Tensorflow 图计算引擎概述

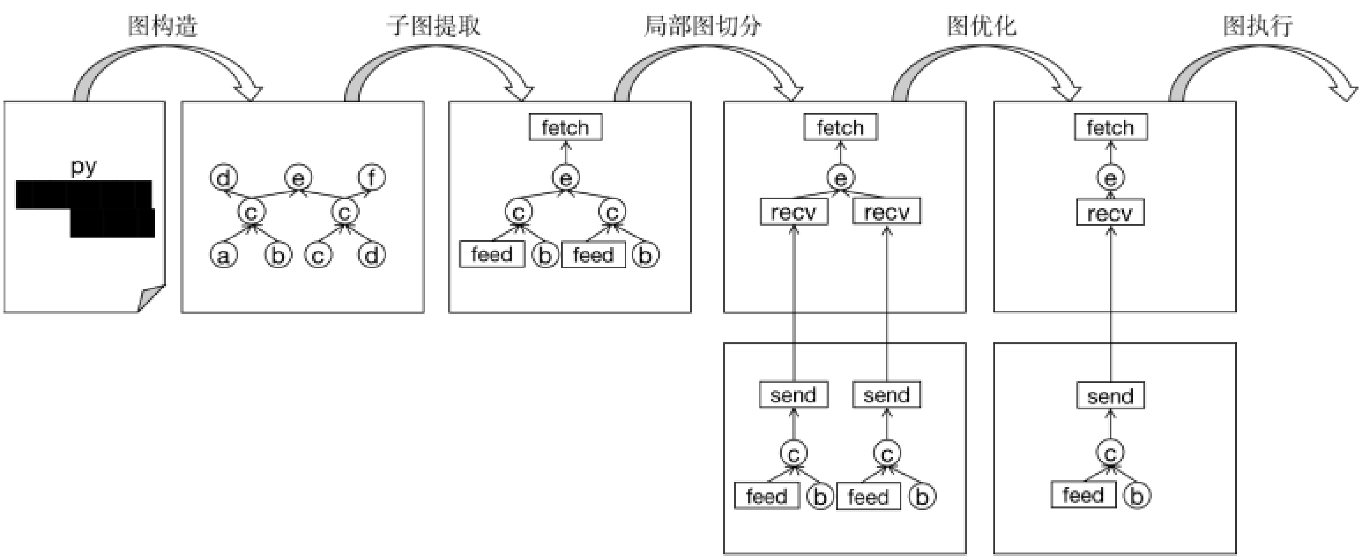
977 Jiang XIAO

2019年7月28日 at pm2:35 (last edited 2021年1月31日 at pm2:43)

tensorflow的用户可以使用多种语言来构造的自己的图, 但各种语言的API最终都会经由C API 进入tensorflow 运行时. 可以说, 对于运行时代码, 其上边界就是C API. 比如, 通过python描述的一张网络, 就是通过类似下面的几个python-C接口进入运行时的.

```
#9  0x00007f9de118daa4 in PyEval_EvalFrameEx ()
#10 0x00007f9de118f0bd in PyEval_EvalCodeEx ()
```

tensorflow整体上可以看做一个"图语言的编译器", 和所有编译器的优化以及翻译的功能类似, Graph在运行时中的处理过程, 可以分为 **图构造->图优化->图执行** 几个阶段. 其中, 图优化随同图构造一同被执行.



