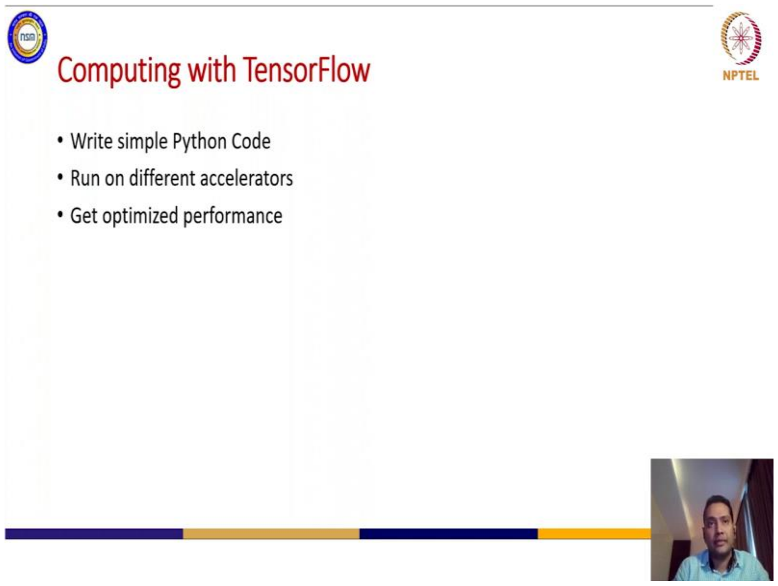
Applied Accelerated Artificial Intelligence

Dr. Satyajit Das
Department of Computer Science and Engineering
Indian Institute of Technology, Palakkad

Lecture - 28
Accelerated TensorFlow - XLA Approach Part - 1

Good evening everybody. So, today we will start with the XLA Approach. So, previously we have seen in tensor flow of course, its quite vast and in terms of flexibility control and different types of approaches that we have seen, we have seen some distribution strategies pipeline strategies and so on.

(Refer Slide Time: 00:21)



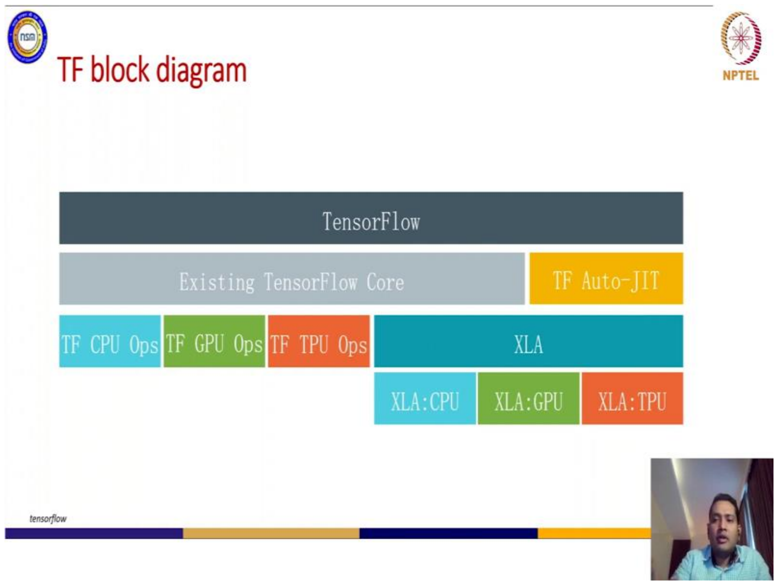
Computing with TensorFlow

- Write simple Python Code
- Run on different accelerators
- Get optimized performance

So, the basic flow is write simple Python code run on different accelerators depending on different targets that you may have and you get optimized performance with different approaches right. So, but today we will do something extra something bit different from the traditional flow of tensor flow writing codes for your models and building deep neural network models for your targets.

So, here diversion is not in terms of writing the code. So, the exactly same thing whatever you have written everything will be there just in terms of the composition strategy we will be taking a different path. And that you will see is giving you much more better performance and lot more opportunities for optimization.

(Refer Slide Time: 01:41)



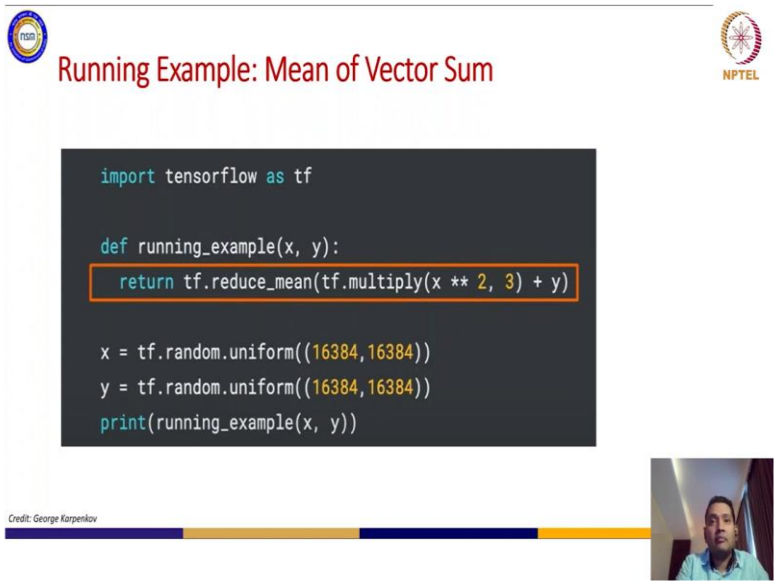
TF block diagram

So, the basic TensorFlow code if you see, so in the format you have the tensor flow where you will be building your models whatever you see actually. And in the middle layer you have the existing TensorFlow core which is written in C++ and that is being be in back end which will which is providing support for your target devices like GPUs, CPUs, TPUs and multiple devices with multiple nodes and so on.

So, this is the traditional TensorFlow compilation and here now you will try to add one more layer in this which is the TensorFlow auto JIT. Which will actually compile your TensorFlow core for your XLA devices, this is one another layer of made up device. So, this is basically the target for this JIT compiler will be XLA device which is very and highly optimized device in terms of operations.

So, you will see that the TensorFlow which supports maybe thousands of operations, but XLA device supports very limited number of operations and those are very very highly optimized. And that is how you will get the optimized performance for the target which will actually the converted code or code generated by the XLA compiler will be for the target devices like CPU GPU and TPU and so on.

(Refer Slide Time: 03:22)



Running Example: Mean of Vector Sum

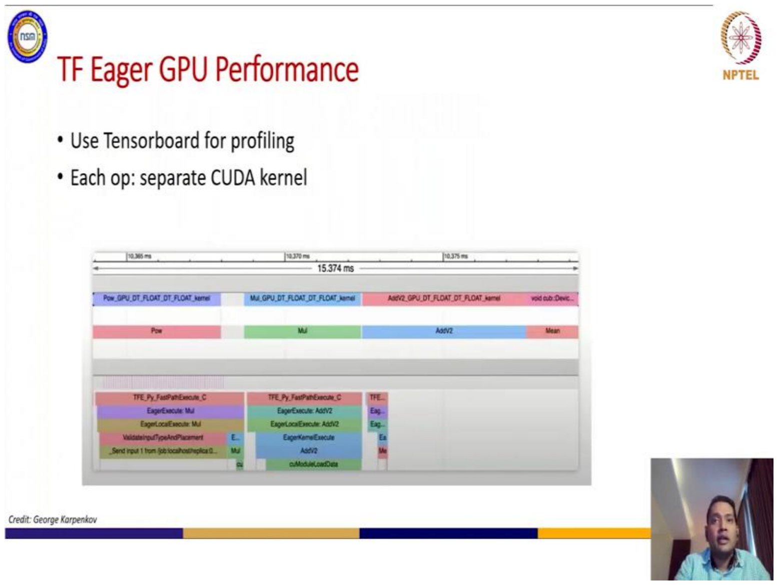
```
import tensorflow as tf

def running_example(x, y):
  return tf.reduce_mean(tf.multiply(x ** 2, 3) + y)

x = tf.random.uniform((16384,16384))
y = tf.random.uniform((16384,16384))
print(running_example(x, y))
```

Credit: George Karpenkov

So, now what XLA is all about that we will see in a bit, but let us look at one traditional way of writing code and inter interfering it how it is going to be executed. So, as you can see here one code is there where you are actually multiplying this power of 2 into 3. So, basically you are operating on these 2 tensors which is x and y and they are defined as let us say with these dimensions and these shapes. And then when you are running this example; that means, you are operating on x and y how that is being written here right.

Now, if it is eager execution it will be evaluated immediately from the Python byte code and if you want to optimize the performance again, so then you will you would like to add one wrapper before that, but that wrapper is we will see.

