Tensorflow XLA Service 详解 II

1190 Jiang XIAO
2019年9月26日 at pm1:55 (last edited 2020年6月20日 at am9:34)

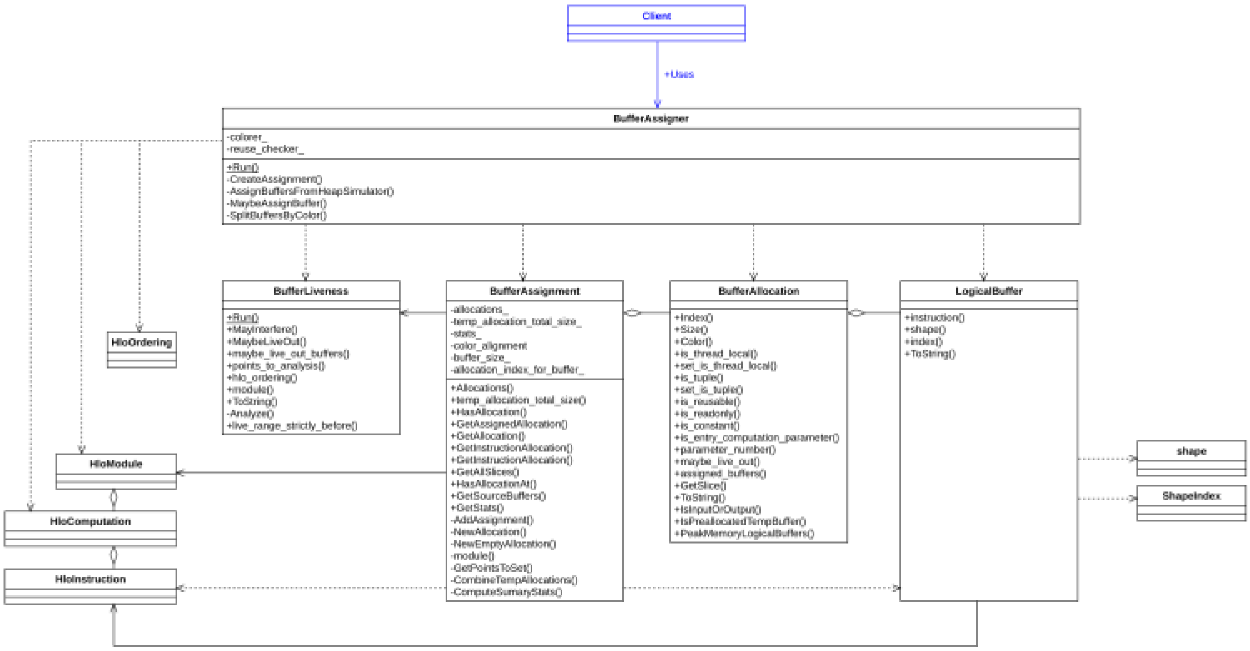
本文主要介绍在XLA service阶段针对HloInstruction做的一些显存优化, 对于训练框架来说, 显存优化的工作至关重要, 主要是由于现阶段GPU+CUDA远没有CPU+Linux组合强大, 后者有完善的建立在虚拟内存基础上的内存管理机制, 内存的高效使用由linux kernel来负责, 即便物理内存不足, 还可以使用swap, 内存压缩等技术确保内存的高效供应, 而在GPU+CUDA里, 这方面的工作很大程度让渡给了程序员自己来搞定, GPU程序接触到的就是物理显存, 如果程序的显存申请超过显存容量, 整个程序就会直接coredump, 此外, 显存本身就集成在GPU板卡上, 无法像内存一样扩展, 而GPU本身造价昂贵, 最后, 在深度学习训练中, 大力出奇迹的现状下, 显存的消耗明显超过的摩尔定律, 这也加剧了显存供求关系的矛盾, 正式由于训练框架做了大量的优化, 才能让模型跑起来.