全图构造及其优化

Session初次构造时, 应用层代码中定义的数据流图转换成GraphDef格式, 经由C API传入DirectSession.Extend(), 参考调用栈如下

```
        PyEval_EvalCodeEx()
          PyEval_EvalFrameEx()
```

```
_wrap_ExtendSession()
  tensorflow::ExtendSession()
    tensorflow::ExtendSessionGraphHelper()
      tensorflow::SessionRef::Extend()
        tensorflow::DirectSession::ExtendLocked()
          tensorflow::DirectSession::MaybeInitializeExecutionState(out_already_in
            if out_already_initialized:
              return
              flib_def_.reset(new FunctionLibraryDefinition())
            tensorflow::GraphExecutionState::MakeForBaseGraph()
                std::unique_ptr<GraphExecutionState> ret(new GraphExecutionState(g
                if (!ret->session_options_->config.graph_options().place_pruned_gr
                  tensorflow::GraphExecutionState::InitBaseGraph()
                    OptimizationPassRegistry::Global()->RunGrouping(PRE_PLACEMENT)
                    Placer placer()
                    placer.Run()
                    OptimizationPassRegistry::Global()->RunGrouping(POST_PLACEMENT
            out_already_initialized = false
          if already_initialized:
            flib_def_->AddLibrary(graph.library())
            std::unique_ptr<GraphExecutionState> state
            execution_state_->Extend(graph, &state))
            execution_state_.swap(state)
_wrap_TF_SessionRun_wrapper()
  tensorflow::TF_SessionRun_wrapper()
    tensorflow::TF_SessionRun_wrapper_helper()
      TF_SessionRun()
        TF_Run_Helper()
          tensorflow::SessionRef::Run()
            tensorflow::DirectSession::Run()
              DirectSession::GetOrCreateExecutors(executors_and_keys)
                CreateExecutors()
                  std::unique_ptr<ExecutorsAndKeys> ek(new ExecutorsAndKeys);
                  std::unordered_map<string, std::unique_ptr<Graph>> graphs;
                  CreateGraphs(&graphs)
                    if (options_.config.graph_options().place_pruned_graph()):
                      MakeForPrunedGraph()
                        ret->InitBaseGraph()
                          if (session_options_ && session_options_->config.graph_o
                            PruneGraph()
                              if (options.use_function_convention):
                                feed_rewrites.emplace_back(new subgraph::ArgFeedRe
                                fetch_rewrites.emplace_back(new subgraph::RetvalFe
                                ValidateFeedAndFetchDevices()
                              else:
                                feed_rewrites.emplace_back(new subgraph::RecvFeedR
                                fetch_rewrites.emplace_back(new subgraph::SendFetc
                              subgraph::RewriteGraphForExecution(graph, feed_rewri
                          OptimizationPassRegistry::Global()->RunGrouping(PRE_PLAC
                          Placer placer()
                          placer.run()
                          OptimizationPassRegistry::Global()->RunGrouping(POST_PLA
                        graph_ = new_graph.release();
                        ret->BuildGraph()
                          OptimizeGraph()
                            grappler::RunMetaOptimizer()
                              MetaOptimizer::Optimize()
                          if (session_options_ == nullptr || !session_options_->co
```

```
                                            PruneGraph()
                                          std::unique_ptr<ClientGraph> dense_copy(new ClientGraph)
                                    else:
                                      execution_state->BuildGraph();
```

-0- 初次构造图, tensorflow/core/common_runtime/graph_execution_state.cc
-3-26- 构造图的两个入口, -3-如果不允许对全图进行Prune, 就在Session构造的时候进行OptimizationPassRegistry优化, 并在Session::Run的时候执行Grappler全图优化, 而如果允许Prune, 那么全图的Prune, OptimizationPassRegistry优化和Grappler优化都会放在Session::Run, 随着Executor的首次构造一同构造Graph并优化
-11- 初次构造图, tensorflow/core/common_runtime/graph_execution_state.cc
-15- InitBaseGraph, 根据配置决定是否进行Prune, 核心是完成Placement. 这里不会进行Prune,
-16- 执行 PRE_PLACEMENT 优化器
-18- 遍历Graph, 根据算法为每个Node分配DEVICE. 原则是在满足必要的约束的前提下, 尽可能满足上层的放置要求. XLA JIT使用DEVICE_GPU_XLA_JIT和DEVICE_CPU_XLA_JIT，
-19- 执行 POST_PLACEMENT 优化器
-24- 扩展构造图, Extend()代码内有注释解释整个扩展流程
-29- C API入口
-33- 将每个Graph分割为若干子图(Graph, Partition), 同时, 针对每个Partition, 构造相应的 Executor.
-40- InitBaseGraph, 根据配置决定是否进行Prune, 核心是完成Placement. 这里会进行Prune
-53- XLA JIT是通过注册XLA_CPU_DEVICE和XLA_GPU_DEVICE来接入图计算引擎， 所以，XLA JIT实现OptimizationPassRegistry的9个Pass也就不难理解, 参考https://www.tensorflow.org/guide/graph_optimization
-56-62- BuildGraph, 根据Graph构造ClientGraph, 生成可独立执行的子图。BuildGraph执行前, 所在的GraphExecutionState对象一定是已经执行过InitBaseGraph了. 这里会根据配置决定是否进行Prune, 核心是Grappler优化, 如果通过MakeForBaseGraph进到这里(-62-), 就会进行Prune, 如果通过MakeForPrunedGraph进到这里, 就不会进行Prune, 所以, Prune是两种方式构造的全图都会进行, 区别在于, MakeForBaseGraph是InitBaseGraph不进行Prune –> InitBaseGraph进行Placement –> BuildGraph进行Grappler –> BuildGraph进行Prune, 而MakeForPrunedGraph是InitBaseGraph进行Prune –> InitBaseGraph进行Placement –> BuildGraph进行Grappler –> BuildGraph不进行Prune, 看, 只是Prune时机不同而已.
-57- Grappler，Grappler是全图优化。

