# *How TensorFlow schedules operators ?*

# Overall Scheduling Process (I)

- A global thread pool is created at the time the `Session` object is created.
- User's computation graph will be ***partitioned to multiple sub-graphs across the device***.
- One executor will be created for one sub-graph.
    - *For example, in a training task using 2 GPU cards, the computation graph will be partitioned into three sub-graphs: one for CPU, two for GPU1 and GPU2 respectively.*
- The thread pool is shared among all executors.

# Overall Scheduling Process (II)

Each executor run operators in its sub-graph one by one, and the execution flow is bounded by dataflow dependencies among nodes.

For each executor:

1. a `ready` queue is created and is intialized with the root node
2. if there is an idle thread in the thread pool, let's call it `thread A`, ***the root node will be directly scheduled to `thread A` to run***

# Overall Scheduling Process (III)

***NOTE: Now the execution is in `thread A`***:

- Once the root node(operator) finishes execution, it will annotate its successor nodes as ready by pushing them into the `ready` queue.
- Operators in the `ready` queue will be scheduled to run one by one.

# Overall Scheduling Process (IV)

***NOTE Again: Now the execution is in `thread A`***:

For asynchronous kernels:

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

For synchronous kernels:

- synchronous kernels are registered as expensive kernels and inexpensive kernels.
- `thread A` will execute *one* expensive kernel ***or*** *all* the inexpensive kernels.

# Some More Implementation Detials

- 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 ThreadPpool 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.

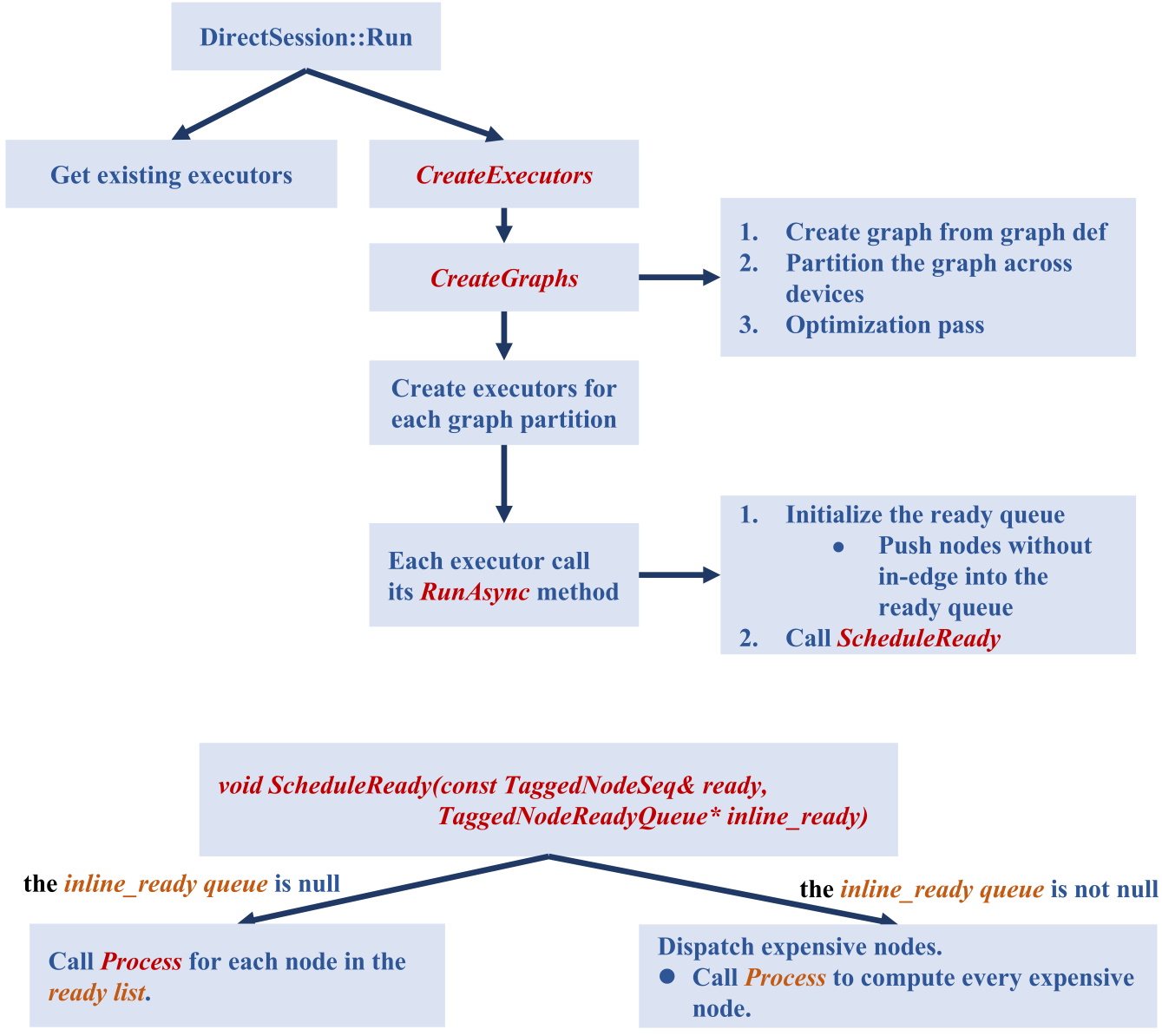
# Execution Flow (II)