XLA Service的显存优化设计思想与tensorflow整体一样遵循"静态图"的设计: 先整体优化, 再落地实施. 其中, xla/service/buffer_assignment.cc 是整个显存优化的核心, 在1.14版本中, xla/service/支持两种后端: cpu和gpu, 纷纷针对两种backend有进一步的优化算法, 本文主要针对GPU的优化逻辑进行分析

核心文件

内存优化公共:
xla/service/buffer_assignment 内存优化核心文件
xla/service/buffer_liveness.cc 内存片生命周期分析
GPU相关:
xla/service/gpu/buffer_allocations.cc BufferAllocation的组合
xla/service/gpu/gpu_hlo_scheudle.cc   Hlo的处理顺序, 与显存的优化策略息息相关, 简单地说, 按照BFS并行执行的HloInstruction消耗的显存肯定大于所有的HloInstruction都顺序执行.

核心抽象



HloSchedule

XxxSchedule是TF的代码风格, 类似的有更底层用于Thunk调度的ThunkSchedule, 以及Service提供的HloSchedule. HloSchedule中最重要的就是封装了HloOrdering.

HloSchedule

XLAService内存优化的本质是处理LoigicalBuffer和BufferAllocation之间的关系, 原则是使用尽可能少的BufferAllocation去承载尽可能多的LogicalBuffer, 而如何使用的更少, 就涉及到了对Hlo图的分析, 就涉及到了Ordering的问题, 使用不同策略生成Ordering, 直接影响两个LogicalBuffer之间的约束关系, 最简单的, 在图遍历中, 使用DFS和BFS的2种方式遍历会导致图上节点的内存依赖关系大有不同.

HloOrdering是描述HloInstruction加载序列的基类, 派生类有PredecessorHloOrdering, DependencyHloOrdering 和 SequentialHloOrdering, 其中, DependencyHloOrdering基于依赖关系, 所以可以并行, 性能更高, 但耗更多的内存, 而SequentialHloOrdering完全串行, 性能相对低, 但可以节约更多内存, 而 PredecessorHloOrdering 是个虚类, 需要子类进一步填充predecessors_, 这也是GPU后端使用的方式.不同的Ordering会影响内存的依赖关系, 进一步影响Launch到GPU后Kernel的执行序列.

```
// An HLO ordering based on data dependencies in the HLO graph. In this partial
// order, instruction A executes before instruction B only if there is a path
// from A to B in the HLO graph. For example, given the following graph:
/*
          param
         /     \
      negate   exp
          \     /
           add
*/
// DependencyHloOrdering gives the following executes-before relations:
//   param executes before negate, exp, and add
//   negate executes before add
//   exp executes before add
//   add executes before nothing
```

```
// negate and exp are not ordered because the dependencies allow either to
// execute before the other (or in parallel). DependencyHloOrdering ordering
// allows maximum parallelism and enables any execution order which satisfies
// data dependencies. This requires pessimistic assumptions about buffer live
// ranges and can result in more memory used than more constrained orderings.
class DependencyHloOrdering : public PredecessorHloOrdering {

// An HLO ordering based on a total order of instructions in each computation.
// The computation total order is a sequencing of all of its instructions in
// the computation (eg, {inst0, inst1, inst2,...}) as in single-threaded
// execution. For example, given the following HLO graph:
/*
          param
         /     \
      negate   exp
          \    /
           add
*/
// and the following sequence:
//
//  {param, negate, exp, add}
//
// SequentialHloOrdering gives the following executes-before relations:
//   param executes before negate, exp, and add
//   negate executes before exp and add
//   exp executes before add
//   add executes before nothing
// This is more constrained than DependencyHloOrdering in this example because
// negate and exp are ordered (negate before exp). This enables param to share
// the same buffer as exp (param buffer is dead after exp). Generally, this
// ordering enables more buffer sharing (reduced memory usage) because buffer
// interference is reduced relative to DependencyHloOrdering.
class SequentialHloOrdering : public HloOrdering {
```

GpuHloSchedule

Build()	构造GpuHloSchedule单例, 同时会根据Stream的数量采用BFS或Sequence的LaunchOrder,, 这个LaunchOrder会用于构造GpuHloOrdering, 以及thunk_launch_order_
ThunkLaunchOrder()	拿到上述thunk_launch_order_
ConsumeHloOrdering()	拿到上述GpuHloOrdering

GpuHloOrdering

```
class GpuHloOrdering : public PredecessorHloOrdering : public HloOrdering
```

在使用Multi-stream的情况下, 一方面stream no的分配算法会遍历图给每个HloInstruction分配不同的stream no, 另一方面, GpuHloOrdering也会据此选择BFS算法来生成Ordering存入thunk_launch_order以及HloOrdering.predecessors_ , 后续的显存优化会据此优化显存, 确保Multi-Stream在实际执行的过程中不会产生显存问题.

