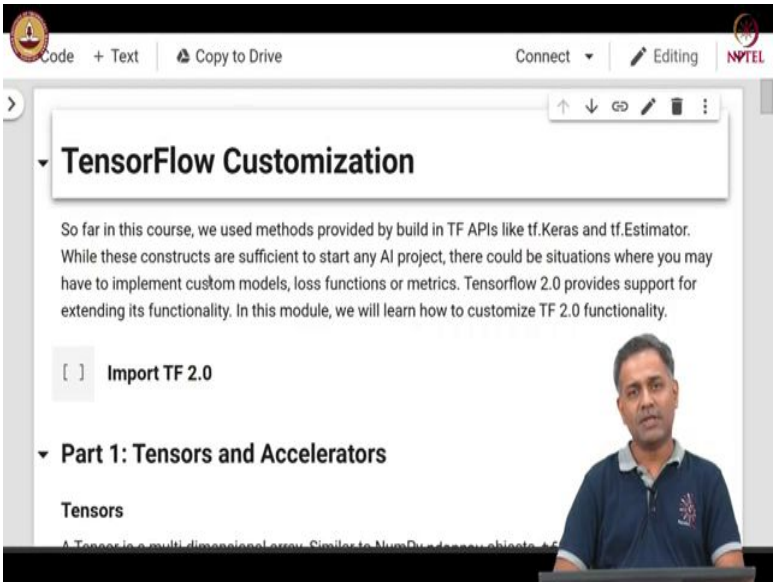
Practical Machine Learning with Tensorflow
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 35
TensorFlow Customization

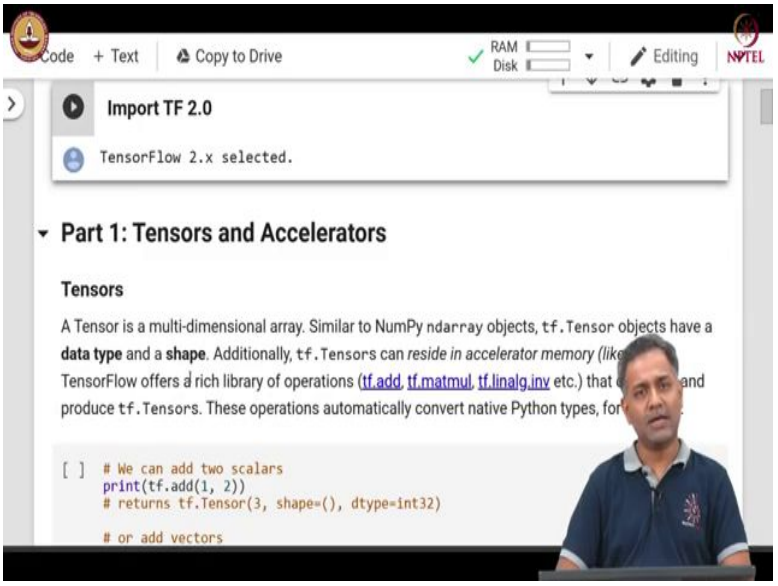
So, far in this course we used TensorFlow functionality to implement machine learning models. There are many situations in practical world where; we may have to implement new machine learning models or use the existing TensorFlow functionality and extend it to suit our needs.

(Refer Slide Time: 00:27)



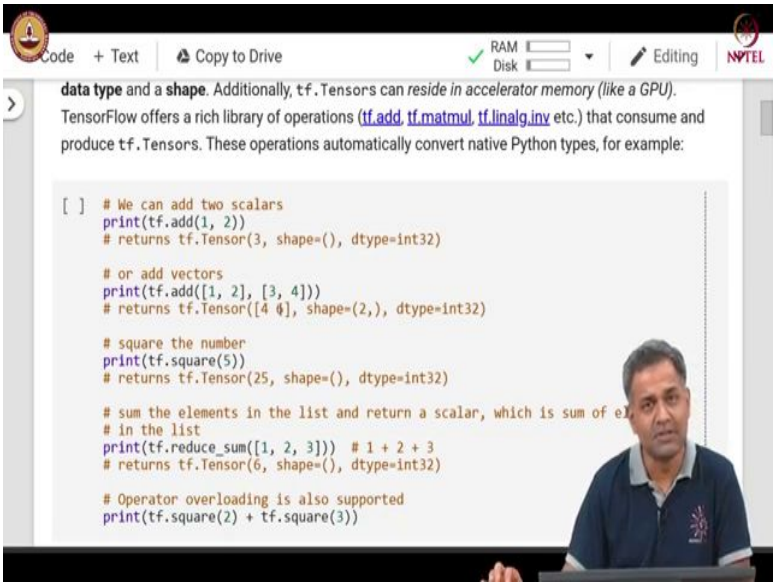
TensorFlow 2.0 provides support to extend its functionality. In this session, we will learn how to extend functionality of TensorFlow 2.0. Before we begin, let us review the basic concepts behind tensors and how to use tensors with hardware accelerators. Let us import TensorFlow 2.0.