(Refer Slide Time: 04:18)



TF Eager GPU Performance

- Use Tensorboard for profiling
- Each op: separate CUDA kernel

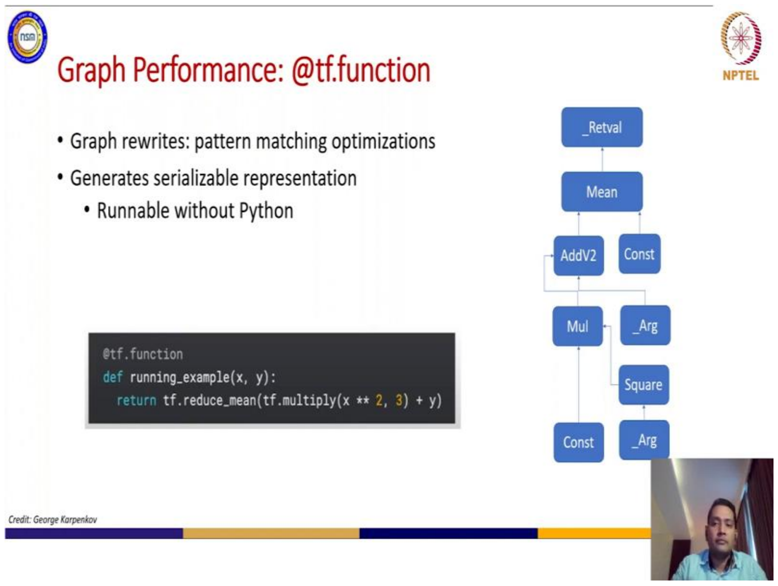
Credit: George Karpenkov

So, when you are compiling this code is essentially you can see that there are several operations. One is this power operation, one is multiply operation, one is addition operation right, one is mean operation and so on. So, let us say four operations we have. And if you are compiling for your target GPU let us say. So that means, you are actually generating four kernels for the GPU target GPU, so that you are seeing here. So, one is for power, one is for multiplication, one is for addition and mean and so on.

So, these four kernels you are actually generating so; that means, your code is essentially converted into your target GPU. And each operation is essentially the CUDA kernel. So, basically you underneath your CUDA C++ is been generating the code for you. So now, what is the concern here? The concern here is the number of kernels that you are generating for the simple one line code you are generating four kernels.

So, imagine if you are doing one TensorFlow let us say model training how many number of operations that you will generate and those many number of kernels and they will be called again and again. You can think about the complexity that you are going to burden with the target GPU that yeah.

(Refer Slide Time: 05:58)



Graph Performance: @tf.function

- Graph rewrites: pattern matching optimizations
- Generates serializable representation
    - Runnable without Python

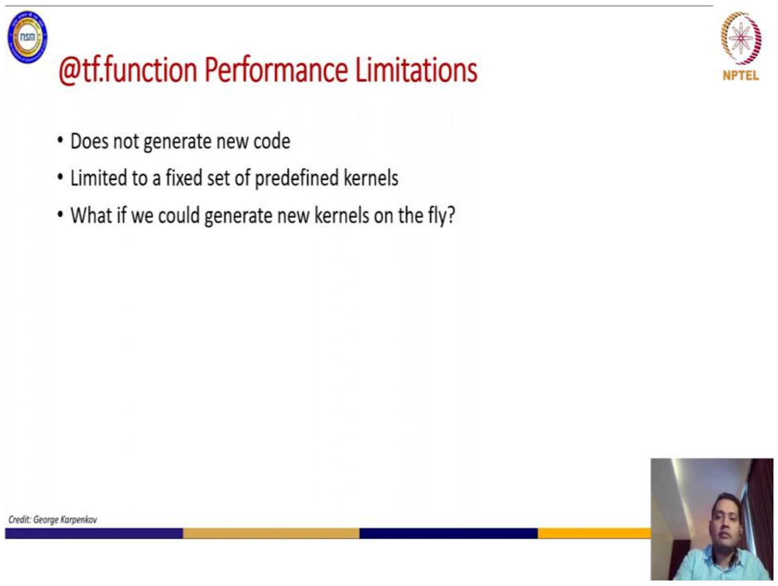
```
@tf.function
def running_example(x, y):
  return tf.reduce_mean(tf.multiply(x ** 2, 3) + y)
```

Credit: George Karpenkov

So, but we want to do is that we of course, first optimization is you add this tf.function one time. So, what it will be do? It will serialize the core basically you do this wrapper create this wrapper for at tf.function and you define that function inside the wrapper and that will actually generate the target computational of ok. So, this is your graph as you can see square multiplication addition mean all these operations making up the graph right.

And the performance gain is in terms of you are not running anymore in the Python front. So, and you can run it anywhere then right you can deploy anywhere this graph. So, that way you are getting some optimization with the tf.function, but it is not enough you still have the number of operations that are actually being generated as kernels inside your code.

(Refer Slide Time: 07:00)


@tf.function Performance Limitations

- Does not generate new code
- Limited to a fixed set of predefined kernels
- What if we could generate new kernels on the fly?

Credit: George Karpenkov

So, the limitations are the function tf.function run time is not generating new codes actually, the essentially the codes are the primitive ones right. And it is limited to fixed number of predefined kernels, so whatever number of kernels you have and those kernels are from the library and they will take the operations from the kernels and they will make this.

Now, what if you can generate new optimized code on the fly depending on the targeting right. So, new optimized code on the fly. So, there are three more concerns here after the graph code generated, new code generated for new kernels. So that means, you need to somehow optimize those kernels into a new kernel into a new set of kernel.

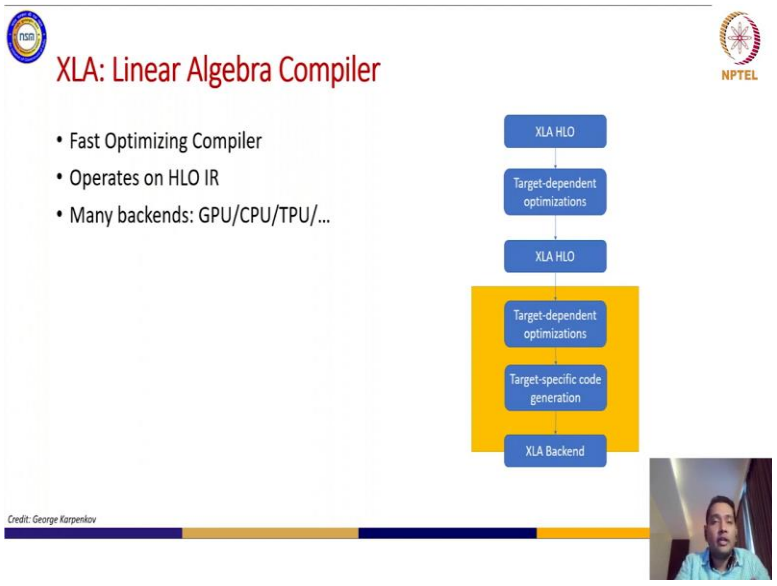
So that means, you need some new compilation right and on the fly JIT compilation will give you on the fly and code new code generate will be happening on the fly for your target lines.

(Refer Slide Time: 08:12)

XLA Compiler tensorflow.org/xla

So, that is where this XLA compiler which is for still it is in experimental phase and lots of research is going on, but the stable version is with us. So, we can use that. So, XLA compilation flow I mean flow we will see, but XLA compiler is the answer which we were talking about the limitations of tf.function.

(Refer Slide Time: 08:42)


XLA: Linear Algebra Compiler

- Fast Optimizing Compiler
- Operates on HLO IR
- Many backends: GPU/CPU/TPU/...

XLA HLO
Target-dependent optimizations
XLA HLO
Target-dependent optimizations
Target-specific code generation
XLA Backend

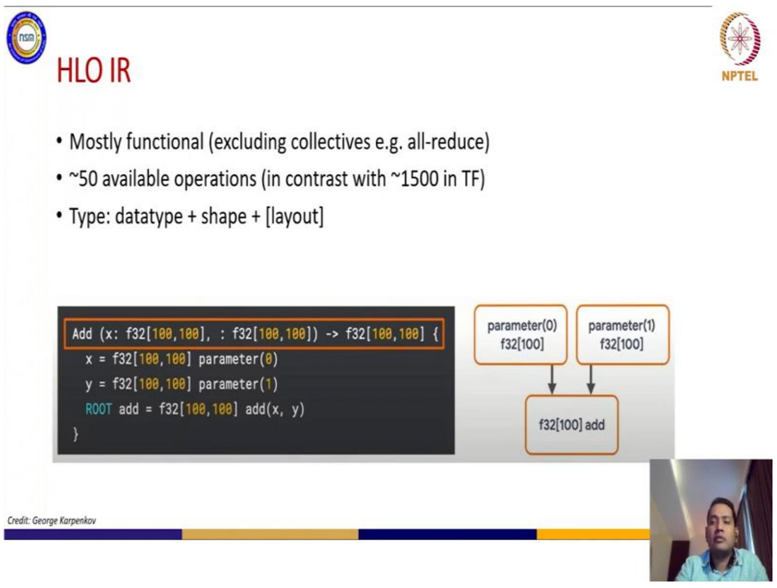
Credit: George Karpenkov

So, what is happening here? The XLA compilation flow is taking one high level optimization core which is XLA HLO ok XLA HLO. By the way the XLA stands for accelerated linear algebra. So, most of the operations you are seeing inside generating one model is basically your linear algebraic operations and optimization for these operations. So, accelerated linear algebra compiler and. So, for this compiler the input for this compiler is HLO which is high level optimized XLA.