BufferAssigner

BufferAssigner 用于构造BufferAssignment对象, 可以理解为BufferAssignment的wrapper

BufferAssignment

内存管理入口类. 所有关于内存优化的内容均在其Run()中实现, 在nvptx_compiler.cc:RunBackend()调用

BufferLiveness

描述一块内存的生命周期, “computes liveness of the output buffers of HLOs and their interference”, 为显存优化的决策提供支持. BufferLiveness 的构造过程经历3层分析, 从内而外, 即从前到后:
LogicalBufferAnalysis: 构造LogicalBuffer, 每个Instruction都可以有多个LogicalBuffer, 通过Accept机制遍历所有的Instruction
TuplePointsToAnalysis: 构造PointToSet, 需要之前的LogccalBuffer, 最终也是通过Accept机制遍历所有的Instruction二者其实的都是Hlo的DFS遍历handler

```
class TuplePointsToAnalysis : public DfsHloVisitorWithDefault
class LogicalBufferAnalysis : public DfsHloVisitorWithDefault
```

```
1. NVPTXCompiler::RunBackend()
2.   BufferAssigner::Run()
3.     assigner::CreateAssignment()
4.       liveness = BufferLiveness::Run(module, std::move(hlo_ordering)  //class BufferL
5.         liveness = new BufferLiveness()
6.         liveness->Analyze()
7.           points_to_analysis_ = TuplePointsToAnalysis::Run()
8.             logical_buffer_analysis = LogicalBufferAnalysis::Run()
9.               analysis = new LogicalBufferAnalysis()
10.              analysis.Analyze()
11.              return analysis
12.            analysis = new TuplePointsToAnalysis(logical_buffer_analysis)
13.            analysis.Analyze()
14.            return analysis
15.          maybe_live_out_buffers_ = points_to_analysis_->GetPointsToSet(root).CreateF
16.        return liveness
17.      assignment(new BufferAssignment(module, std::move(liveness)
```

-2- 规划内存的总入口, 注意入参
-15- liveout的部分只发生在root instruction(TODO)

LogicalBuffer

“虚拟内存”, LogicalBuffer 对内存需求的抽象, 内存优化的实质就是调整LogicalBuffer和BufferAllocation的关系, 将来会被HloValue替换掉. 同时, 整个内存优化过程都使用color的概念, 相同的color表示可以融合, 这点在BufferAllocation和BufferSet中都有用到, 显然, 处于一个BufferAllocation之下的LogicalBuffer都有相同的color, 要不也不会在一起, 相应的BufferAllocation实例也使用想用的color进行标记
什么样的两个LogicalBuffer会被分配给同一个BufferAllocation?
1. kWhile, kCall, kConditional 的LogicalBuffer
2. `cannot_merge_buffer_sets()`: !(两个buffer的id不同(不是一个buffer) && 会发生干涉)

```
1.  void LogicalBufferAnalysis::NewLogicalBuffer(HloInstruction* instruction,  const Shap
2.    CHECK_EQ(logical_buffers_.size(), next_buffer_id_);
3.    logical_buffers_.emplace_back( absl::make_unique<LogicalBuffer>(instruction, index,
4.      //LogicalBuffer
5.      : BufferValue(instruction, index, id),
6.        id_(id)
7.        const Shape& shape = ShapeUtil::GetSubshape(instruction->shape(), index);
8.      index_(index) {}
9.    output_buffers_[std::make_pair(instruction, index)] = logical_buffers_.back().get()
10.   ++next_buffer_id_;
11. }
```