(Refer Slide Time: 01:10)



As you know a tensor is a multi dimensional array, it has shape and data type. It is similar to np.ndarray with an exception that tensor can also reside in the accelerators memory like GPU.

(Refer Slide Time: 01:33)

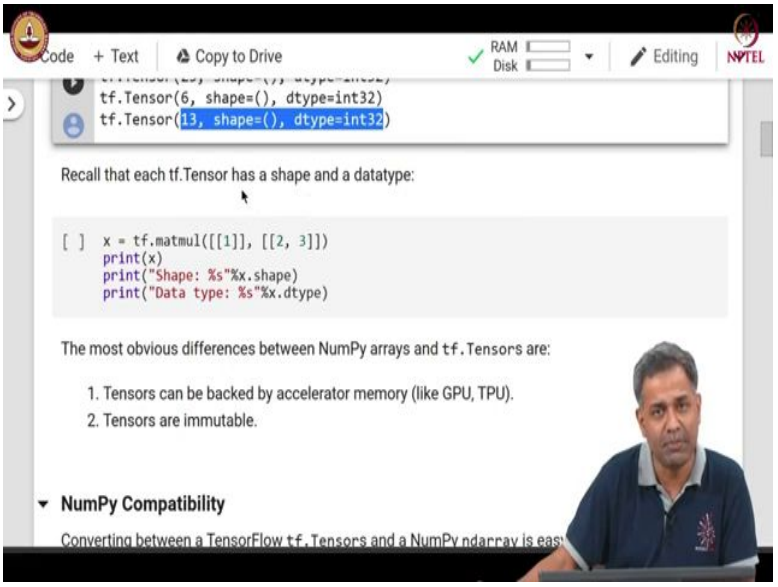


TensorFlow offers a rich library of operations like addition, multiplication. It consumes and produces tf.Tensors. This operations automatically converts native Python types, let us look at a few examples.

We can add two scalars as shown here. We simply add number 1 and 2. This returns a scalar tensor containing a value of 3. It is a scalar tensor having integer32 elements in it. In a similar manner, we can add two vectors and this addition operation returns a tensor which is a vector with integer values in it.

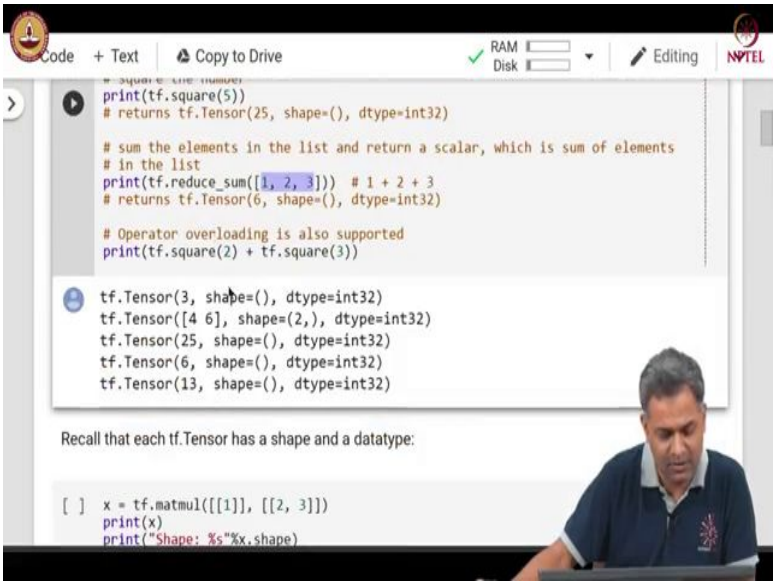
We can also square the number, which returns again a scalar tensor. We can also sum the elements in the list.

