The Execution Process of a Tensor in a Deep Learning Framework

Written by Xiaoyu Zhang; Translated by Xiaozhen Liu, Chenyang Xu, Yakun Zhou

This article focuses on what is happening behind the execution of a Tensor in the deep learning framework **OneFlow***. It takes the operator `oneflow.relu` as an example to introduce the Interpreter and VM mechanisms that need to be relied on to execute this operator. Hope this article will enlighten you on the system design of deep learning frameworks.*

First of all, let’s look at the following code in PyTorch:

Run it with PyTorch we get:

In the above code, the input Tensor `x` is fed to the operator `relu`, and the result is printed. Everything looks simple and normal. But if someone asks you if you are clear about what is happening behind this and when the Cuda Kernel corresponding to `relu` is called by the GPU, you may not be so clear. We are used to directly using deep learning frameworks and do not think more about it, so we may not have a deep understanding of the principles behind it.

Yet in this article, I will try to unravel this problem. But I’m not going to use PyTorch as an example but use OneFlow. Why use OneFlow as an example? That’s because First of all, I work in OneFlow inc and I know its execution mechanism better than PyTorch’s, which means my operation will be smoother when it comes to call chain tracing. Second, many design ideas for the execution mechanism are unique in OneFlow, so I believe this article will enlighten readers on the system design of deep learning frameworks.

So, let’s take a look at the execution process of a Tensor in OneFlow deep learning framework. For the sake of simplicity, this article only considers the single-node single-GPUs mode, and does not involve OneFlow’s unique consistent mode (distributed-related).

The Bridge between Python and C++

If we run the following code with OneFlow:

The system first creates an input Tensor on the GPU, and then calls the c++ functional interface `relu` which is exported to Python. Here involves Python wrapper and C++ `relu` functor, which are related to pybind11 binding. The operator `flow.relu` in the above Python code above eventually calls the ReLU C++ Functor implementation. Let's take a look at the code:

In this code, `op_` is a pointer to `OpExpr`, which calls the OpBuilder function in the constructor to create a new OpExpr. From the actual calling code `OpInterpUtil::Dispatch<Tensor>(*op_, {x});`, we can see that the construction and the execution of the operator are separate (because the Dispatch function distributes OpExpr, input Tensor, and others at the same time, and does not directly distribute the result Tensor of execution, so the real execution of operator has not started yet). The `OpInterpUtil::Dispatch` here is to distribute OpExpr, input Tensor, and other parameters (the operator ReLU has no parameters other than the input parameter), which also means the real execution has not started yet.

OpExpr can be simply understood as a unified abstraction of the OneFlow operators. opExpr can be broadly divided into BuiltinOpExpr, FunctionOpExpr, and other kinds of OpExpr, where BuiltinOpExpr can be subdivided into UserOpExpr and non-UserOpExpr, and users can build UserOpExpr by OpBuilder.

We do not have to fully understand the definition of OpExpr, we only need to know that a new OpExpr is built by OpBuilder here, and the new OpExpr has key information such as Op name, the ProtoBuf object of `UserOpConf proto_`, and the name of the input and output Tensor. Then, following this Dispatch function, we can find that the `Apply` method of the `GetInterpreter` function is called in `oneflow/core/framework/op_interpreter/op_interpreter_util.cpp`.

Here the OpExprInterpContext object will store the dynamic properties of the operator, device information, distribution information, etc. We do not focus on this object here because it is empty for `Relu` Functor. Next, we can analyze the Interpreter object.

Interpreter

From the above calling process, we can see that the operator in the Python layer actually calls the functor interface which is exported to Python. The Functor interface will submit the OpExpr, input Tensor and dynamic attribute `attr` to the Interpreter for processing, because the `GetInterpreter` function above means getting an Interpreter object. The Interpreter class is specifically to interpret the operator execution process, and the Dispatch function in the Relu Functor above is to distribute tasks to the Interpreter for execution. OneFlow's Interpreter is further divided into types such as Eager Mirrored Interpreter, Eager Consistent Interpreter, and LazyInterpreter. The example in this article does not involve distributed-related information, so the input Tensor is all Mirrored Tensor and calls Eager Mirrored Interpreter. Mirrored Tensor is independent on each rank, which is similar to PyTorch's Tensor.