-1- LogicalBuffer只存Shape(确切的是SubShape) id 和inst指针
-3- 所有的LogicalBuffers对象都会丢到, LogicalBufferAnalysis::logical_buffers中
-7- 所以, 一个LogicalBuffer只对应一个Subshape的内存, 不是整个Instruction

BufferAllocation

BufferAllocation 物理内存的抽象, 一个 BufferAllocation 实例最终对应一块连续物理内存, 通常, 一块物理内存会被多个不干涉的LogicalBuffer公用.一个BufferAllocation实例可以分为:

```
Status BufferAssignment::ComputeSummaryStats() {
  for (auto& allocation : Allocations()) {
    if (allocation.is_entry_computation_parameter()) {
      stats_.parameter_allocation_count++;
      stats_.parameter_allocation_bytes += allocation.size();
    }
    if (allocation.is_constant()) {
      stats_.constant_allocation_count++;
      stats_.constant_allocation_bytes += allocation.size();
    }
    if (allocation.maybe_live_out()) {
      stats_.maybe_live_out_allocation_count++;
      stats_.maybe_live_out_allocation_bytes += allocation.size();
    }
    if (allocation.IsPreallocatedTempBuffer()) {
      stats_.preallocated_temp_allocation_count++;
      stats_.preallocated_temp_allocation_bytes += allocation.size();
    }
    stats_.total_allocation_count++;
    stats_.total_allocation_bytes += allocation.size();
```

BufferAllocation::Slice

BufferAllocation里的一片, 关联在该BufferAllocation的每一个LogicalBuffer都会有对应的Slice

GetSlice 根据 buffer, 重新构造一个 Slice 对象

```
BufferAllocation::Slice BufferAllocation::GetSlice(
    const LogicalBuffer& buffer) const {
  const OffsetSize os = FindOrDie(assigned_buffers_, &buffer);
  return Slice(this, os.offset, os.size);
}
```

ShapeIndex

一个 ShapeIndex 对应的是一个 LogicalBufferList, 真正使用的时候是用的 ShapeIndex 来索引到buffer

```
1.  StatusOr<BufferAllocation::Slice> BufferAssignment::GetUniqueSlice
2.    for (const LogicalBuffer* buffer : GetPointsToSet(instruction).element(index))
3.      //elements()
4.      return tree_.element(index).buffers
```

-4- index是ShapeIndex, 描述一个Shape的索引, 不是Intuction. buffers 是 BufferList，此处可以看到, 直接用index来索引得到的BufferList,

```
1.     IrEmitterUnnested::BuildInitializerThunk()
2.         return {absl::make_unique<MemzeroThunk>(a, nullptr)};
```

-2- 使用 ShapeIndex 索引 直接->Allocation::Slice->Thunk,

BufferAllocations

GPU后端对一组BufferAllocation的抽象

```
1.   std::vector<se::DeviceMemoryBase> buffers_
```

-1- 物理内存对象, se::DeviceMemoryBase是StreamExecutor对一块线性内存资源的抽象，提供了最基础offset+size的计算方法

```
1.  StatusOr<std::unique_ptr<BufferAllocations>> BufferAllocations::Builder::Build()
2.    for BufferAllocation::Index i in num_buffers:
3.      BufferAllocation allocation = buffer_assignment->GetAllocation():
4.      if address  already registered:
5.        buffer_allocations->SetBuffer(i, *address);
6.      if (allocation.maybe_live_out() || allocation.IsPreallocatedTempBuffer()):
7.        se::DeviceMemoryBase buffer_address;
8.        buffer = memory_allocator->Allocate(device_ordinal, buffer_size)
9.        buffer_address = buffer.Release();
10.       buffer_allocations->SetBuffer(i, buffer_address);
```