(Refer Slide Time: 02:58)



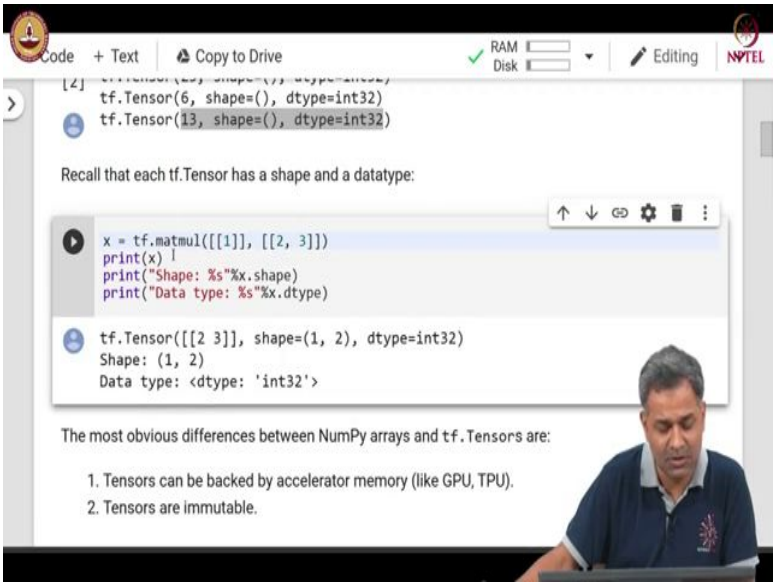
We can also apply the tensor operations and do addition. So, in that sense operation overloading is also supported. Let us run this snippet of the code and examine the output.

(Refer Slide Time: 03:13)



We can see that the outputs are as per our expectations. Recall that, each tensor has a shape and a data type.

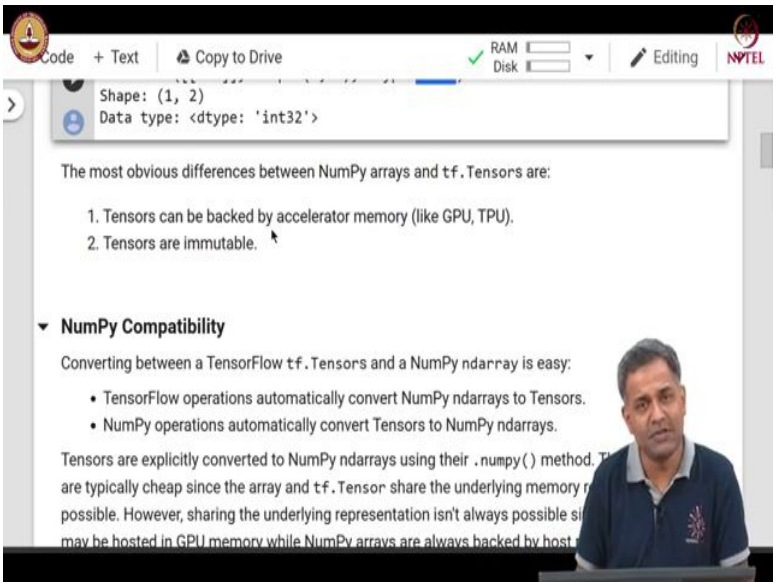
(Refer Slide Time: 03:48)



And as we have seen earlier in this course we can examine the shape of the tensor using .shape attribute and data type using .dtype. Here we multiply two vectors using matrix

multiplication operations which produces a tensor which is a matrix of shape (1, 2) and it holds integer elements in it.

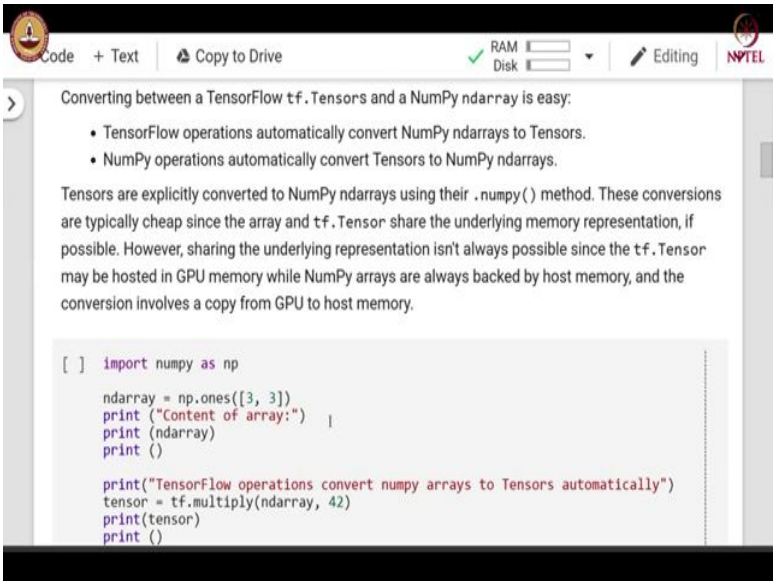
(Refer Slide Time: 04:24)



The most obvious difference between NumPy arrays and tf.Tensors are tensors can be backed up by accelerator memory and tensors are immutable. Converting between a TensorFlow tf.Tensors and NumPy array is easy. TensorFlow operations automatically converts NumPy arrays to tensors.