And depending on the target dependent optimizations it will again generate another set of XLA HLO and this will in turn get into converted into a code generated for the XLA back and then XLA back end will actually provide the kernels for the target GPU, TPU and CPU. So, operates on the HLO IR, but how this HLO IR actually looks and its fast enough to not to get noticeable for large models ok.

So now, how these optimizations are happening just a bit of theory we will see because we need to learn what is happening behind the scene actually. Because the way you will apply this linear algebraic compiler or XLA just in time compiler, there are many options to be to use it in many several environments. And actually where you will use that to understand this you need to understand a bit basics like how it is optimizing and things. So, that we will talk about and we will directly go into how we can implement this essentially right.

(Refer Slide Time: 10:42)



HLO IR

- Mostly functional (excluding collectives e.g. all-reduce)
- ~50 available operations (in contrast with ~1500 in TF)
- Type: datatype + shape + [layout]

```
Add (x: f32[100,100], : f32[100,100]) -> f32[100,100] {
  x = f32[100,100] parameter(0)
  y = f32[100,100] parameter(1)
  ROOT add = f32[100,100] add(x, y)
}
```

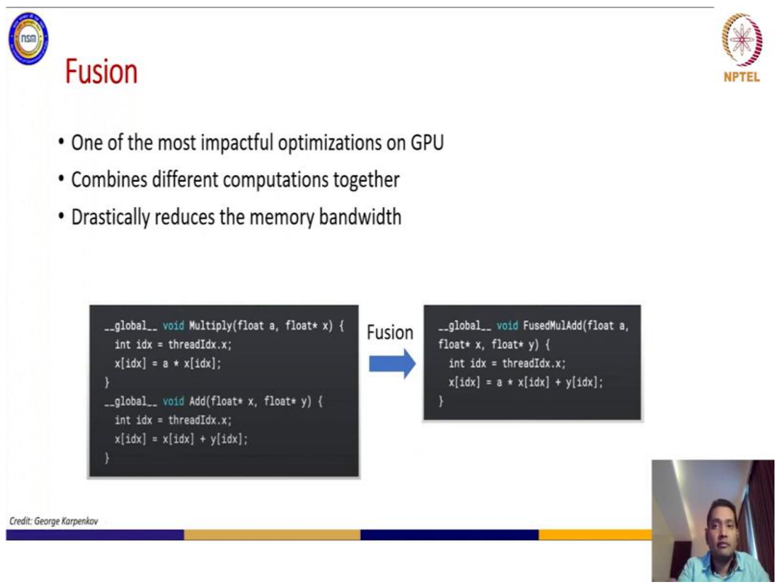
Credit: George Karpenkov

So, the HLO IR or HLO intermediate representation whatever you call it is basically functional in nature ok of course, there are some limitations in terms of all reduce algorithms that we have seen in the previous class that is not possible. So, that is actually not compatible for HLO IR. And the number of operations that I was talking about is very very less in terms in contrast with the TensorFlow operations that are.

So, if you see total amount of tensor flow operations 1500 TensorFlow operations are there, in HLO IR you have 50 operations are there around 50. And its strictly typed data type plus shape should be static in nature. So, now as you can see I was talking about these 50 operations so; that means, these operations are highly optimized which of course, are subset of these 1500 operations and all the operations are not actually compilable in XLA.

So, that is where you need to know which are the operations you are going to compile, to get it optimized for your target model training.

(Refer Slide Time: 12:00)



Fusion

- One of the most impactful optimizations on GPU
- Combines different computations together
- Drastically reduces the memory bandwidth

```
__global__ void Multiply(float a, float* x) {
  int idx = threadIdx.x;
  x[idx] = a * x[idx];
}
__global__ void Add(float* x, float* y) {
  int idx = threadIdx.x;
  x[idx] = x[idx] + y[idx];
}
```

Fusion

```
__global__ void FusedMulAdd(float a,
float* x, float* y) {
  int idx = threadIdx.x;
  x[idx] = a * x[idx] + y[idx];
}
```

Credit: George Karpenkov

So, let us talk about the fusion operation here because this is how the new operations or new kernels being generated. And the target kernels will be generated by this fusion operation and the fusion operation is essentially drastically reducing the memory bandwidth, how? Let us say.

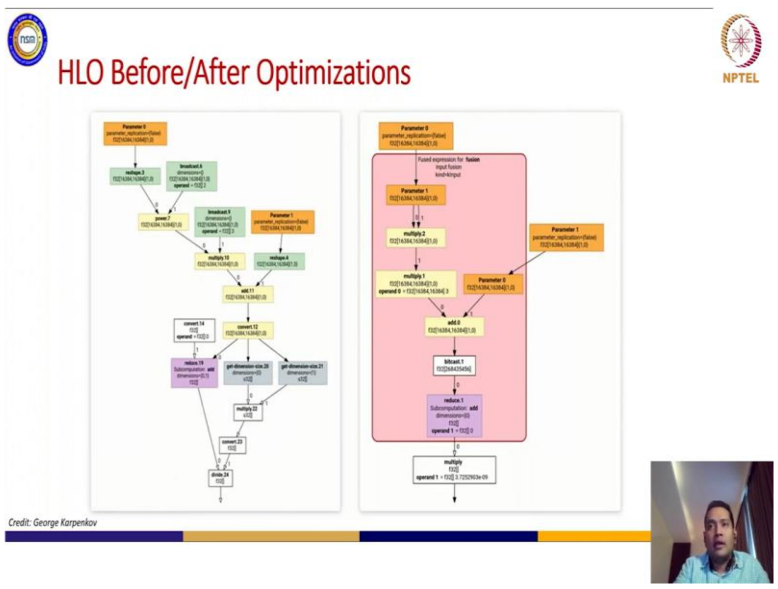
So, from this previous example we have this multiply add operations and these kernels are being generated right. So, in this multiply kernel you can see two read operations were there and here one read one write and here two read and one write operation right.

So, in total you have three read operations two write operations to execute these two kernels.

Now if we infuse these two operations inside your compilation flow into a let us say fused mul add ok where you just have these two instead of three read operation and one two write operation. Now, you have two read operations and one write operation. So, you can see from this very simple example of just using this multiply and add operation of course, they are actually based on vectors ok, it is not a simple multiply and addition operation.

But from this simple multiplier or addition operation you can see how many number of read and write instructions or operations we are getting reduced and you can imagine for how many operations we can reduce in a full-fledged tensor flow model training. So, right this is the fused operation and this is the generated kernel that we want to generate from the XLA compilations right the fused operations.

(Refer Slide Time: 13:56)



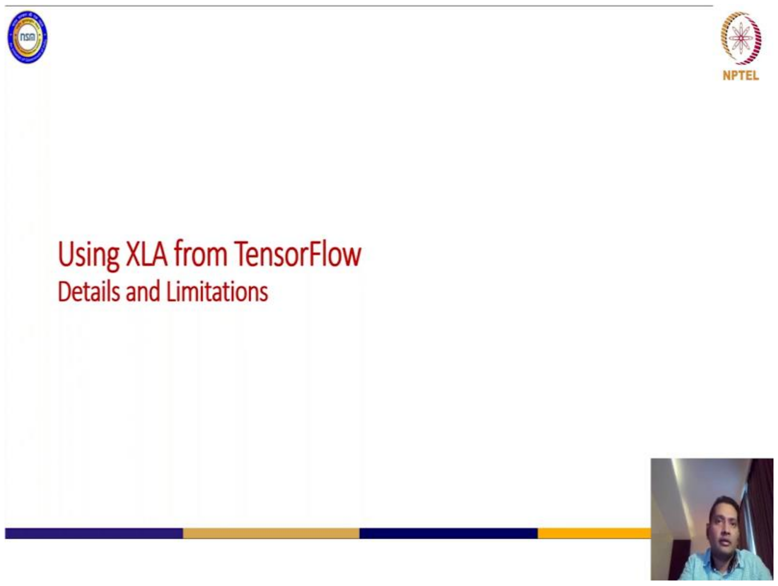
HLO Before/After Optimizations

Credit: George Karpenkov

So, if you see the IR before fusing a ok. So, basically this is the IR before going into the XLA compilation flow. So, this is very complex operation all the complex computational flow you have this complex graph. And after fusing all the operations which are actually capable of fusing, it will automatically analyze and you will fuse the those operations and you will have this fused graph.