-2- 每个 buffer 都有一个 BufferAllocation
-8- 真正分配新的buffer

优化过程

```
NVPTXCompiler::RunBackend()
  hlo_schedule = GpuHloSchedule::Build(*module, *stream_assignment, pointer_size_))
    std::unique_ptr<GpuHloSchedule> schedule(new GpuHloSchedule)
  //this analysis figures out which temp buffers are required to run the computation
  BufferAssigner::Run(hlo_schedule->ConsumeHloOrdering(), BufferSizeBytesFunction(), Bu
    //BufferSizeBytesFunction()
    shape_size = ShapeSizeBytesFunction()
      ShapeUtil::ByteSizeOf(shape, pointer_size);
        ByteSizeOfTupleIndexTable(shape, pointer_size);
          return pointer_size * shape.tuple_shapes_size();
    //Run()
```

```
BufferAssigner assigner
assigner.CreateAssignment(HloModule, hlo_ordering, buffer_size)
  liveness = BufferLiveness::Run(module, std::move(hlo_ordering)
    liveness = new BufferLiveness()
      liveness->Analyze()
        points_to_analysis_ = TuplePointsToAnalysis::Run()
          logical_buffer_analysis = LogicalBufferAnalysis::Run(module)
            std::unique_ptr<LogicalBufferAnalysis> analysis(new LogicalBufferAnalys
            analysis->Analyze();
              NewLogicalBuffer()
          analysis(new TuplePointsToAnalysis(module, logical_buffer_analysis.Consum
          analysis->Analyze()
            for computation in module_->MakeNoFusionComputations():
              computation->Accept(this)
              for instruction in computation->instructions():
                if instruction is not kFusion:
                  continue
                GatherFusionInstructions(instruction, fusion_instructions)
            for instruction in fusion_instructions:
              instruction->fused_expression_root()->Accept(this)
          return analysis
        for computation in module->computation():
          if computation->IsFusionComputation():
            continue
          for instruction in computation->instructions():
            for alias_buffer in points_to_analysis_->GetPointsToSet().CreateFlatten
              if aliased_buffer->instruction() != instruction:
                alias_buffers_.insert(aliased_buffer)
          if computation == module_->entry_computation():
            maybe_live_out_buffers_ = points_to_analysis_->getPointsToSet().CreateF
    return liveness
  assignment(new BufferAssignment(module, std::move(liveness)

  flat_hash_set<const LogicalBuffer*> colocated_buffers;
  BuildColocatedBufferSets(&colocated_buffer_sets)
    points_to_analysis = buffer_liveness.points_to_analysis()
    module->input_output_alias_config().ForEachAlias([]{AddSetToColocatedBufferSets
    for computation in module->MakeComputationPostOrder():
      for instruction in computation->MakeInstructionPostOrder():
        if instruction->opcode()== HloOpCode::kWhile:
          AddBufferToColocatedSet()
          AddSetToColocatedBufferSets()
        else if == HloOpCode::kCall:
          //...
        else if == HloOpcode::kConditional:
          //...
    // Given a list of colocated buffer sets (each colocated buffer set represents
    // the logical buffers that would be assigned to the same physical buffer),
    // try to merge the sets if the buffers can be shared. Returns the merged set.
    new_colocated_buffer_sets = MergeColocatedBufferSets()
      // Given the interference map of a graph (the list of interfering node indice
      // for each node), perform graph coloring such that interfering nodes are
      // assigned to different colors. Returns the assigned color of the nodes, whe
      // the colors are represented as integer values [0, color_count).
      std::vector<int64> ColorInterferenceGraph()
      // Assign a color to each colocation set in colocated_buffer_sets, such that
      // the sets that can be merged are assigned with the same color.
      auto assigned_colors = ColorInterferenceGraph(interference_map);
        for node in nodes: //nodes: 0 ~ node_count-1
          for neighbor in interference_map[node]:
```

```
          color = assigned_colors[neighbor]
          if color != kColorUnassigned:
            available_colors[color] = false
          ...
    std::vector<ColocatedBufferSet> new_colocated_buffer_sets(num_sets)
    for i in colocated_buffer_sets.size():
      buffer_sets = colocated_buffer_sets[i]
      new_colocated_buffer_sets[assigned_colors[i]].insert(buffer_set.begin(), bu
    return new_colocated_buffer_sets;
  swap(colocated_buffer_sets, new_colocated_buffer_sets)