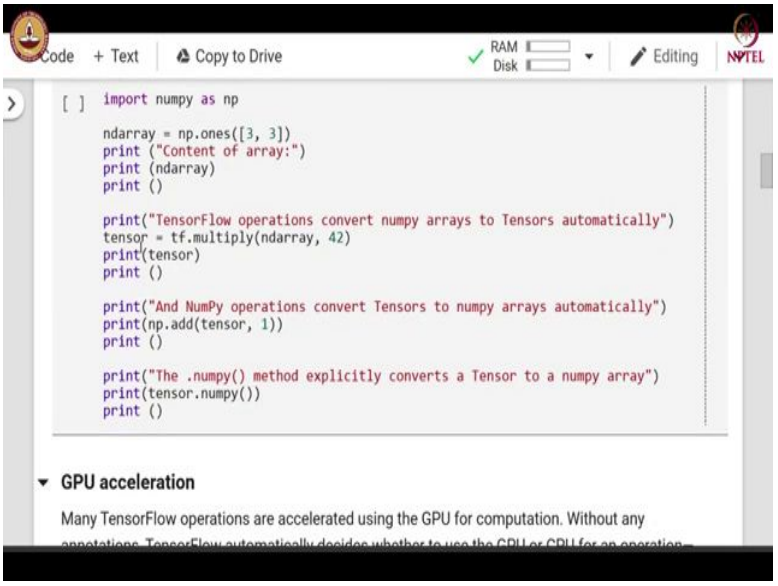
NumPy operations automatically converts tensors to NumPy array. Tensors are explicitly converted to NumPy array using .numpy method. These conversions are typically cheap, because the array and tf.Tensor share the same underlying memory representation.

(Refer Slide Time: 05:10)



However, sharing the underlying representation is not always possible since tf.Tensor can also reside in the accelerators memory. If the tf.Tensor is hosted in GPU, then we have to first make a copy to memory and then carry out the conversion to the NumPy. Let us look at an example.

(Refer Slide Time: 05:36)

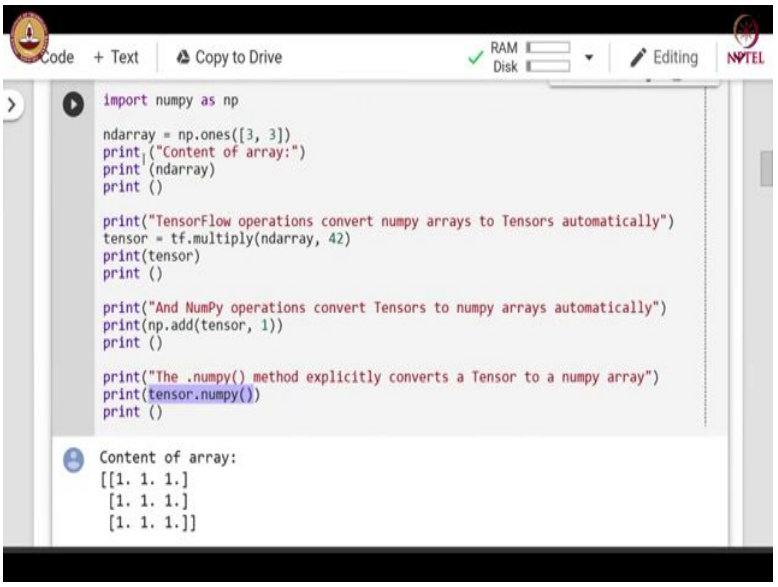


Here we take a NumPy matrix which is 3 by 3 matrix, which is initialized to ones. We later convert it to tensor, we multiply this particular matrix by number 42. And, then we perform