So, you can see for a full fledged training pipeline you can fuse those operations and you can generate new graph for that and which will be optimized highly optimized. And will be compatible for your target devices as well as you can see that if you want to store the model if you have fused operations. Now how much storage efficiency you are getting in terms of storing the model itself also. So, for memory constraint devices storing the model will also be helpful using this XLA compiler.

(Refer Slide Time: 15:17)

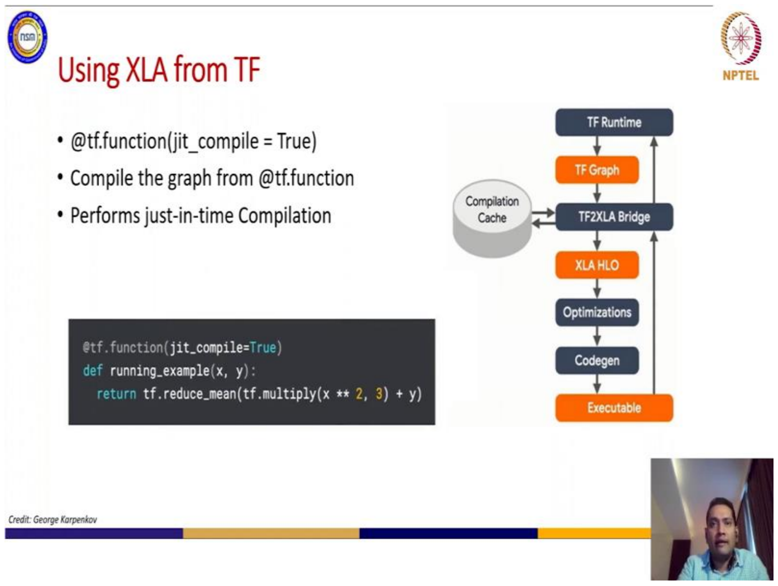


Using XLA from TensorFlow

Details and Limitations

Now, what are the details and how we will use that inside a TensorFlow let see that and also we will discuss a bit of limitations that will give you a flavour of the research that is going on still. So, we will see how to use that and part of the limitations in a compiler.

(Refer Slide Time: 15:37)



Using XLA from TF

- @tf.function(jit_compile = True)
- Compile the graph from @tf.function
- Performs just-in-time Compilation

```
@tf.function(jit_compile=True)
def running_example(x, y):
  return tf.reduce_mean(tf.multiply(x ** 2, 3) + y)
```

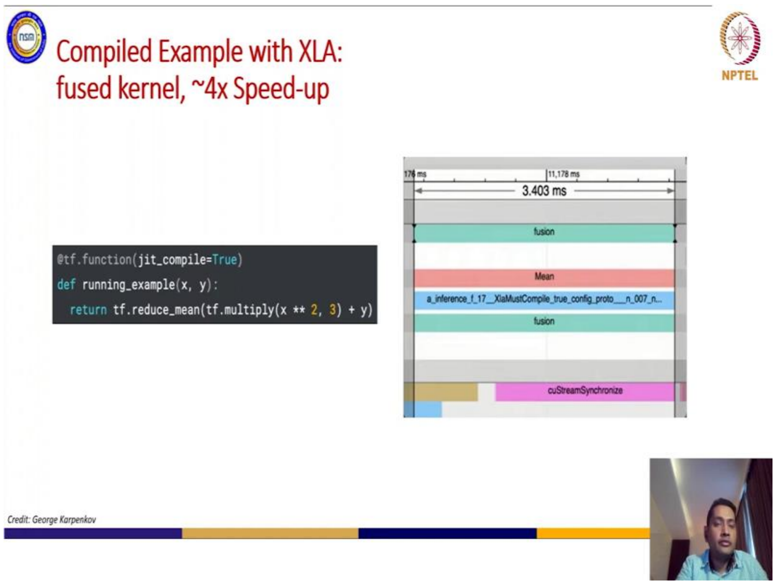
Credit: George Karpenkov

So, as I was mentioning that tf.function is essentially serializing the computational graph for your TensorFlow model any TensorFlow, code for that matter. And inside the jit compile 2 if you make it True so; that means, you are enabling the XLA compiler ok. And the graph will now whatever has been generated by this tf.function runtime will be actually compiled; that means, converted into HLO IR then all the optimizations will happen and then you will generate the code for the target XLA device. And it performs just in time compilation, so it is on the fly.

So, how it happens? Basically you are trying to start from the tf function which is the tensor flow tf.function run time you have the graph, you have the bridge. Basically the compilation cache will actually do the bridging because you will see, in few slides that this compilation cache which is generating the XLA HLO actually making the bridge between the graph and the HLO.

That will happen in the cache and depending on the availability of already compiled code or already code generated XLA HLO, you will not be generating new course for that. So, its highly efficient in that sense. Then you do the optimizations code generation and produce the executions. So, that is the simple flow that we have.

(Refer Slide Time: 17:32)



Compiled Example with XLA: fused kernel, ~4x Speed-up

```
@tf.function(jit_compile=True)
def running_example(x, y):
  return tf.reduce_mean(tf.multiply(x ** 2, 3) + y)
```

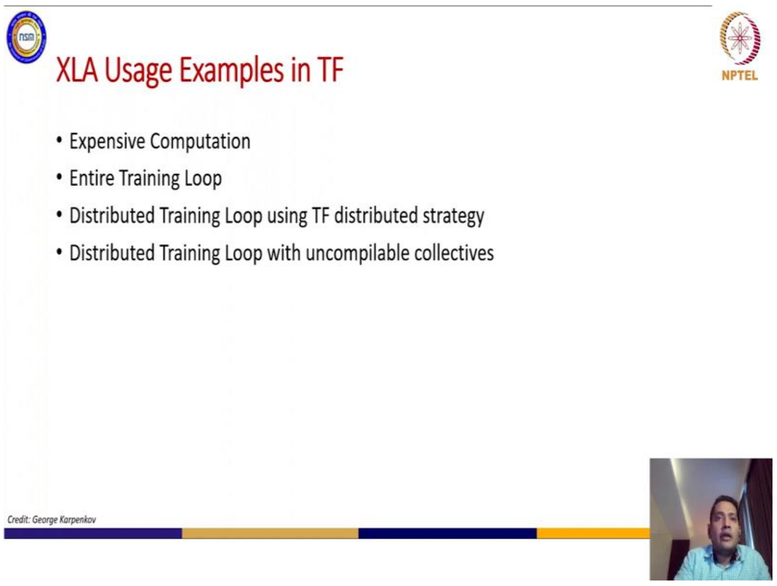
176 ms
11,178 ms
3.403 ms
fusion
Mean
a_inference_f_17__XlaMustCompile_true_config_proto___n_007_n...
fusion
cuStreamSynchronize

Credit: George Karpenkov

So, now once you have enable the tf.function runtime as well as jit compilation equal to True for this particular function that we were trying to. So, we are we have now apply the function with the runtime as well as we have enable the compilation for the jit compilation True for this particular function.

So, now, we have generated the fused operation which was earlier four operations ok four operations four kernels basically and now we have one kernel and you can see that how much time you have reduced. So, for this simple example it was around three times three to 5 times right. So, yeah on an average four times speed up you will get right for the simple fusing of operations.

(Refer Slide Time: 18:23)


XLA Usage Examples in TF

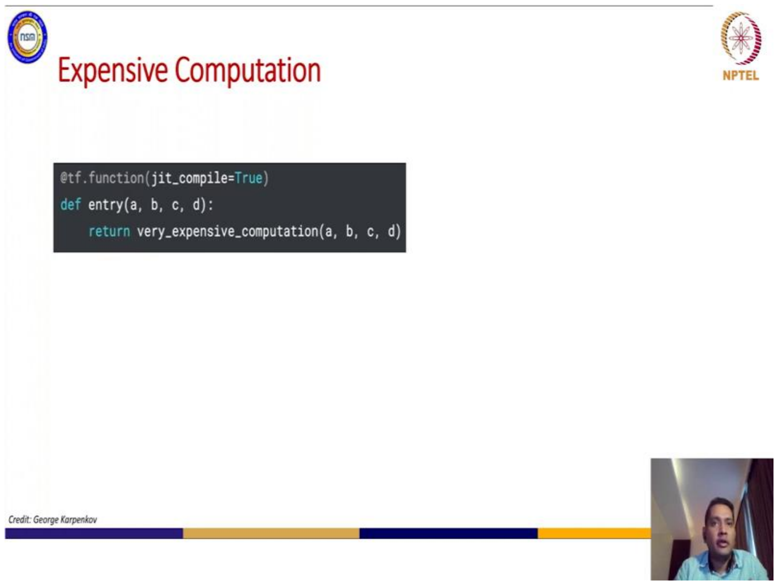
- Expensive Computation
- Entire Training Loop
- Distributed Training Loop using TF distributed strategy
- Distributed Training Loop with uncompilable collectives

Credit: George Karpenkov

So, now what are the computations you want to fuse of course, the expensive ones and you can fuse the entire training loop right, you can apply the JIT compilation True for your distributed training loop. But of course, for now it is only supporting the mirrored strategy, but we will see and also you can distribute you can apply the distributed training loop with uncompilable collectives kernels.