colorer_(assignment->liveness())
GatherComputationsByAllocationType(module, &thread_local_computations, &global_co
// For each buffer set in 'colocated_buffer_sets', assigns all buffers in the
// same set to the same buffer allocation in 'assignment'.
AssignColocatedBufferSets(colocated_buffer_sets, &colocated_buffers, &colocated_a
  for colocated_buffer_set in colocated_buffer_sets:
    BufferAllocation* allocation = nullptr
    for LogicalBuffer buffer in colocated_buffer_set:
      instruction-> buffer->instruction()
      computation = instruction->parent()
    for LogicalBuffer buffer in colocated_buffer_set:
      buffer_size = assignment->buffer_size_(*buffer);
      if allocation == nullptr:
        allocation = assignment->NewAllocation(*buffer, buffer_size)
        colocated_allocations->insert(allocation->index())
      else:
        assignment->AddAssignment(allocation, *buffer, 0, buffer_size)
          allocation->AddAssignment(buffer, offset, size);
            assigned_buffers_.emplace(&buffer, offset_size);
      colocated_buffers->insert(buffer)
GatherComputationsByAllocationType(module, &thread_local_computations, &global_co
// First assign buffers for global computatations. Temporary buffers for
// sequential computations are collected in 'buffers_to_assign_sequentially'.
flat_hash_map<const HloComputation*, flat_hash_set<const LogicalBuffer*>>buffers_
for computation in global_computations:
  // Assigns buffers to the instructions in the given computation. "assignment"
  // is modified to reflect the new buffer assignments. If is_thread_local is
  // true, then all assigned buffers have the is_thread_local flag set to
  // true.
  AssignBuffersForComputation(colocated_buffers, buffers_to_assign_sequentially,
    std::vector<const LogicalBuffer*> sorted_buffers;
    for (auto* instruction : computation->instructions()):
      for buffer in assignment->points_to_analysis().GetBuffersDefinedByInstructi
        sorted_buffers.push_back()
    for instruction : computation->MakeInstructionPostOrder():
      post_order_position.emplace(instruction, position);
    // Every sequential computation must get an entry in the
    // buffers_to_assign_sequentially map, even if we end up with an empty set
    // of buffers. This ensures we can correctly determine whether to run
    // whole-module heap simulation.
    buffers_to_assign_sequentially->emplace(computation, flat_hash_set<const Logi
    std::vector<BufferAllocation::Index> new_allocation_indices;
    // A sorted multimap from size to indices of colocated allocations.
    std::multimap<int64, BufferAllocation::Index> colocated_allocation_size_to_in
    for index in colocated_allocations:
      for buffer_offset_size in assignment->GetAllocations(index).assign_buffers
        if IsTuple():
          consider_reuse = false
          break;
      if considering_reusing:
```

```
        sorted_colocated_indices.push(index)
    while !sorted_colocated_indices.empty():
      auto index = sorted_colocated_indices.top()
      sorted_colocated_indices.pop()
      // Returns the size of the allocation. Necessarily this must be at least as
      // large as any LogicalBuffer assigned to this allocation.
      colocated_allocation_size_to_indices.emplace(assigne...size(), index)
    for buffer in sorted_buffers:
      if colocated_buffers.contains(buffer)
        continue
      instruction = buffer->instruction()
      // Function which returns the buffer size for a given logical buffer (shape
      buffer_size = assignment->buffer_size_(*buffer)
      if instruction->opcode() == HloOpcode::kConstant:
        BufferAllocation* allocation = assignment->NewAllocation(*buffer)
      ...
      if ... ==
      if ... ==
      if ... ==
      if buffer->...:
        for operand in instruction->operands():
          for operand_slice in assignment->GetAllSlices():
            BufferAllocation allocation = assign.GetMutableAllocation(operand_sl
            if(MaybeAssignBuffer(allocation, buffer, assignment)):
              //bool BufferAssigner::MaybeAssignBuffer()
                ...
              assigned_operand = true
          if assigned_operand
            break
      if reuse_colocated_checker_ ... && !assignment->HasAllocation(*buffer):
        it = colocated_allocation_size_to_indices.lower_bound(buffer_size)
        while(it != colocated_allocation_size_....)
          allocation = assignment->GetMutableAllocation(it->second)