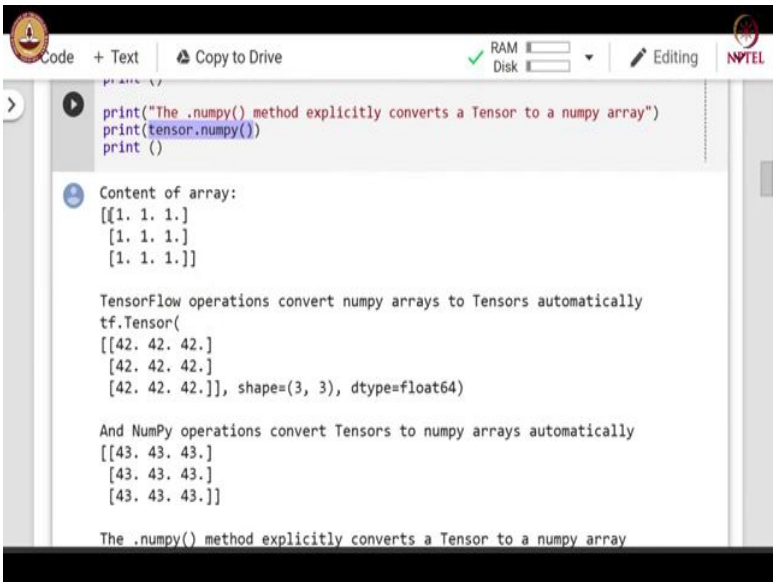
addition using np.add command. So, here you are applying tensor operations on NumPy array where here we are applying NumPy operations on tensors.

And in this particular statement, we show how to get NumPy representation for the tensor. Let us run this code and examine the output.

(Refer Slide Time: 06:42)

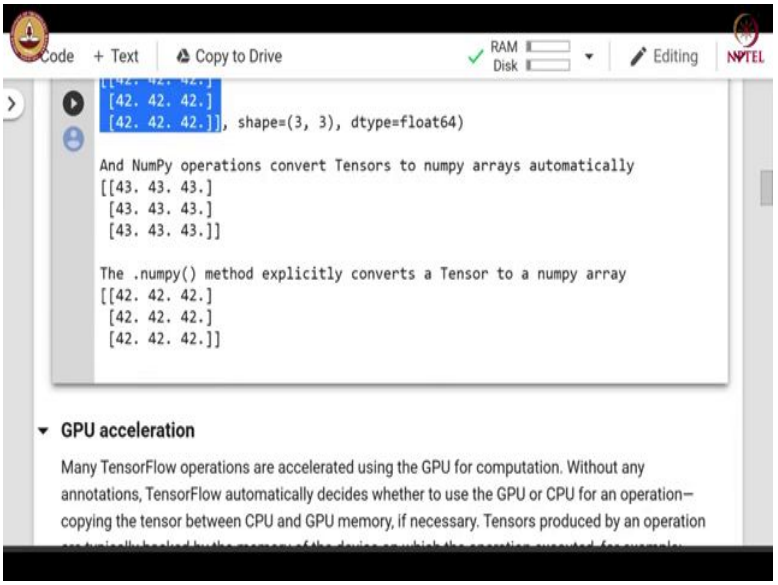


(Refer Slide Time: 06:50)



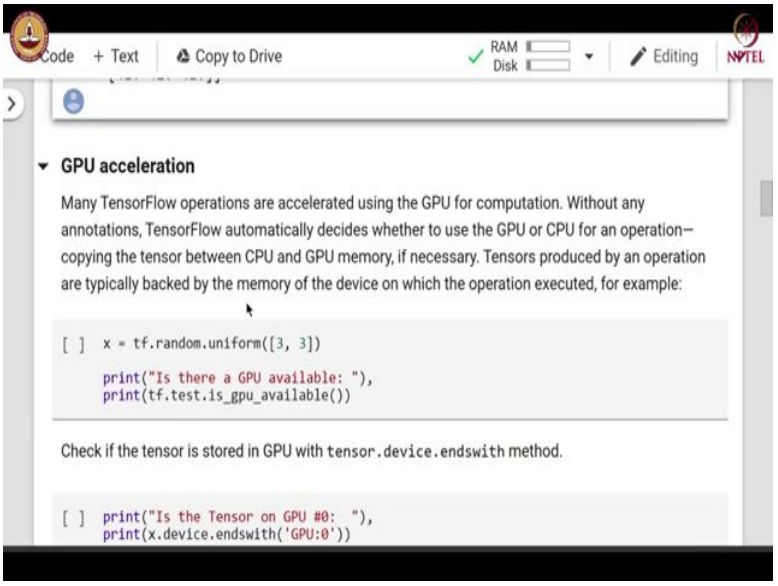
We can see that are array is, you can see that your matrix to begin with 3x3 matrix containing all ones. We added 42 it. We added, we multiplied this particular matrix by 42 getting 42 at each location in the matrix. Then we add 1 to this particular matrix through np.add operation. So, we get content 43 at each cell in the matrix. And finally, we convert the tensor to numpy array note that the tensor was finally, holding values 42 at each cell in the matrix.