So, some of the kernels or some of the compiling collectives are not compilable because it has very less number of operations that we have seen right around 50 operations compared to your 1500 operations in tensorflow. So, all the operations will not be compilable. So, some of these operations might be uncompilable and when you are distributing the training loop there are ways how you can actually specify some of the part of your training strategy or distribution strategy to be a compiled by the XLA and not to be by the compiler itself. So, its amazing.

(Refer Slide Time: 19:43)



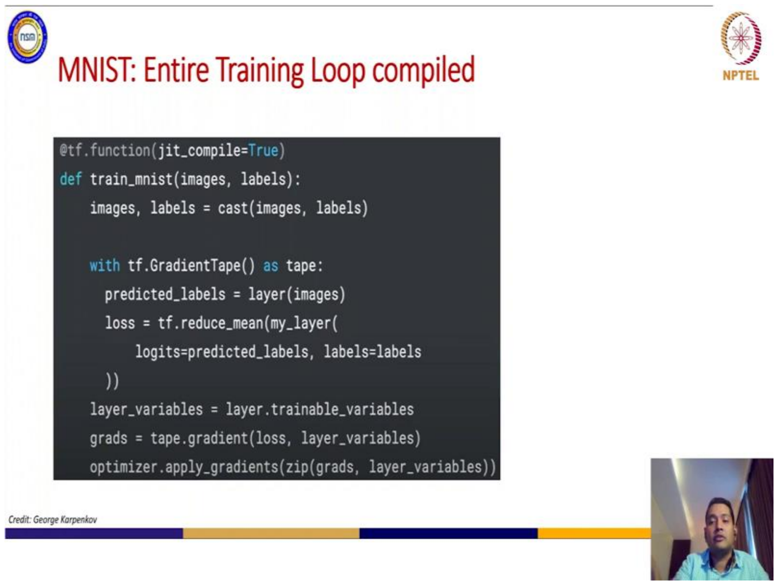
Expensive Computation

```
@tf.function(jit_compile=True)
def entry(a, b, c, d):
    return very_expensive_computation(a, b, c, d)
```

Credit: George Karpenkov

So, expensive computation, so let us say we have this expensive computation as very expensive computation function right. We can define one function and wrap it with function runtime and in jit compilation True as simple as that. You can have entire function fused and generated the optimized graph for that.

(Refer Slide Time: 20:11)



MNIST: Entire Training Loop compiled

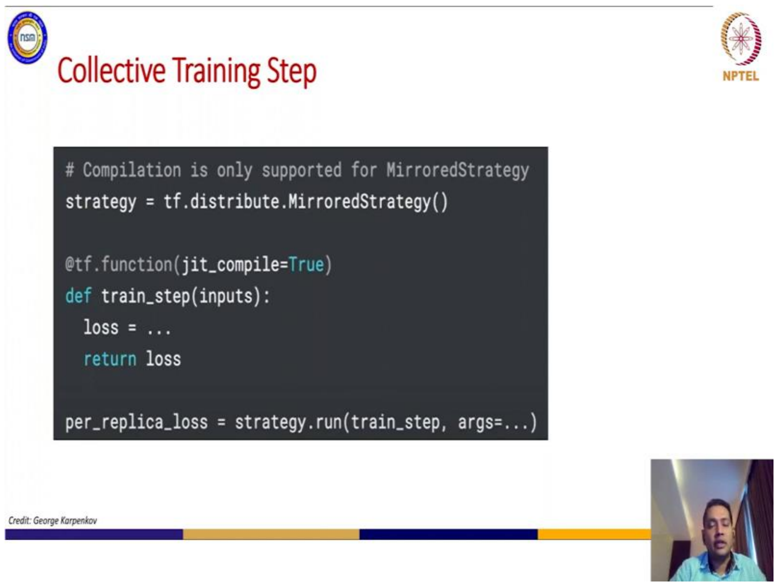
```
@tf.function(jit_compile=True)
def train_mnist(images, labels):
    images, labels = cast(images, labels)

    with tf.GradientTape() as tape:
      predicted_labels = layer(images)
      loss = tf.reduce_mean(my_layer(
          logits=predicted_labels, labels=labels
      ))
    layer_variables = layer.trainable_variables
    grads = tape.gradient(loss, layer_variables)
    optimizer.apply_gradients(zip(grads, layer_variables))
```

Credit: George Karpenkov

You can have entire training loop as I was mentioning. So, now entire training loop if you want to do that you can define one training let us say module for that right. And in this function you have the training basically generating the gradients and updating the gradients and so on and so forth. So, the entire loop here you are actually wrapping it with a tf.function and jit compilation. So, that is one case.

So, complex operation in terms of time or expensive operations or expensive functions rather the inter training loop you can wrap it with.

(Refer Slide Time: 20:56)



Collective Training Step

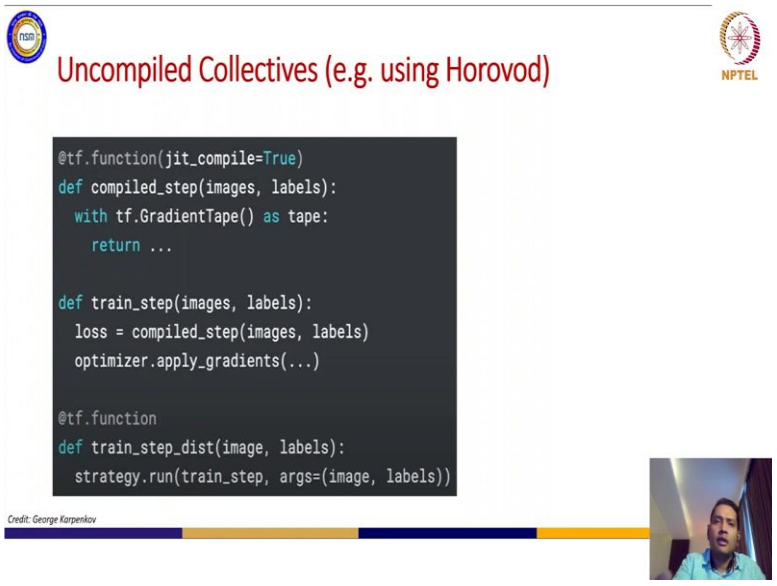
```
# Compilation is only supported for MirroredStrategy
strategy = tf.distribute.MirroredStrategy()

@tf.function(jit_compile=True)
def train_step(inputs):
  loss = ...
  return loss

per_replica_loss = strategy.run(train_step, args=...)
```

Credit: George Karpenkov

Now, next the distribution strategy if you want to distribute the training loop that also you can do. So, defining strategy how you have to define the strategy we have seen in the previous class and this strategy was for the mirrored strategy. So, we have studied several other strategies also, but for this is only applicable for the mirrored strategy and.

You can see that the inter training step is wrapped with this function run time and when you are actually calling these train step under this strategy. So, that will be essentially compiled to your XLA using XLA compiler.

(Refer Slide Time: 21:44)



Uncompiled Collectives (e.g. using Horovod)

```
@tf.function(jit_compile=True)
def compiled_step(images, labels):
  with tf.GradientTape() as tape:
    return ...

def train_step(images, labels):
  loss = compiled_step(images, labels)
  optimizer.apply_gradients(...)

@tf.function
def train_step_dist(image, labels):
  strategy.run(train_step, args=(image, labels))
```

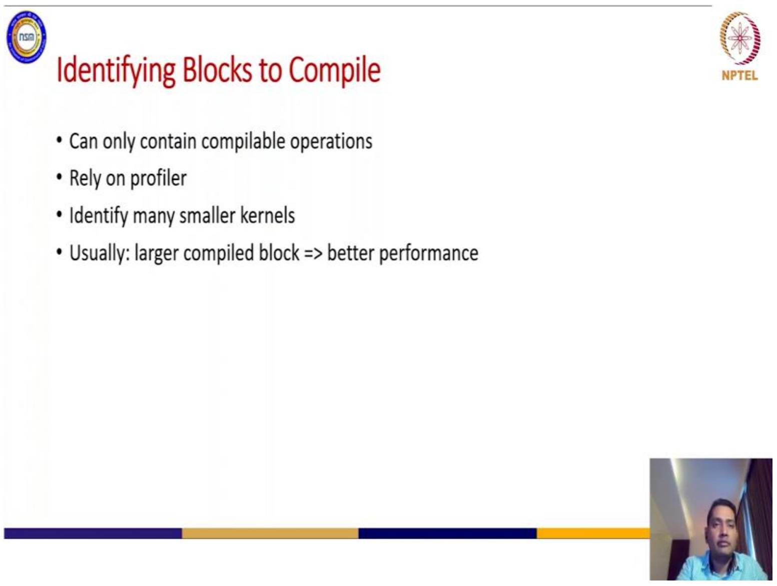
Credit: George Karpenkov

Now, this is the entire training loop right and in some distribution strategies like pro mode right. So, where some of the steps let say optimizer update right gradient updates are not supported not compiler. So, what we will do in that sense? So, what its simple again in step essentially is a loop for your training calculating loss is supported in your XLA compiler and you define one function for that compile step where you will computing the gradients. So, all these backward paths everything is happening there.