The next `Apply` in the above code actually calls the `NaiveInterpret` function in the `oneflow/core/framework/op_interpreter/eager_mirrored_op_interpreter.cpp` file. The `NaiveInterpret` function first takes the OpExpr object, input and output Tensor, and an `OpExprInterpContext` object to derive the operator's device, output dtype, and output shape, etc.

Then, based on the derived meta-information (meta-information corresponds to the TensorMeta class object, which put the basic information of the Tensor such as shape, dtype, stride, etc. into a class for easy management), it builds `BlobObject`, `input_eager_blob_objects` and `output_eager_blob_objects` corresponding to the input and output respectively (which can be interpreted as the data pointers of the input and output Tensor).

Last, the execution kernel, the input and output Tensor data pointers, and the `OpExprInterpContext` object are sent to OneFlow's virtual machine (which can be interpreted as OneFlow's Eager runtime) in the form of instructions to execute and obtain the results.

Let’s look at how `NaiveInterpret` function achieves the final results in segments. The first piece of code is shown as follows:

This code iterates through the list of input Tensor, compares the device of each input Tensor with the default device passed in by the function. Once the device of the input Tensor is found to be different from the default device, an exception will be thrown. For examples like input Tensor on CPU but `nn.Module` on GPU, error checks can be performed and error messages can be printed to inform that the devices do not match. If the devices match, the `eager_blob_objects` of the input Tensor will be added to the `input_eager_blob_objects` list. The `eager_blob_object` of the input Tensor is a pointer of `EagerBlobObject` type, which is a data pointer to the input Tensor and later it will help to interact with OneFlow's virtual machine (VM).

> *Here is an additional explanation of the relationship among Tensor, TensorImpl, TensorMeta and BlobObject in OneFlow. Tensor and TensorImpl use the bridge-pattern design. Tensor is responsible for interfacing with Python and autograd upwards; TensorImpl is responsible for real data downwards. TensorMeta extracts the basic information of Tensor such as shape, dtype, and stride into a type and puts them together for easy management. BlobObject is the actual data object and has data pointers. The virtual machine uses this type to complete the computing task as instructed.*

The second piece of the code is as follows:

First, the code declares a pointer `output_eager_blob_objects` of type `EagerBlobObjectList`, and `output_tensor_metas`, which stores the meta-information of output Tensor. And then it iterates through the output Tensor list to determine whether the "ith" Tensor already has a value.

If not, a pointer of MirroredTensor type will be requested and initialized to `tensor_impl`. And the value of `output_tensor_metas` at index `i` will be updated to the Tensor meta-information of `tensor_impl` to prepare for the next shape and type derivation (if there is a value, it is the inplace call. In addition, we can find that the `BlobObject` with a value is the same object as the `BlobObject` of some input.

If the output Tensor already has a value (inplace mode), judge whether it has a data pointer of type `EagerBlobObject`. If so, take this data pointer and put it into `output_eager_blob_objects` of type `EagerBlobObjectList` just requested in the list. Subsequent shape derivation and dtype derivation will also use this `output_eager_blob_objects`.

The third piece of the code:

This code is Op’s derivation of device, shape and dtype. `user_op_expr.has_device_infer_fn()` is used to determine whether the current OpExpr has a device information derivation function. If not, the output Tensor's device information will be updated to the current `default_device`.

If it has, take the device information directly from `user_op_expr`. Whether or not it has been derived here has already been determined when registering the user operator. We see whether or not the register has a device derivation function in `UserOpExpr::Init` of the `oneflow/core/framework/op_expr.cpp` file. In addition, we can see which operators implement the device derivation function in the `oneflow/ir/include/OneFlow/OneFlowUserOps.td` file.

Next, `InferPhysicalShapeAndDType` in OpExpr is called to complete the shape and dtype derivation of the output Tensor. Follow-up on the `InferPhysicalShapeAndDType` function shows that it calls the shape derivation and dtype derivation functions which are defined when the user operator is being registered.

Then it iterates through `output_eager_blob_objects` and updates or checks the objects based on the already derived TensorMeta (the TensorMeta check is the result of the possible existence of the Inplace mentioned above, and the TensorMeta before and after the inplace cannot be changed).

The last piece of code:

The last piece of code is the most critical step when Interpreter interacts with the VM. `user_op_expr.MutKernel4Device` builds a StatefulOpKernel on `op_device` and sets the `is_shape_synced_` value of each `EagerBlobObject` object in `output_eager_blob_objects` to False. `is_shape_synced_` set to False means that the shape of the output Tensor is determined at runtime. The shape of the output Tensor can be obtained after the Kernel is executed.

