scaling efficiency. So, here we are defining the scaling efficiency as the total number of process per unit time. So, how many of the samples are we processing per unit of time divided by the number of GPUs and the overall thing divided by the samples process per unit per GPU.

Which means, I am trying to do the processing for a GPU which has lesser amount of load because you have scaled it across divided by the number of samples that you would have done in a single GPU when you had only one single GPU and that kind of defines the efficiency, am I scaling at 85 percent efficiency to give an example.

Suppose I was able to do 1000 samples per second in a single GPU when I go to two GPU ideally in a 100 percent scaling I should do 2000 samples per second, but if I am not doing 2000 samples per second if it reduced, it will reduce to 85 percent or 95 percent based on this particular formula that we have.

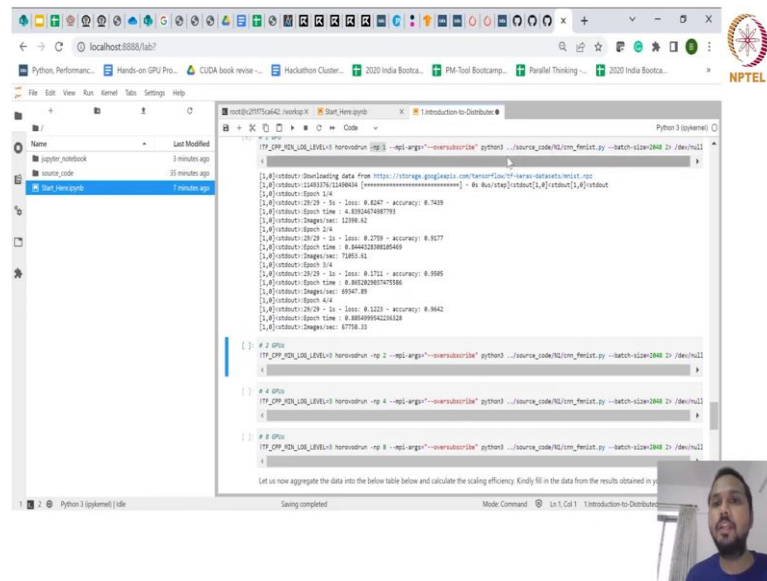
(Refer Slide Time: 18:14)



So, here we are showing you the concept of scaling for the fashion MNIST data set using synchronous training using a framework called as Horovod we are going to see how to use Horovod later on also, but here what we are talking about is the efficiency. The first thing I am doing is we are going to run one GPU and I have enabled certain logs to make sure that we are able to get more details when we are running this.

And also as you can see there are certain additional parameters we will talk about those additional parameters later on, but we are running it on a Fashion MNIST for a synchronous data set. And we have converged on the batch size we are using the same batch size which is 2048. So, I am going to run using only 1 GPU even though I have 8 GPUs with me.

(Refer Slide Time: 19:15)



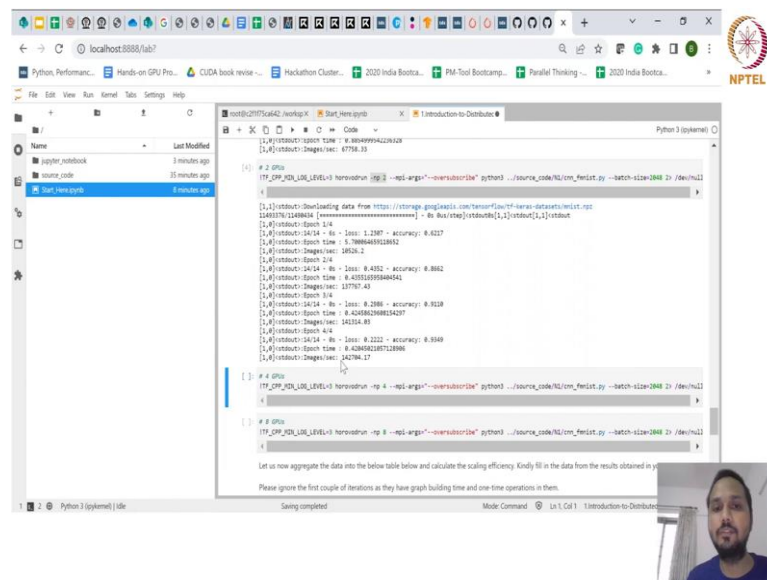
```
TF_CPP_MIN_LOG_LEVEL=3 horovodrun --np 1 --mpi-args="--over-subscribe" python3 .../source_code/NL/cnn_ferList.py --batch-size=2840 --data-dir=/data/nl21
```

Epoch	Time	Loss	Accuracy
1	1.24037612408034	0.8247	0.7439
2	1.24037612408034	0.8247	0.7439
3	1.24037612408034	0.8247	0.7439
4	1.24037612408034	0.8247	0.7439
5	1.24037612408034	0.8247	0.7439
6	1.24037612408034	0.8247	0.7439
7	1.24037612408034	0.8247	0.7439
8	1.24037612408034	0.8247	0.7439
9	1.24037612408034	0.8247	0.7439
10	1.24037612408034	0.8247	0.7439

Let us now aggregate the data into the below table below and calculate the scaling efficiency. Kindly fill in the data from the results obtained in your notebook.