Now, in this distribution strategy applying gradients which is the updatation stage for your parameters is not supported. So, you just keep it out of that function and when you are defining that strategy use that strategy with this train step and it will automatically keep this graph out of that applied gradients pipeline. So, you can see there is flexibility to define its usage as you wish.

So, basically you can define which functions are going to be compiled with XLA and which functions will not. But you can use both you can use both together in inside your training pipeline.

(Refer Slide Time: 23:13)



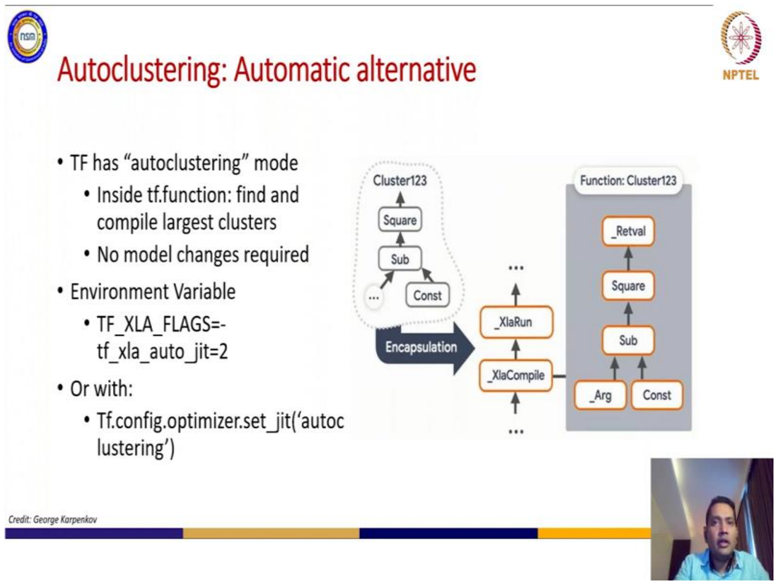
Identifying Blocks to Compile

- Can only contain compilable operations
- Rely on profiler
- Identify many smaller kernels
- Usually: larger compiled block => better performance

So, that is very simple, but is actually bit difficult is to identify the blocks which will be actually compiled in XLA. Of course, there will be some blocks which will be uncompilable and some blocks which are compilable. A very nice strategy is you can apply anywhere ok you can apply tf.function wrap it with any function that you want to accelerate, you do just in time compiler true. If that is uncompilable that will give you an error that is the simple way or you can rely on the profiler also.

So, you can rely like let say these many kernels you have you want to fuse them ok. So, these kernels are getting bottleneck by the function calls and so, you can use tensor profiler and you can see the kernels which are getting bottleneck any profiler you can use dl graph also you can use tensor support you can use. But the main thing is that you can identify many such smaller kernels usually the larger compiler compile block is actually gives you better performance in terms of XLA compilation.

(Refer Slide Time: 24:43)



Autoclustering: Automatic alternative

- TF has "autoclustering" mode
  - Inside tf.function: find and compile largest clusters
  - No model changes required
- Environment Variable
  - TF_XLA_FLAGS=-tf_xla_auto_jit=2
- Or with:
  - Tf.config.optimizer.set_jit('autoclustering')

Cluster123
Square
Sub
...
Const
Encapsulation
...
_XlaRun
_XlaCompile
...
Function: Cluster123
_Retval
Square
Sub
_Arg
Const

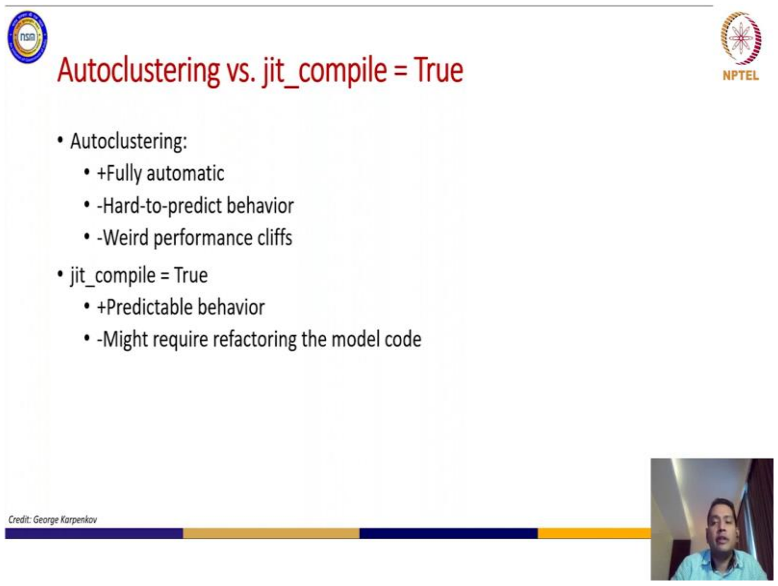
Credit: George Karpenkov

So, you can just try it out and see what will be the best for your particular training loop that you are defined. If you are not so sure, there flexibility as well that it is called auto clustering. So that means, if you keep this environmental variable TF XLA FLAGS equal to tf xla auto jit equal to 2 which is essentially enabling the auto clustering inside your environment. That means, you do not need to identify the kernels or operations to be fused and wrap it with the kernel dot runtime of course, there is another way which is tf dot optimizer dot set jit auto clustering.

It will identify, so it will analyze the entire computational graph and it will identify these are the operations that I want to cluster them and optimize and fuse them into a XLA compile. So, basically this kind of functional fusion it will do encapsulate the entire cluster. So, this kind of automatic clustering can be used, so if you want to if you are not so sure in the beginning just to see what are the kernels are being let us say optimized converted.

And fused into your optimized operations that you can check and you can explore the performance that you are again you are achieving.

(Refer Slide Time: 26:09)



Autoclustering vs. jit_compile = True

- Autoclustering:
  - +Fully automatic
  - -Hard-to-predict behavior
  - -Weird performance cliffs
- jit_compile = True
  - +Predictable behavior
  - -Might require refactoring the model code

Credit: George Karpenkov

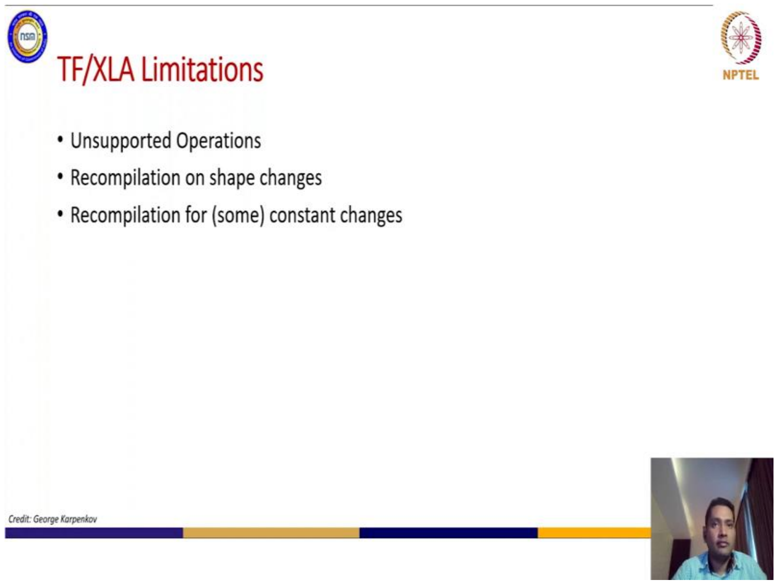
But there are some glitches here because this is fully automatic you do not know actually right. So, you can go either way you can get very high performance or maybe very low performance right. It is hard to predict the behavior now what to predict behavior means if you change a simple line or any even if one parameter inside your training loop and you have enabled the auto clustering. That means, each time you are running that even if you have made a very simple change it can make drastic change inside your target compilation.

So, you might get some weird performance clips for that matter. So, its better if you just know the function itself you can just wrap it with jit compilation True and use the predictable behaviour, whatever you can have you have the optimized performance for this particular function or you will get error if its not compilable.

Now, not compilable means let say you are loading some data from your resource maybe local disk or remote sources. And in the process of loading you are actually let say decoding something or maybe applying some image processing which is let us say image encoding decoding which is not particularly supported in GPUs or maybe XLA target device right.

So, in that case XLA will return you an error and that will actually be uploaded to the CPU part right. So, its automated. So, you just see the error and change your jit compilation True function wrap up. But anyway I mean you get some predictable behaviour that it will get this kind of output.