(Refer Slide Time: 07:36)



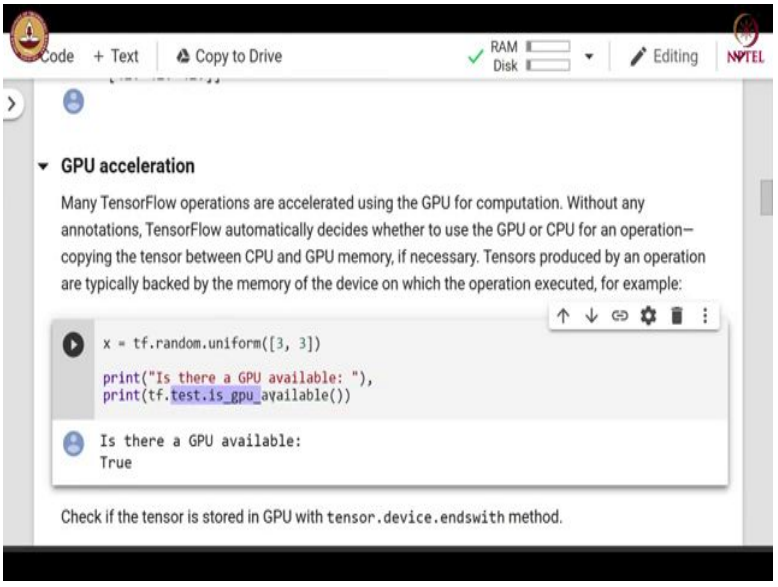
And that is why we see the output of 43. So, this np.add operation converts tensor into nd array and then adds 1, that is why the output of this is not copied to the original tensor. And, hence we see 42 here and not 43.

(Refer Slide Time: 08:09)



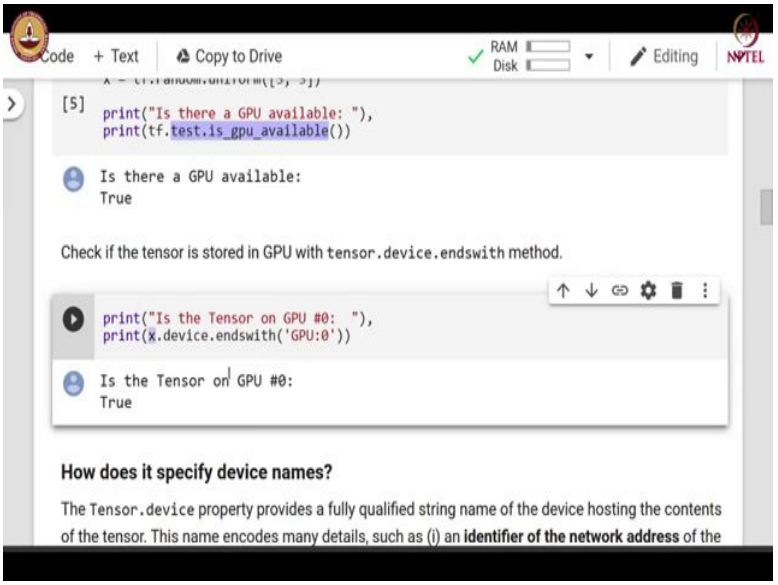
Many TensorFlow operations can be accelerated using GPUs without any annotations. TensorFlow automatically decides whether to use GPU or CPU for any operation. When it decides to use a GPU, it copies the data to GPU and uses it for performing the desired operation. Tensors produced by the operation are typically backed by memory of the device on which the operation was executed. Let us understand this concept with an example.

(Refer Slide Time: 08:49)



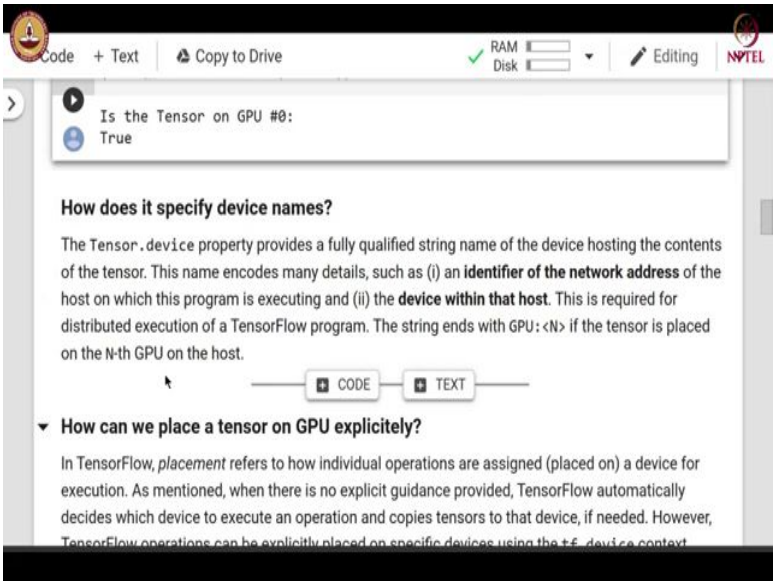
We define a 3x3 matrix with random numbers drawn from a uniform distribution. We check if GPU is available with us. Since, we use GPU hardware accelerator for platform; GPU is available to us that is why we can see that each GPU available as true. We check whether GPU is available using test.is_gpu_available method.