Why should it be set to False by default? Because for an Op, whether its shape needs to be derived is the Op’s attribute, so it will be False by default. There is a flag in StatefulOpKernel, from which we can know which operators are at dynamic shape. If not, its flag is set to True, which means it is synchronized (no need to synchronize). `builder->LocalCallOpKernel` function is to build the instruction for the virtual machine. And PhysicalRun is responsible for sending this instruction to the VM and executing it to get the final result.

VM Introduction

The runtime of OneFlow Eager is abstracted as a virtual machine (VM). When we run the code `flow.relu(x)`, a `LocalCallOpKernel` instruction is sent to the VM by the interpreter above. VM requests memory for the output Tensor when it executes this instruction, calls ReLU's Cuda Kernel to perform the computation and writes the result to the output Tensor.

Let’s first know about some concepts of the VM and then look at the key code for further understanding.

During the running of a OneFlow program, the virtual machine is constantly polling in the background, executing new instructions if available, and continuing to poll if they are not. A virtual machine has two types of threads: scheduler thread and worker thread (if we run a Python script, the script runs in the main thread). Virtual machine’s polling is in the scheduler thread while the worker thread handles blocking operations, which is slow and not suitable for the scheduler thread.

Instruction is the smallest unit of virtual machine execution. The types of instructions in OneFlow are `AccessBlobByCallback`, `LocalCallOpKernel`, `ReleaseTensor`, etc. `AccessBlobByCallback` is to read and modify the value of a Blob, while `LocalCallOpKernel` is to run an Op, and `ReleaseTensor` is to release the memory of the Tensor whose declaration cycle has ended. Each instruction carries a `parallel_desc` indicating the instruction is executed on which devices (e.g. only on device 1, or on all devices), and binds a StreamType indicating the instruction is executed on which stream (at the beginning of the article, ReLU's corresponding `LocalCallOpKernel` is executed on a CudaStream).

Taking `LocalCallOpKernel` as an example, there are the following types of instructions depending on the StreamType:

According to the `cpu.LocalCallOpKernel` instruction, its stram_type is bound to `CpuStreamType`, and the definition in `oneflow/core/eager/cpu_opkernel_instruction_type.cpp` is as follows:

https://gist.github.com/YakunZhou/7370c732248302e5b20a245370f0450a

Each streamType can set whether this stream works on the scheduler thread, set the jobs like initializing, querying instruction status, and completing instruction computation.

> *Streams are the device abstraction in the virtual machine, and each stream corresponds to a device. In addition, instructions have infer and compute processes. Infer is the derivation of meta-information, while compute is to start the computational kernel for execution.*

Next, we'll look at the dependencies between instructions. The virtual machine instructions are executed disorderly, but there is a requirement for the order of execution of instructions that have dependencies. For example, if the user issues two instructions 'a' and 'b', 'a' instruction needs to modify the value of Blob 'c', but 'b' instruction needs to read the value of Blob 'c', then 'a' instruction has to be executed before the 'b' instruction.

So how are the dependencies between instructions constructed? The dependencies between instructions are implemented by relying on the operands carried by the instructions. The main types of operands are const, mut, and mut2. The const corresponds to input (read), and mut and mut2 correspond to output (write).

The instruction 'a' above has a mut operand 'c', and the instruction 'b' has a const operand 'c'. This way, 'a' dependency can be established between 'a' and 'b' by checking the type of 'c' in the instruction 'a' and 'b':

The infer of ‘b’ must be completed after the infer of ‘a’, and the compute of ‘b’ must be after the compute of ‘a’. The mut2 operand deals with some ops (such as unique) whose output shape can only be determined at the compute stage. For example, if a holds ‘c’ in the form of mut2 operand ‘a’, then both infer and compute of ‘b’ need to happen after computing ‘a’.

From the `LocalCallOpKernelPhyInstrOperand` instruction defined in `oneflow/core/eager/local_call_opkernel_phy_instr_operand.h`, it overloads `ForEachConstMirroredObject`, `ForEachMutMirroredObject`, and `ForEachMut2MirroredObject` three methods, corresponding to const, mut, mut2 operations respectively. In each overloaded method, call the incoming callback function (`const std::function<void(vm::MirroredObject* compute)>& DoEach`) to build dependencies between instructions. Let's take const as an example:

This line of code, `for (int64_t index : opkernel().input_tuple_indexes4const_ibns())` is used to traverse the const operand in the StatefulOpKernel object. It gets its subscript in the Input Tuple to get `index`. Then retrieve the `EagerBlobObject` object corresponding to this subscript according to `index`.