图切分及其优化

如果允许对图进行剪枝(pruned)和切分(partitioned), 就会在首次Session.run()中随着Executor的构造一同进行。

```
PyEval_EvalFrameEx()
  _wrap_TF_SessionRun_wrapper()
    tensorflow::TF_SessionRun_wrapper()
      tensorflow::TF_SessionRun_wrapper_helper()
        TF_SessionRun()
          TF_Run_Helper()
            tensorflow::SessionRef::Run()
              tensorflow::DirectSession::Run()
                DirectSession::GetOrCreateExecutors(executors_and_keys)
                  CreateExecutors()
                    std::unique_ptr<ExecutorsAndKeys> ek(new ExecutorsAndKeys);
                    std::unordered_map<string, std::unique_ptr<Graph>> graphs;
                    CreateGraphs(&graphs)
                      if (options_.config.graph_options().place_pruned_graph()):
                        MakeForPrunedGraph()
```

```
16.           else:
17.             execution_state->BuildGraph();
18.               OptimizationPassRegistry::Global()->RunGrouping(POST_REWRIT
19.           Partition(popts, &client_graph->graph, &partitions)
20.           OptimizationPassRegistry::Global()->RunGrouping(POST_PARTITIONI
21.         ek->items.reserve(graphs.size());
22.         ProcessFunctionLibraryRuntime()
23.         GraphOptimizer optimizer()
24.         for iter in graphs:
25.           optimizer.Optimize(&partition_graph)
26.             for rounds < kMaxRounds; ++rounds:
27.               RemoveListArrayConverter()
28.               RemoveDeadNodes()
29.               RemoveIdentityNodes()
30.               ConstantFold()
31.               OptimizeCSE()
32.               FixupSourceAndSinkEdges()
33.               ExpandInlineFunctions()
34.             std::unique_ptr<Graph> copy(new Graph(g->flib_def()));
35.             CopyGraph(*g, copy.get());
36.             graph->swap(copy);
37.           LocalExecutorParams params;
38.           params. = ...
39.           item->graph = partition_graph.get();
40.           NewExecutor(&item->executor)
41.             ExecutorFactory::GetFactory(executor_type, &factory)
42.               auto iter = executor_factories()->find(executor_type);
43.               *out_factory = iter->second;
44.             factory->NewExecutor(params,..., out_executor) //如果"DEFAULT"
45.               NewLocalExecutor()
46.                 ExecutorImpl* impl = new ExecutorImpl(params, std::move(g
47.                 *executor = impl;
48.       executors_.emplace(&ek)
49.     FunctionCallFrame call_frame()
50.     DirectSession::RunInternal()
51.       const size_t num_executors = executors_and_keys->items.size();
52.       ExecutorBarrier* barrier = new ExecutorBarrier()
53.       Executor::Args args;
54.        thread::ThreadPool* pool = ...
55.       Executor::Args::Runner default_runner = [this, pool](Executor::Args
56.       for item in executors_and_keys->items:
57.         args.runner = default_runner
58.         args.runner = [this, device_thread_pool](Executor::Args::Closure
59.         item.executor->RunAsync(args, barrier->Get());
60.       WaitForNotification()
```

-5- C API 入口

-10- 将每个Graph分割为若干子图(Graph, Partition), 针对每个Partition, 构造相应的 Executor.

-18- 执行 POST_REWRITE_FOR_EXEC 优化器

-19- 子图分割

-20- 执行 POST_PARTITIONING 优化器

-24- 逐个优化每一个子图, 预期中的ConstatFold和CSE等优化措施都在这里执行

-40- 为该 Partition 构造 Executor, 所以, 一共有多少个子图, 就有多少个 Executor, 考虑到图在执行过程中可能会发生修改, 所以所有的Executor会缓存起来