(Refer Slide Time: 09:20)



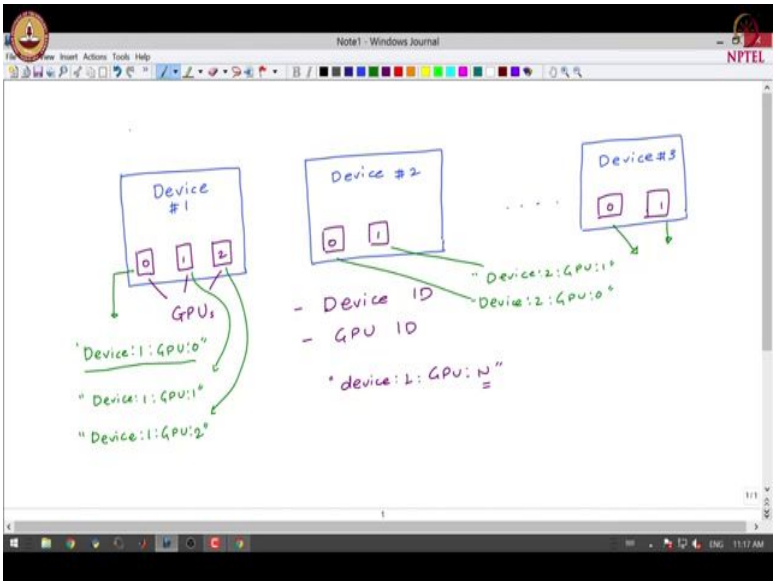
Next we check if the tensor is stored on the GPU. We check that using Tensor.device.endswith method. You can see that this particular tensor is stored on GPU.

(Refer Slide Time: 09:45)



Let us try to understand how the device names are specified in TensorFlow. Tensor.device property provides a fully qualified string name of the device hosting the contents. The name encodes many details such as identifier of the network address of the host on which this program is executing and the device within that host.

(Refer Slide Time: 10:20)

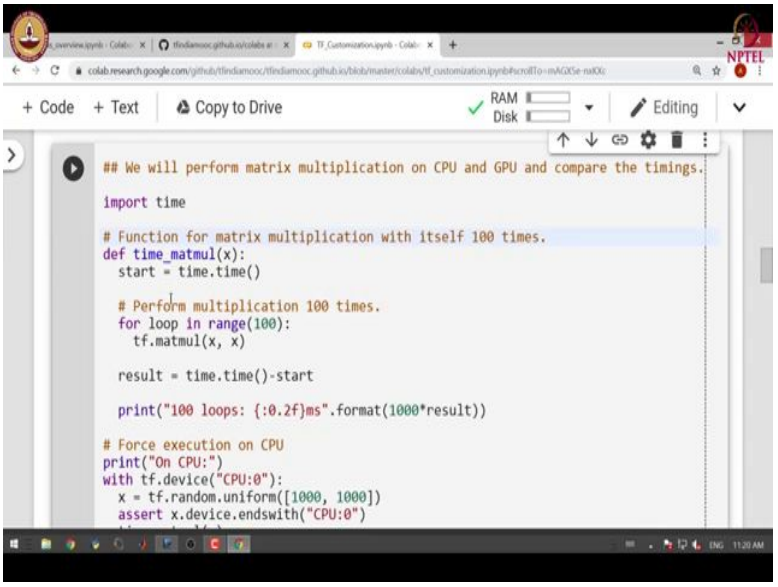


So, there are two parts to it. One device may have multiple GPUs. Firstly, there exists a numbering of devices in the system. And each device may have multiple GPUs. Each GPU within a device is identified by a GPU ID.

So, in order to understand where exactly the tensor is stored, you need to know the device ID and the GPU ID within the device. So, what we do is; we have a string that will have things like, "Device:1:GPU:N" and here, N is the ID of the GPU.