Then call the callback `DoEach` on `compute_local_dep_object` on `EagerBlobObject`, which is equivalent to consuming this `compute_local_dep_object` in a const way. The mut and mut2 are similar.

Here is also an explanation of how exactly the inter-instructional dependencies of the virtual machine are established. In the `HandlePending` member function in `oneflow/core/vm/virtual_machine_engine.cpp`, the `ConsumeMirroredObjects` function `for (const auto& operand :`

operands) calls the ForEachMutMirroredObject function for each operand. For example, for mut:

The DoEachT is ConsumeMutMirroredObject, consuming Mut Mirrored Object. Continue to follow the implementation of ConsumeMutMirroredObject:

The AccessMirroredObject adds to the list of directives that will access the mirrored_object.

The RwMutexedObject locks the reads and writes of mirrored_object. After we have the instruction dependency, we can construct the instruction edge. After the instruction edge is constructed, the virtual machine can execute a Dag composed of instruction nodes. One effective way to handle Dag is topological sorting, but in the virtual machine of OneFlow, this is done through ready_instruction_list and pending_instaruction_list. When the scheduler polls, it only needs to process these two lists continuously. Here is another look at the instruction side of the construction process, in this part of ConsumeMirroredObjects:

It will analyze the relationship between two instructions, such as one read and one write, or two read and write, to construct instruction edges separately and connect the two instructions.

Therefore, the instruction dependency of the virtual machine is not embedded in the virtual machine. Instead, it is realized by consuming the instruction’s operand. In addition to consuming the operand to construct the instruction dependency, it can also consume the device. Taking the mut operand of the LocalCallOpKernelPhyInstrOperand instruction as an example, here will get the device corresponding to StatefulOpKernel, such as cuda. Then each device method also has a `local_dep_object` member; each instruction consumes the `local_dep_object` member in the form of mut `local_dep_object`. In this way, the two instructions before and after are executed on the same device. So the execution order of these two instructions must be a dependency that needs to be executed in the order of launch because they both consume the same `local_dep_object` in mut.

> *The local_dep_object here is an object specially used to help the virtual machine build the instruction side. EagerBlobObject and Device hold this object. Then consuming it in sequential order establishes the connection between instructions.*

The Overall Call Chain of VM and Interpreter

This section goes through the call chain of Interpter and virtual machine macroscopically. First, the Python layer calls OneFlow's Op will be sent through the Interpreter to build the VM instructions and execute them. Taking ReLU as an example, the last step in the Interpreter is:

Then follow up the implementation of LocalCallOpKernel:

`auto instruction = intrusive::make_shared<vm::InstructionMsg>...` This code constructs a new instruction and binds a `parallel_desc` to it, indicating which devices to execute on (for example, only on number 0 execute on the card, or execute on all cards) and a StreamType, indicates on which stream the instruction is executed. And the `auto phy_instr_operand = JUST(vm::LocalCallOpKernelPhyInstrOperand::New...` above this code is used to bind instructions and operands. Now the instructions are available. The next step is to interact with the VM based on these newly created instructions to build the instruction side and execute it. The interface for this interaction is `PhysicalInterpreter::Run` (jump in from `PhysicalRun`).

Jump to the definition of `RunPhysicalInstruction`, in `oneflow/core/eager/eager_oneflow.cpp`:

Its input parameters are the `mut_instruction_list` and `eager_symbol_list` (objects in the virtual machine) of the global `InstructionsBuilder` object defined in the place where we construct the instruction. Jump to `RunPhysicalInstruction(instruction_list, eager_symbol_list)` to see the following definition:

The `virtual_machine->Receive(instr_msg_list)` here will get the instructions just constructed.