- When `seesion.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 dispathed to other theads 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:
  - ***dispath 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 dispathing it***.

```
void ExecutorState::ScheduleReady(const TaggedNodeSeq& ready,
                                  TaggedNodeReadyQueue* inline_ready) {
  if (ready.empty()) return;

  ...

  if (inline_ready == nullptr) {
    // Schedule to run all the ready ops in thread pool.
    for (auto& tagged_node : ready) {
      runner_([=]() { Process(tagged_node, scheduled_usec); });
    }
    return;
  }

  const GraphView& gview = impl_->gview_;
  const TaggedNode* curr_expensive_node = nullptr;
  for (auto& tagged_node : ready) {
    const NodeItem& item = *gview.node(tagged_node.node->id());
    if (tagged_node.is_dead || !item.kernel_is_expensive) {
      inline_ready->push_back(tagged_node);
    } else {
      if (curr_expensive_node) {
        runner_(std::bind(&ExecutorState::Process, this, *curr_expensive_node,
                          scheduled_usec));
      }
      curr_expensive_node = &tagged_node;
    }
  }
  if (curr_expensive_node) {
    if (inline_ready->empty()) {
      inline_ready->push_back(*curr_expensive_node);
    } else {
      runner_(std::bind(&ExecutorState::Process, this, *curr_expensive_node,
                        scheduled_usec));
    }
  }
}
```

# 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 funciton 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);
```

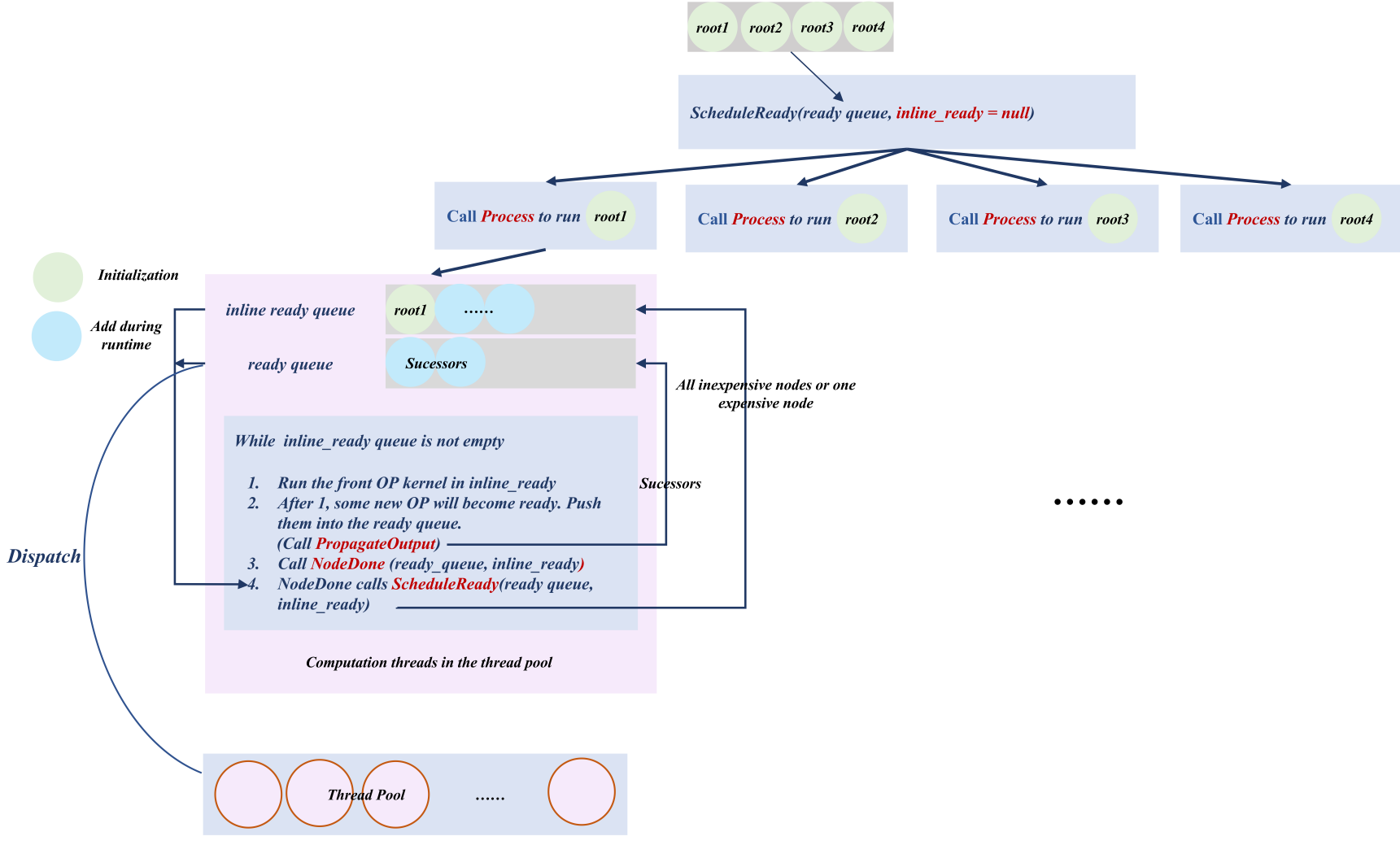
*in the above call, the last parameter is the inline_ready queue, which is always null, for asynchronous kernels.*

## 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.*


root1
root2
root3
root4
ScheduleReady(ready queue, inline_ready = null)
Call Process to run root1
Call Process to run root2
Call Process to run root3
Call Process to run root4
Initialization
Add during runtime
inline ready queue
root1
.....
ready queue
Sucessors
All inexpensive nodes or one expensive node
While inline_ready queue is not empty
1. Run the front OP kernel in inline_ready
2. After 1, some new OP will become ready. Push them into the ready queue. (Call PropagateOutput)
3. Call NodeDone (ready_queue, inline_ready)
4. NodeDone calls ScheduleReady(ready queue, inline_ready)
Sucessors
Dispatch
Computation threads in the thread pool
Thread Pool
.....