

How TensorFlow schedules operators ?

Overall Summarization (I)

- A `global` thread pool is created at the time the `Session` object is created.
- The user-defined computation will be ***partitioned to multiple sub-graphs across the device.***
- One executor will be created for one sub-graph.
 - *For example, in training tasks using 2 GPU cards, the graph will be partitioned into three parts: one for CPU, two for GPU card1 and card2.*
- The thread pool is shared among all executors.
- Fundamentally, the execution flow is bounded by the data flow dependencies among nodes in the computation graph.

Overall Summarization (II)

Each executor then will run operators in its sub-graphs "one by one"

- initialize a `ready` queue with root nodes on each device.
- If there are multiple root nodes in one device, ***each root node will be directly run*** by a thread if there are available threads in the thread pool.
- ***NOTE: Now the execution is in the computation thread:***
 - Once a node finishes computing, it will annotate its successor nodes as ready, which means pushing the successors into the ready queue.
 - Then all the OP kernels in the `ready` queue (kernels to be executed) will be examined one by one.

Overall Summarization (III)

Asynchronous kernels: most GPU kernels

- will be directly dispatched to threads in the thread pool.
- *If all the kernels in the ready queue are asynchronous, the current thread will execute the last one instead of dispatching it.*

Synchronous kernels: all CPU kernels

- will be divided into two parts: expensive kernels and inexpensive kernels.
- the current thread will execute **one** expensive kernel **or all** the inexpensive kernels.

Some More Notes

- In local training (`DirectSession` is called), TensorFlow uses Eigen's ThreadPool.
- Threads in the thread pool are scheduled by [Eigen implementation](#). TensorFlow does not schedule the threads itself.
- The worker function implemented in TensorFlow [only make a binary decision](#): *run the OP kernel in current thread or dispatch it to other threads.*
 - Then all the other things are left to Eigen scheduling.
- The Eigen ThreadPool that TensorFlow used is using this queue implementation: [RunQueue](#). There is one queue per thread.
 - *Currently I cannot explain more details about Eigen's scheduling algorithm.*

Now, Let's look into implementation details.

Execution Flow (I)

- The execution flow starts from Python end
 - define a `session` , call its `Cor` .
 - `session` 's `Cor` will call TensorFlow's C API :
`TF_NewSession`
- C API `TF_NewSession` uses the [factory method pattern](#).
 - The local training invokes `NewSession` implemented in `DirectSession`.
 - `NewSession` invokes `DirectSession` 's `Cor`
 - a. create a global thread pool
 - b. add device.

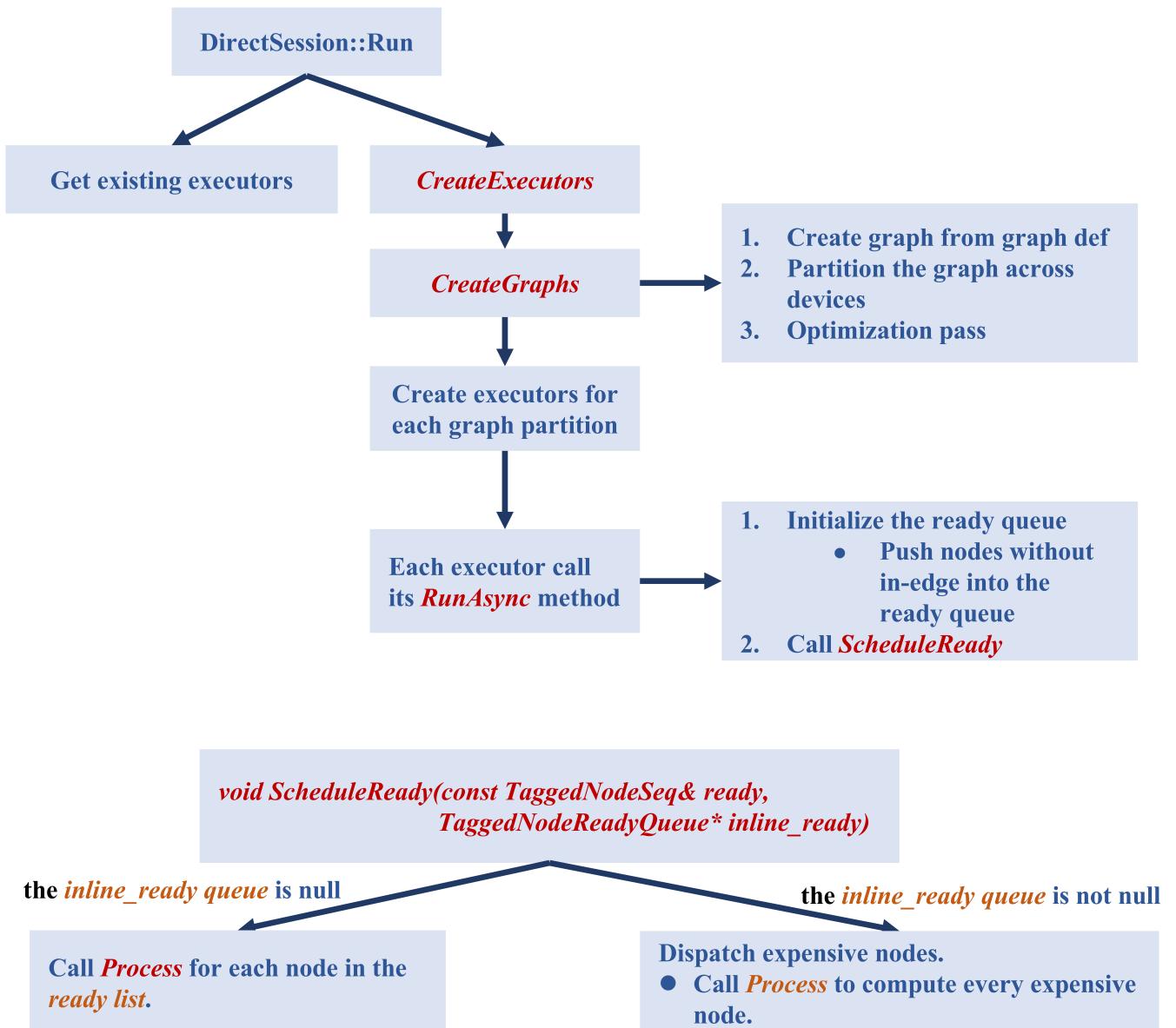
Execution Flow (II)