(Refer Slide Time: 28:16)



TF/XLA Limitations

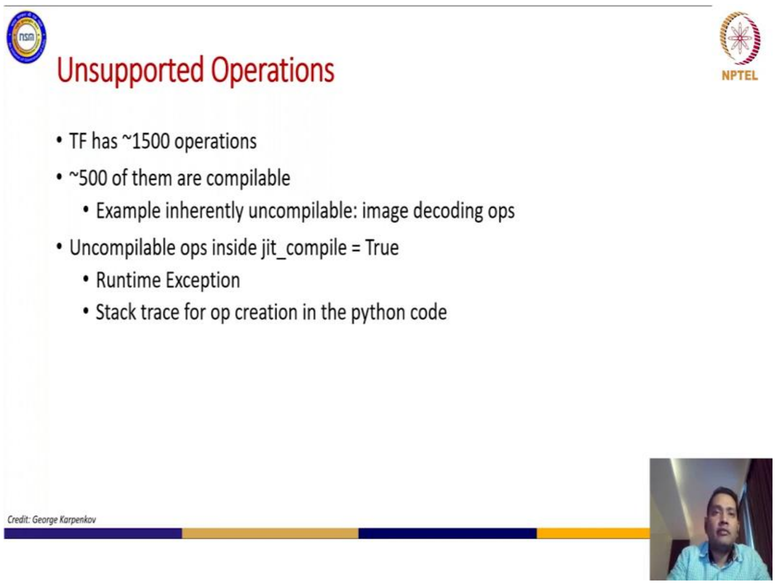
- Unsupported Operations
- Recompilation on shape changes
- Recompilation for (some) constant changes

Credit: George Karpenkov

So, but there are some limitations that you need to know because you need to know some unsupported operations as I was mentioning that image encoding decoding which is not supporting in XLA compilation. And recompilation will happen every time you change the shapes because it is actually based on the cache based IR generation.

So, we will see one example such example and that will actually reduce the performance that you want to achieve recompilation will happen also if you have some constant changes. So, recompilation means, so its on the fly compilation. So, recompilation means it will take a bits time if you are; if you are changing something and recompiling one such example that you will see here.

(Refer Slide Time: 29:14)



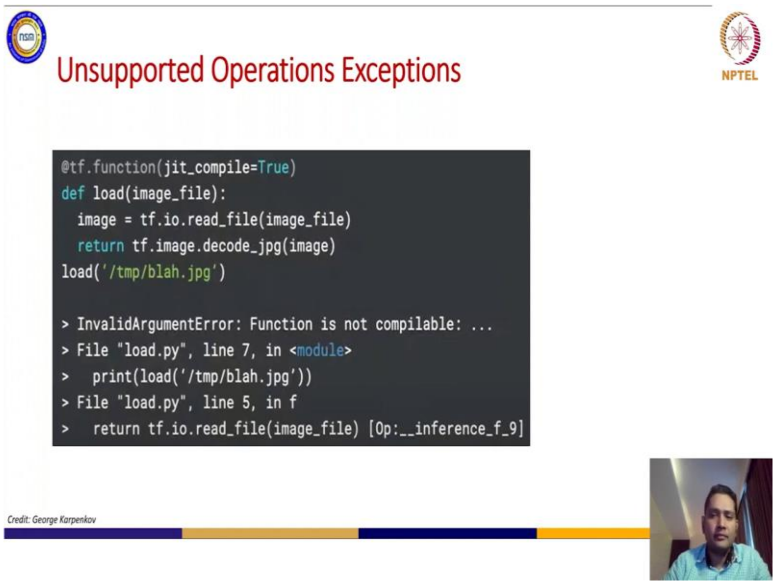
Unsupported Operations

- TF has ~1500 operations
- ~500 of them are compilable
    - Example inherently uncompilable: image decoding ops
- Uncompilable ops inside jit_compile = True
    - Runtime Exception
    - Stack trace for op creation in the python code

Credit: George Karpenkov

Let say you have well there may be unsupported operations may be many of them image decoding operations are not supported and. So, you will get run time exception as I was mentioning that you will get an error and you can press that wherever it is happening and you can change that for your target XLA compilation.

(Refer Slide Time: 29:43)



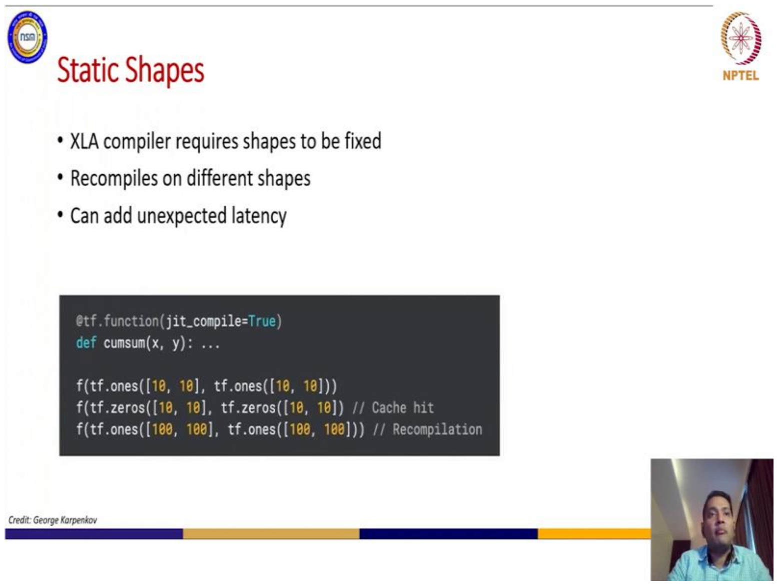
Unsupported Operations Exceptions

```
@tf.function(jit_compile=True)
def load(image_file):
  image = tf.io.read_file(image_file)
  return tf.image.decode_jpg(image)
load('/tmp/blah.jpg')

> InvalidArgumentError: Function is not compilable: ...
> File "load.py", line 7, in <module>
>   print(load('/tmp/blah.jpg'))
> File "load.py", line 5, in f
>   return tf.io.read_file(image_file) [Op:__inference_f_9]
```

Credit: George Karpenkov

So, you will get some error like this. So, you will just return that this is the error and this is the reference basically the image decode that you are using to process this data or while loading the data ok. So, this process you wanted to optimize with this tf.function run time and this operation is not supported it will give an error and you know that this is not supported if you do not know of course, you will get a list of operations that is supported in TensorFlow dot org and you will go to the XLA and or the HLO optimizer or ir what is what are the operations that are supported that you can get a list.

(Refer Slide Time: 30:36)



Static Shapes

- XLA compiler requires shapes to be fixed
- Recompiles on different shapes
- Can add unexpected latency

```
@tf.function(jit_compile=True)
def cumsum(x, y): ...

f(tf.ones([10, 10], tf.ones([10, 10]))
f(tf.zeros([10, 10], tf.zeros([10, 10]) // Cache hit
f(tf.ones([100, 100], tf.ones([100, 100])) // Recompilation
```

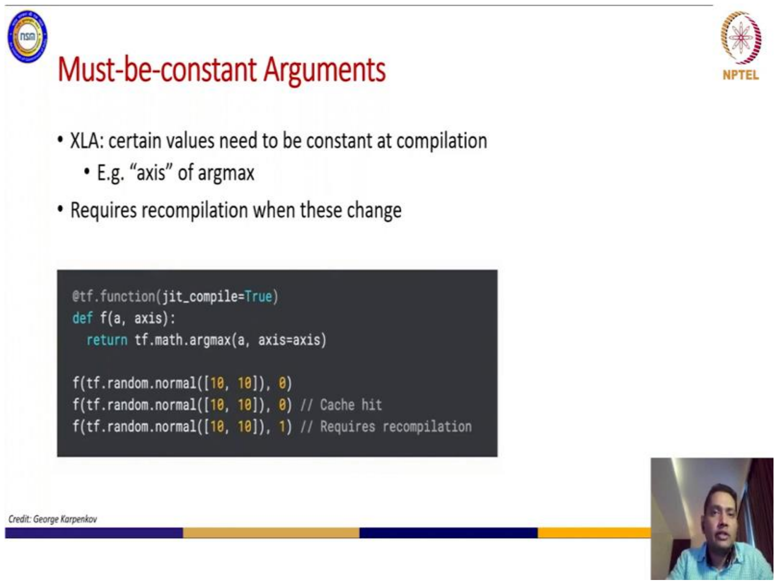
Credit: George Karpenkov

Well, so now when what are the pieces when you will get some performance glitches when you have static shapes change. So, basically if you see that this basically this function this cumulative sum ok and this function is targeting these vectors right or these tensors. And now you have the shapes of these tensors 10 10 10 10 in the next computation also or next computation graph you can see, Is the same dimension 10 10 10 10 so; that means, it will not again compile for this particular section.

It will use already cached IR generated and it will generate the code for you because it has all the dimensions same it has because it was typed strictly typed XLO XLA HLO is strictly typed and shaped also must be static. So, if you have the same shapes you your code will not be recompiled and if you have changed. So, basically in the third operation you can see once the number of ones are getting increased here the bound is getting increased.