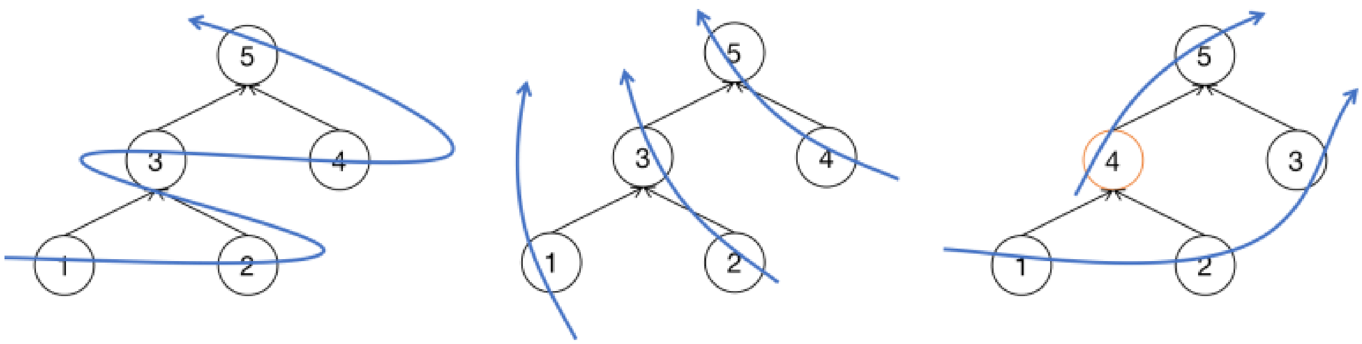
-48- 保存所有的executor_and_keys

-56- 遍历Executor们

-57- 分配线程池, 要么用公共的, 要么每个Executor有自己的线程池, 后续执行节点运算的时候会用到.
-59- 执行该Executor下的图

图执行

tfop的图执行引擎的核心思想是: **不断寻找每一个入度为0 的节点, 执行之, 直到整张图被遍历完成**. Tensorflow并没有简单的使用粗暴的使用线程池来提高性能，而是用了下图中的C方式，以一种比较谨慎的方式进行并行操作：只有当一个节点是expensive的时候，才会开新线程计算



A B C

```
ExecutorImpl::RunAsync(args, barrier->Get()); //executor.cc
  (new ExecutorState(args, this))->RunAsync(std::move(done));
    const Graph* graph = impl_->graph_.get();
    TaggedNodeSeq ready;
    for  Node* n in impl_->root_nodes_:
      DCHECK_EQ(n->in_edges().size(), 0)
      ready.push_back(TaggedNode{n, root_frame_, 0, false});
    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);
      if tagged_node.is_dead || !item.kernel->IsExpensive():
        inline_ready->push_back(tagged_node);
      else
        runner_(std::bind(&ExecutorState::Process, ...)) //tensorflow:::ExecutorStat
          EntryVector outputs
          while (!inline_ready.empty()):
            const NodeItem& item = *gview.node(id);
            s = PrepareInputs(iterm)
            stats = nullptr
            outputs.clear()
            if (stats_collector_ && !tagged_node.is_dead):
              stats = stats_collector_->CreateNodeExecStats(node) --> 应该是NodeExecSt
            if (item.kernel_is_async):
              tensorflow::Device::ComputeAsync()
                tensorflow::HorovodBroadcastOp::ComputeAsync()
            else:
              OpKernelContext ctx(&params, item.num_outputs);
              nodestats::SetOpStart(stats);
                stats->RecordComputeStarted();
              tensorflow::Device::Compute()
                tensorflow::XlaCompileOp::Compute()
              // After item->kernel computation is done, processes its outputs.
```

```
35.         ProcessOutputs(ctx, stats)
36.         for i in item.num_outputs:
37.          const TensorValue val = ctx->release_output(i);
38.            TensorValue value = outputs_[index];
39.            outputs_[index] = TensorValue();
40.            return value
41.          Entry* out = &((*outputs)[i]);
42.          out->val.Init(std::move(*val.tensor));
43.         nodestats::SetMemory(stats, &ctx);
44.          stats->SetMemory(ctx);
45.           auto* ms = stats_->mutable_memory_stats();
46.           ms->set_persistent_memory_size(ctx->persistent_memory_allocated());
47.        if !launched_asynchronously:
48.         MaybeMarkCompleted()
49.         // After processing the outputs, propagates the outputs to their dsts.
50.         // Contents of *outputs are left in an indeterminate state after
51.         // returning from this method.
52.         PropagateOutputs(tagged_node, &item, &outputs, &ready);
53.         completed = NodeDone();
54.          ScheduleReady(ready, inline_ready);
55.       if completed:
56.        ScheduleFinish()
```

-1- 图执行入口, executor.cc
-7- 准备ready节点, 即入度为0的节点
-12- op调度器, 整个执行引擎就是通过不断执行的ScheduleReady()来驱动的
-16- 图执行核心逻辑, tensorflow::::ExecutorState::Process() 处理ready的节点
-24- 创建节点执行状态
-25- 执行节点计算逻辑, 先执行AsyncOpKernel, 再执行OpKernel, tfop的Compute()方法就在这里被调用的
-35- 一个节点的计算输出会保存在输入的OpKernelContext中, 此处将其取出
-44- TODO: NodeExecStatsWrapper 的?
-47- 对于同步加载的节点
-52- 将该节点输出传播给其后继节点
-18- 遍历每一个传入的ready节点
-13- 对于每一个待处理的tagged_node, 如果是 dead || 非expensive || 当前线程的inline_node为空, 使用当前线程执行该节点计算
-28- 否则将其放入线程池, 由其他线程执行

Related:
[Tensorflow OpKernel机制详解]
[Tensorflow Op机制详解]
[Tensorflow Optimization机制详解]
[Tensorflow 图计算引擎概述]

技术 Tensorflow, 技术