So, you can see here when I use 1 GPU to train a fashion MNIST data set on a GPU it is giving a throughput of images per second of 67,758 right.

(Refer Slide Time: 19:33)



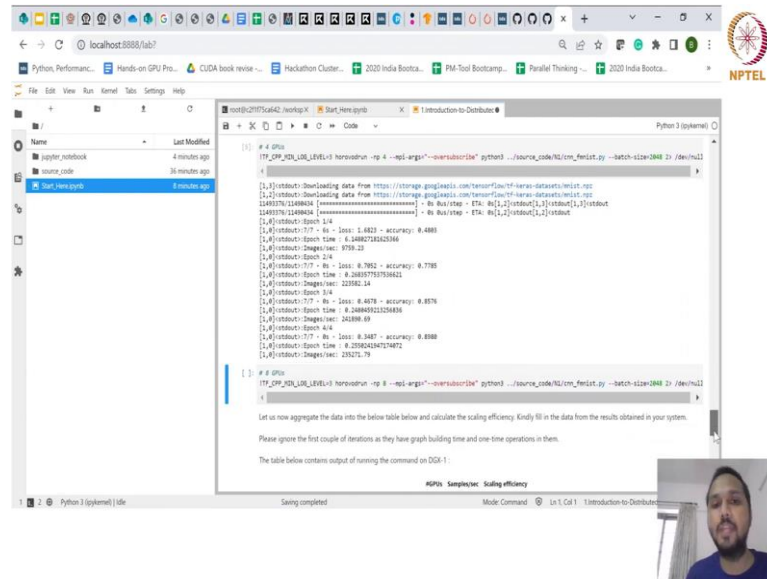
```
TF_CPP_MIN_LOG_LEVEL=3 horovodrun --np 2 --mpi-args="--over-subscribe" python3 .../source_code/NL/cnn_ferList.py --batch-size=2840 --data-dir=/data/nl21
```

Epoch	Time	Loss	Accuracy
1	1.24037612408034	0.8247	0.7439
2	1.24037612408034	0.8247	0.7439
3	1.24037612408034	0.8247	0.7439
4	1.24037612408034	0.8247	0.7439
5	1.24037612408034	0.8247	0.7439
6	1.24037612408034	0.8247	0.7439
7	1.24037612408034	0.8247	0.7439
8	1.24037612408034	0.8247	0.7439
9	1.24037612408034	0.8247	0.7439
10	1.24037612408034	0.8247	0.7439

Please ignore the first couple of iterations as they have graph building time and one-time operations in them.

Now, without changing anything I am just going to provided it with more GPUs in this case I am talking about running it across 2 GPUs and ideally I expect it to give double the images per second to mean. So, you can see here it has gone to almost 1,42,000 images per second and I can train much more faster when I added one more GPU to it.

(Refer Slide Time: 20:04)

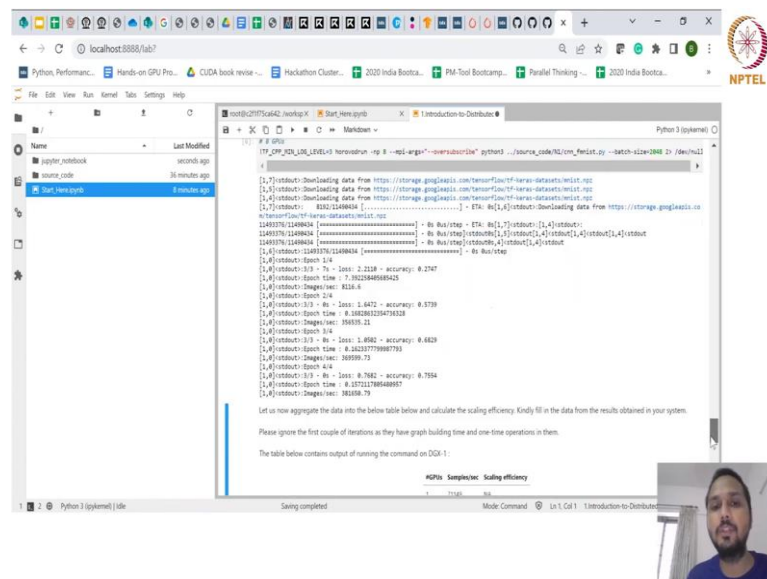


```
# 2 GPUs
TF_CPP_MIN_LOG_LEVEL=3 herculesrun -np 4 --api-args="--over-subscribe" python3 .../source_code/ML/conv_ferret.py --batch-size=2048 2> /dev/null2

[1,1]istudent: Downloading data from https://storage.googleapis.com/tensorflow/tf-datasets/train.py
[1,2]istudent: Downloading data from https://storage.googleapis.com/tensorflow/tf-datasets/train.py
1440376/1440404 [#####] - 8s Bus/step - ETA: 8s [1,2]istudent[1,3]istudent
1440376/1440404 [#####] - 8s Bus/step - ETA: 8s [1,2]istudent[1,2]istudent
[1,4]istudent: Epoch 1/4
[1,4]istudent: Epoch time : 6.4480721625346
[1,4]istudent: 2048000/step
[1,4]istudent: Epoch 2/4
[1,4]istudent: Epoch time : 6.7992 - accuracy: 0.7785
[1,4]istudent: 2048000/step
[1,4]istudent: Epoch 3/4
[1,4]istudent: Epoch time : 6.6578 - accuracy: 0.8576
[1,4]istudent: 2048000/step
[1,4]istudent: Epoch 4/4
[1,4]istudent: Epoch time : 6.3887 - accuracy: 0.8588
[1,4]istudent: 2048000/step
[1,4]istudent: Epoch time : 6.2592436754072
[1,4]istudent: 2048000/step
```