- When `session.run` in Python end is called, `DirectSession::Run` is called.
- Call `DirectSession::RunInternal` to execute a mini-batch computation.

Execution Flow (III) : RunInternal

- DirectSession::GetOrCreateExecutors
 - DirectSession::CreateGraphs
 - Create **several graphs** given the existing GraphDef and the input feeds and fetches
 - Partition the graph across devices
 - ConvertGraphDefToGraph
 - graph optimization pass : *post partition*
- DirectSession::CreateExecutors
- *Then the last and the most important step in RunInternal is to call ExecutorState::RunAsync for each executor.*

Execution Flow (IV)



Operators are scheduled in RunAsync

```
void ExecutorState::RunAsync(Executor::DoneCallback done) {
    // The Callback done here is an execution barrier.

    const Graph* graph = impl_->graph_.get();
    TaggedNodeSeq ready;

    ...

    // Initialize the ready queue.
    for (const Node* n : impl_->root_nodes_) {
        DCHECK_EQ(n->in_edges().size(), 0);
        ready.push_back(TaggedNode{n, root_frame_, 0, false});
    }
    if (ready.empty()) {
        done(Status::OK());
    } else {
        num_outstanding_ops_ = ready.size();
        root_frame_->iterations[0]->outstanding_ops = ready.size();
        done_cb_ = std::move(done);
        // Schedule to run all the ready ops in thread pool.
        ScheduleReady(ready, nullptr);
    }
}
```

ScheduleReady

There is a very important function `ScheduleReady`, let's first summarize its behavior

- `ScheduleReady` scheduled operators based on *two FIFO queue*
 - *ready queue*: operators will be dispatched to other threads in the thread pool.
 - *inline_ready queue*: operators will be run in the current thread.
- If the `inline_ready` queue is empty, all the operator kernels in the `ready` queue will be scheduled in the thread pool.
- If the `inline_ready` queue is not empty, then go through the `ready` queue:
 - *dispatch expensive nodes* to threads in the thread pool
 - *push all inexpensive nodes to `inline_ready`*
 - If all the nodes in `ready` queue are expensive and `inline_ready` is empty, *push the last expensive node to `inline_ready` queue instead of dispatching it.*

ExecutorState::Process

NOTE: *Process is running in the thread from the thread pool.*

- create a `inline_ready` queue and push the `ready` node into the queue.
 - when calling `Process`, **only one** one ready node is passed to it.
- while the `inline_ready` queue is not empty, repeat the following steps:

For **asynchronous** kernels

- call `ComputeAsync` with a callback function `done` which will be called after OpKernel is finished.
- In the `done` callback
 - `ExecutorState::PropagateOutputs` is called, which propagates outputs along out edges, and puts newly ready nodes into the ready queue.
 - Call `NodeDone` and `NodeDone` *calls* `ScheduleReady again with inline_ready being fixed to null.`

```
NodeDone(s, state->item->node, ready, stats, nullptr);
```

in the above call, the last parameter is the `inline_ready` queue, which is always `null`, for asynchronous kernels.

For *synchronous* kernels

- call `Compute`
- call `ExecutorState::PropagateOutputs`
- Call `NodeDone`

```
NodeDone(s, item.node, ready, stats, &inline_ready);
```

Let's put too detailed implementations aside, summarize the overall execution flow.