So that means, it is no longer supported the previous are IR is no longer supported and you need to generate new IR. So, you will again recompile. So, basically it will take some more time you expected that it will happen because all the things are supported. So, basically yes supported compilable, but due to this shape changes inside your code section it is taking actually bit of more time because of recompilation.

(Refer Slide Time: 32:43)



Must-be-constant Arguments

- XLA: certain values need to be constant at compilation
  - E.g. "axis" of argmax
- Requires recompilation when these change

```
@tf.function(jit_compile=True)
def f(a, axis):
  return tf.math.argmax(a, axis=axis)

f(tf.random.normal([10, 10]), 0)
f(tf.random.normal([10, 10]), 0) // Cache hit
f(tf.random.normal([10, 10]), 1) // Requires recompilation
```

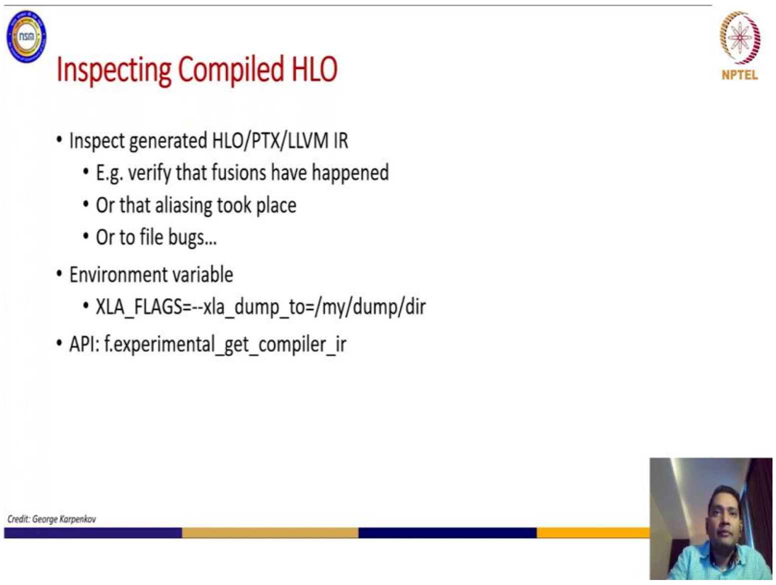
Credit: George Karpenkov

Such another recompilation scenario may appear if you have constant argument changes. Basically you can see that in this function you have two arguments one tensor size of 10 10 and 0 10 10 0. So, again so for this IR is there in the cache again. So, it will not generate and for that you will see that one more change is there in the constant and it needs recompilation.

So, basically. So, such functions where we are actually comparing let say which is the maximum during this given the tensor here ok and. So, you are trying to figure out what are the things happening because of which we are getting performance degradation, that is because your something might be wrong with different constant changes inside your code.

So, these are the limitations that you can see. So, of course, researchers are trying to figure it out how to actually pipeline these transformations and make predictions like what can come in the next and do introduce some kind of dynamic nature, because that is very very important. So, some research is going on in towards that direction.

(Refer Slide Time: 34:15)



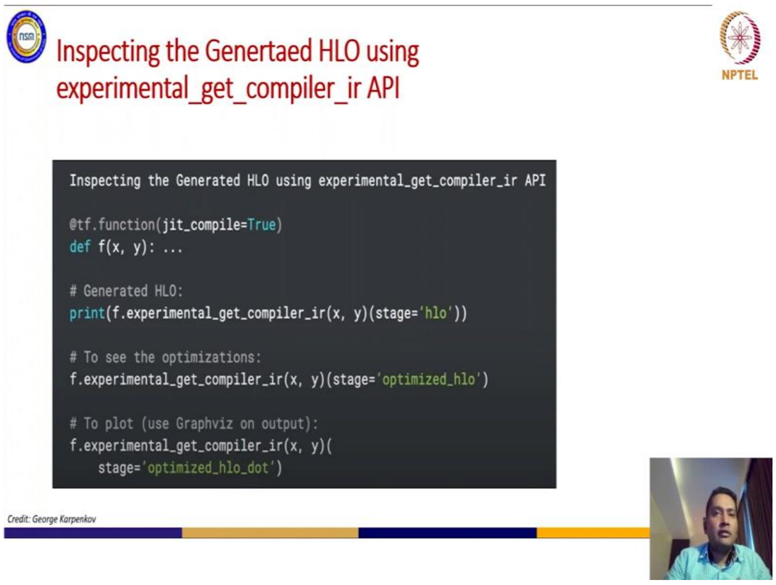
Inspecting Compiled HLO

- Inspect generated HLO/PTX/LLVM IR
  - E.g. verify that fusions have happened
  - Or that aliasing took place
  - Or to file bugs...
- Environment variable
  - XLA_FLAGS=--xla_dump_to=/my/dump/dir
- API: f.experimental_get_compiler_ir

Credit: George Karpenkov

Now, you have the flexibility to inspect your code what is happening actually you can enable this flag ok. So, you can inspect part of the generated HLO basically generated graph or infused graph that you have after the optimization what are the graph that is available both we can expect and see what are the things going on inside your code for your target model.

(Refer Slide Time: 34:51)



Inspecting the Genertaed HLO using experimental_get_compiler_ir API

```
Inspecting the Generated HLO using experimental_get_compiler_ir API

@tf.function(jit_compile=True)
def f(x, y): ...

# Generated HLO:
print(f.experimental_get_compiler_ir(x, y)(stage='hlo'))

# To see the optimizations:
f.experimental_get_compiler_ir(x, y)(stage='optimized_hlo')

# To plot (use Graphviz on output):
f.experimental_get_compiler_ir(x, y)(
    stage='optimized_hlo_dot')
```

Credit: George Karpenkov

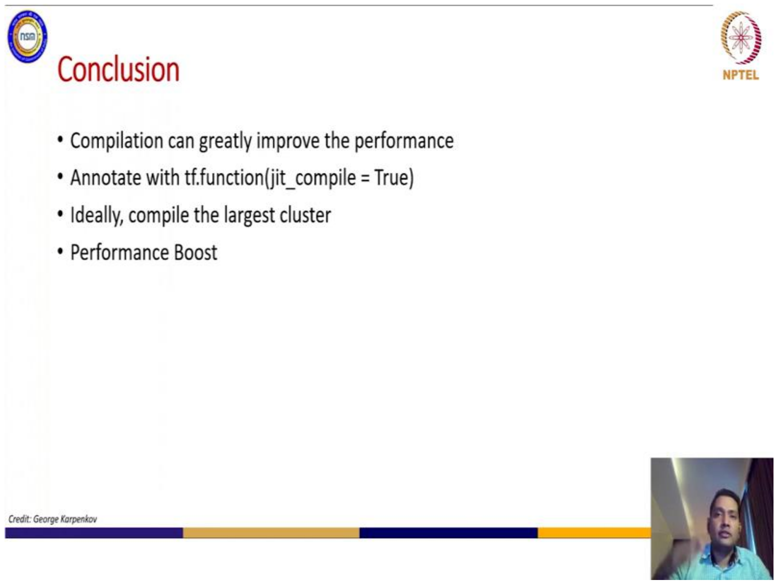
Basically, this is one example where you can apply this. So, basically here defining one function which is wrapped in the function runtime which jit compile True and we are generating. So, basically here we are generating the HLO for that particular ok. So, if dot experimental get compiler ir for this x y we are generating. So, you have x y tensor or whatever and it will actually be in the generated HLO to see the optimizations what are the optimizations that XLA has performed you can call this function f dot.

So, f is your function which is being wrapped by the function runtime which it compile True. So, f.experimental_get _ compiler_ir(x,y)(stage='optimized _hlo') because there are several stages you want to see optimized hlo or what are the optimizes optimization happen. Or you can print the graph into a dot form ok and you can use graphviz or graphviz converter from let say dot file to jpeg and you want to see the graph generated right.

So, f dot experimental get compiler ir will generate the dot file for your target or optimized code computational graph. So, that is about it and we will use that and inside with all the pipelines that we have discussed right. So, starting from your data pipeline with tf ds or tf.data then you have different distribution strategies, then you have mixed precision you have XLA compilation.

So, you can see that there are many many things that you can apply to get much more enhanced performance for your target model.

(Refer Slide Time: 36:59)



Conclusion

- Compilation can greatly improve the performance
- Annotate with tf.function(jit_compile = True)
- Ideally, compile the largest cluster
- Performance Boost

Credit: George Karpenkov

So, we will see all together in one example, but in conclusion. So, we have two the option for tf.function wrapper with jit compiler True annotate with that for the particular function and you will get great performance with that. Basically, in general case you will get for a of course, or a heavy model we will show you that in I mean applying randomly with very small models it might get you bad results even ok sometimes.

Ideally you will compile the largest cluster for that and so that is where you will get the best performance out of. So, let us jump into the demonstration where we will explore more about that.