What if I had 4? So, you can see here it increased from 1,42,704 to 2,35,271.

(Refer Slide Time: 20:33)

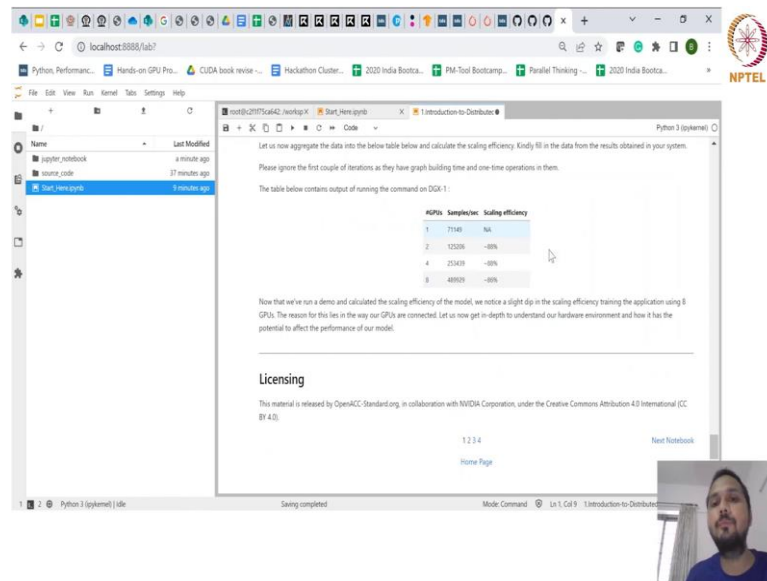


```
# 4 GPUs
TF_CPP_MIN_LOG_LEVEL=3 herculesrun -np 8 --api-args="--over-subscribe" python3 .../source_code/ML/conv_ferret.py --batch-size=2048 2> /dev/null2

[1,2]istudent: Downloading data from https://storage.googleapis.com/tensorflow/tf-datasets/train.py
[1,2]istudent: Downloading data from https://storage.googleapis.com/tensorflow/tf-datasets/train.py
1440376/1440404 [#####] - 8s Bus/step - ETA: 8s [1,2]istudent[1,3]istudent[1,4]istudent
1440376/1440404 [#####] - 8s Bus/step - ETA: 8s [1,2]istudent[1,3]istudent[1,4]istudent
1440376/1440404 [#####] - 8s Bus/step - ETA: 8s [1,2]istudent[1,3]istudent[1,4]istudent
[1,4]istudent: Epoch 1/4
[1,4]istudent: Epoch time : 7.762544694525
[1,4]istudent: 2048000/step
[1,4]istudent: Epoch 2/4
[1,4]istudent: Epoch time : 6.6472 - accuracy: 0.5739
[1,4]istudent: 2048000/step
[1,4]istudent: Epoch 3/4
[1,4]istudent: Epoch time : 6.5892 - accuracy: 0.6629
[1,4]istudent: 2048000/step
[1,4]istudent: Epoch 4/4
[1,4]istudent: Epoch time : 6.3712176988957
[1,4]istudent: 2048000/step
```

And it kind of continues and I am now utilizing all of the 8 GPUs which are there. So, in general this number would change from one machine to the other based on the kind of topology you have and we are going to go into the details of how that works. So, it increases from 2,35,000 to 3,81,000.

(Refer Slide Time: 20:55)



The screenshot shows a Jupyter Notebook interface with a table of scaling efficiency data. The table has three columns: GPUs, Samples/sec, and Scaling efficiency. The data is as follows:

GPUs	Samples/sec	Scaling efficiency
1	7140	NA
2	12306	-88%
4	25349	-88%
8	48929	-88%

The notebook also includes text explaining the scaling efficiency calculation and a licensing notice.

By now you would have observed that ideally if it was scaling at 100 percentage efficiency I should have got some 4,00,000 something right, but it is not scaling at that rate and that is what we were referring to above. So, the numbers the scaling efficiency when you go from 1 GPU to 2 GPU to 4 to 8 is scaling at a particular efficiency.

And this is how you can also measure how well you are able to scale when you keep on adding more and more and with more and more GPUs being added the idea is that you end up for doing faster training without having to compromise on the accuracy. Now there are cases where your accuracy gets compromised and that is what we will cover in the last session.