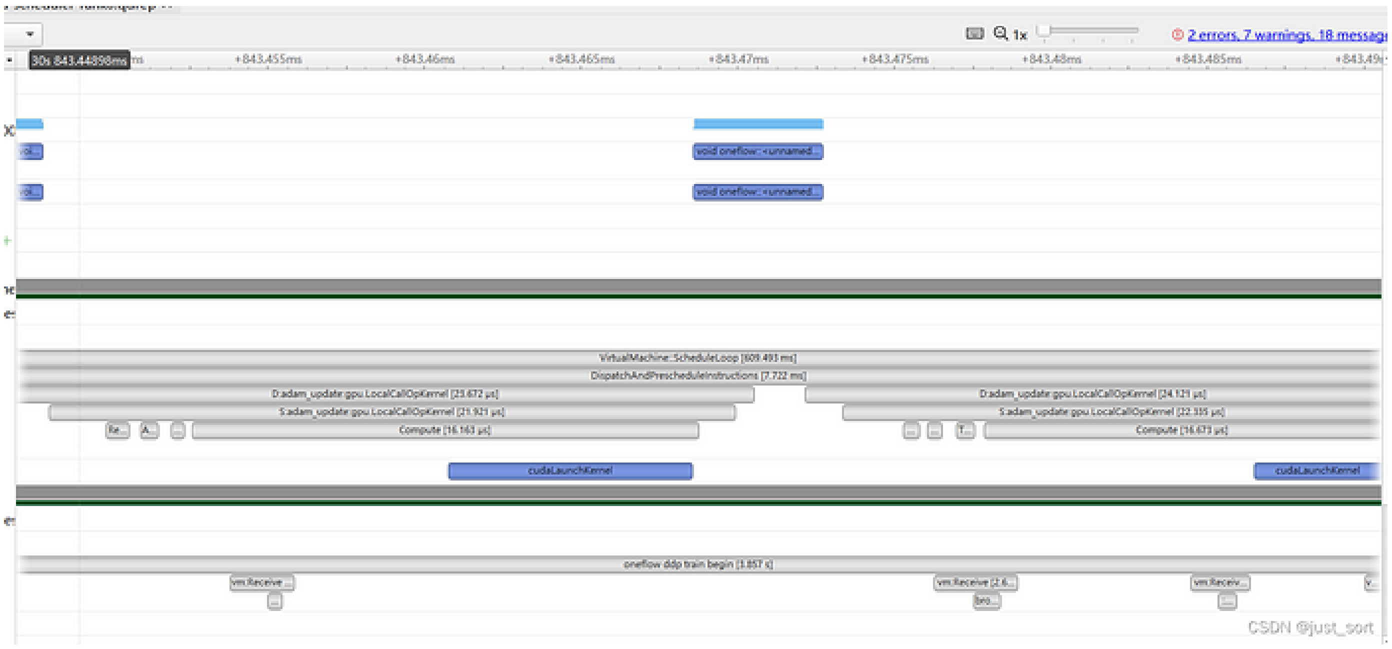
Once the instructions are obtained, they can be processed when the VM's Scheduler thread is polled, i.e. the `VirtualMachineEngine::Schedule` function here in `oneflow/core/vm/virtual_machine_engine.cpp`.

The Schedule function is constantly polling. The overall function can be divided into accepting instructions from the main thread, polling for the completion of instructions, handling blocking instructions, and dispatching ready instructions. In fact, when we click into HandlePending, we can find that it is consuming our `local_dep_opbject` for instruction construction and instruction edge linking, which corresponds to the process analyzed above.

NSYS Result Display

The details about Interpreter and VM are much more complicated than we thought. Finally, I will put a NSYS diagram generated during the training of a certain network.

You can see that the virtual machine is working, the scheduler thread is distributing ready instructions and launching Adam's cuda kernel to execute parameter updates.



30s 843.44898ms
+843.455ms
+843.46ms
+843.465ms
+843.47ms
+843.475ms
+843.48ms
+843.485ms
2 errors, 7 warnings, 18 messag
void oneflow::<unnamed...
void oneflow::<unnamed...
VirtualMachine::ScheduleLoop [609.493 ms]
DispatchAndPrescheduleInstructions [7.722 ms]
D:adam_update:gpu.LocalCallOpKernel [23.672 μs]
S:adam_update:gpu.LocalCallOpKernel [21.921 μs]
Compute [16.163 μs]
D:adam_update:gpu.LocalCallOpKernel [24.121 μs]
S:adam_update:gpu.LocalCallOpKernel [22.335 μs]
Compute [16.673 μs]
cudaLaunchKernel
cudaLaunchKernel
oneflow ddp train begin [3.857 s]
vm::Receive ...
vm::Receive [2.6...
vm::Receiv...
CSDN @just_sort

Summary

This article takes the operator `oneflow.relu` as an example to introduce the Interpreter and VM mechanisms that need to be relied on to execute this operator. Hope it will also be helpful to those who want to understand the execution mechanism of OneFlow Eager.

References