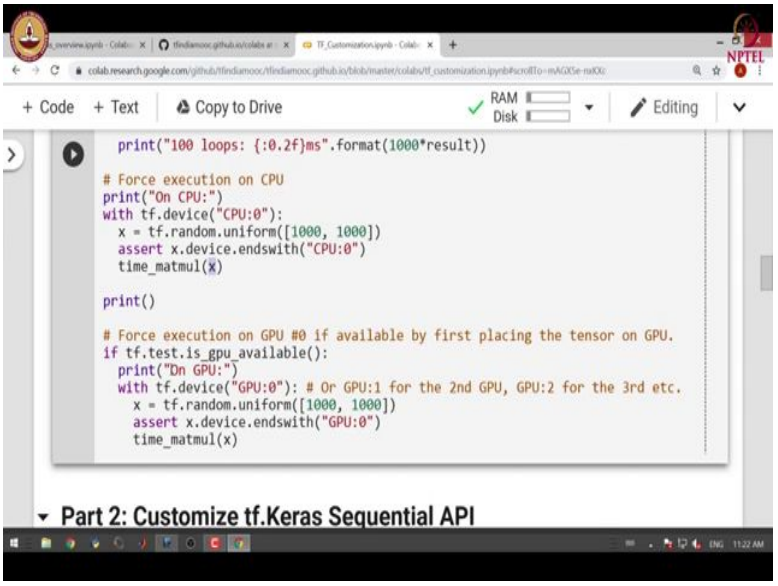
This particular path, gives us a complete identification of the device; where the tensor is stored. We know that TensorFlow automatically decides whether to carry out an operation on a specific GPU device or not. TensorFlow also automatically decides whether to place a tensor on a GPU or not. We can also specify or instruct TensorFlow whether to put a particular tensor on GPU or not. Let us see how to do that particular thing.

(Refer Slide Time: 14:06)



So, here is an example; in this function we multiply the matrix with itself 100 times. And what we do is, we find out how much time it takes to carry out this particular multiplication operation.

(Refer Slide Time: 14:26)

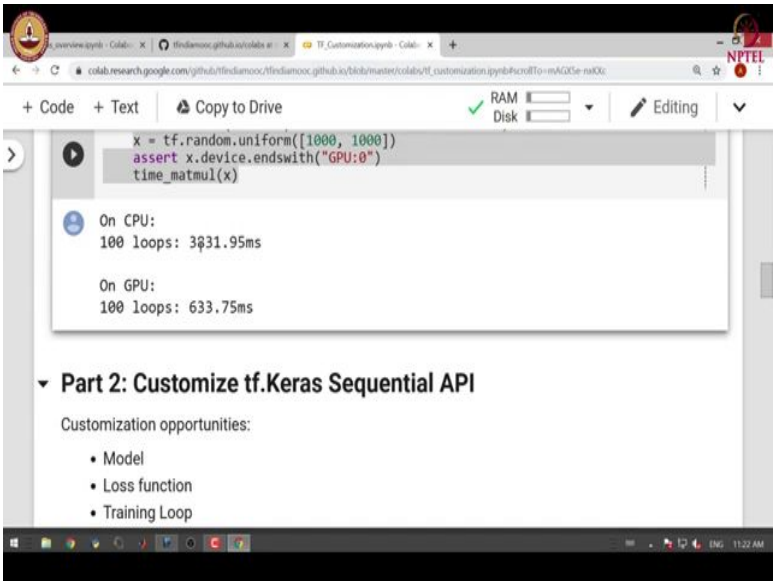


And, here what we will do is we will force this particular operation first on CPU and then on GPU. We can do this in the tf.device context. Here we define the multiplication operation in the context of CPU, we specify CPU with tf.device and the CPU ID string. So, this is the first CPU that is why it is CPU:0.

We define our matrix to be of 1000 by 1000 dimension and each element in the matrix is randomly initialized using uniform distribution. We make sure that the tensor is placed on CPU by asserting that the device associated with x is indeed CPU:0. And then we call the matrix multiplication function and also measure the time it takes to carry out multiplication of this particular matrix with itself 100 times.

We repeat the same operation with GPU. There is one difference before forcing the operation on GPU, we first check whether the GPUs are available to us. Then if GPUs are available to us, what we do is, in the context of the GPU device, we repeat the same operations as we did on CPU. So, let us run this to find out how much time it takes on CPU and GPU.

(Refer Slide Time: 16:22)



So, we can see that on CPU it took 3831.95 milliseconds whether on GPU it took around 633.75 milliseconds. So, it is almost 6 times lesser time taken on GPU to perform multiplication or the matrix to itself 100 times.