          if (MaybeAssignBuffer(allocation)):
            colocated_allocation_size_to_indices.erase(it)
            break
          ++it;
      if(!assignment->HasAllocation(*buffer)):
        // Find the smallest buffer which can be reused iterating from end of
        // new_allocation_indices (smallest) to beginning (largest).
        for allocation_index = new_allocation_indices.size() - 1; allocation_inde
          BufferAllocation* allocation = assignment->GetMutableAllocation(new_al
          if MaybeAssignBuffer(allocation, *buffer, assignment):
            break
      if !assignment->HasAllocation(buffer) && has_sequential_order && !liveness
        // There is a sequential instruction ordering, so we delay assignment of
        // temp buffers until after the loop. We do this right before we decide t
        // create a new allocation, to ensure we've exhausted all the buffer
        // re-use cases above.
        //
        // Entry parameters and thread local buffers were already handled earlie
        // in this loop iteration.  See BufferAllocation::IsPreallocatedTempBuffe
        // for the definition of temp buffers.
        (*buffers_to_assign_sequentially)[computation].insert(buffer);
        continue
      if !assignment->HasAlocation()
        allocation = assigment->NewAllocation(buffer, buffer_size)
        new_allocation_indices.push_back(allocation->index())
// Assigns 'buffers_to_assign_sequentially' using heap simulation, assuming
// the HLO instructions will be executed in the sequential order given by
```

```
// assignment->liveness().hlo_ordering().SequentialOrder. If
// 'run_whole_module_heap_simulation' is true, the heap simulation will be run
// assuming all global computations are sequentially ordered.
// Assign buffers with sequential ordering, if any. If all global computations
// are sequential, we can run heap simuation on the whole module, which
// reduces memory usage.
AssignBuffersWithSequentialOrdering()
  HloSchedule schedule(&assignment->module());
  hlo_ordering = assignment->liveness().hlo_ordering();
  auto get_heap_algorithm = ...
  if (run_whole_module_heap_simulation):
    for (const auto& pair : buffers_to_assign_sequentially):
      HloInstructionSequence* instruction_sequence = hlo_ordering.SequentialOrde
      schedule.set_sequence()
      all_buffers_to_assign.insert()
      color_map = SplitBuffersByColor()
    for (auto& single_colored_set : color_map) :
      result = HeapSimulator::Run()
      AssignBuffersFromHeapSimulator()
  else:
    for (const auto& pair : buffers_to_assign_sequentially):
      instruction_sequence = hlo_ordering.SequentialOrder(*computation)
      color_map = SplitBuffersByColor(buffers_to_assign)
      for (auto& single_colored_set : color_map):
        buffer_value_set = ToBufferValueFlatSet(single_colored_set.second)
        options.buffers_to_assign = &buffer_value_set;
        result = HeapSimulator::Run(get_heap_algorithm(alignment),
                    assignment->points_to_analysis(),
                    assignment->buffer_size_)
          HeapSimulator heap(std::move(algorithm), size_fn, options, &schedule)
          const HloComputation* entry_computation = module.entry_computation();
          HloInstructionSequence& instruction_sequence = schedule.sequence(entry_
          heap.RunComputation(entry_computation, instruction_sequence, points_to_
            for instruction in instruction_sequence.instructions():
              points_to = points_to_analysis.GetPointsToSet(instruction);
              for user in instruction.users():
                ...
            for instruction in instruction_sequence.instructions():
              buffers_defined_by_instruction = points_analysis.GetBufferDefinedBy
              for buffer in buffers_defined_by_instruction:
                ...
              for oprand_buffer in used_buffers[instruction]:
                ...
              for buffer in buffers_defined_by_instruction:
                if !shared: -->
                  alloc_size_by_instruction += size_fn_(*buffer);
                  Alloc(buffer, instruction);
          heap->finish()
            Result result = algorithm_->Finish();
        AssignBuffersFromHeapSimulator(result)
          BufferAllocation* allocation = assignement->NewEmptyAllocation(result.h
          for (const auto& buffer_chunk : result.chunk_map):
            assignment->AddAssignment(allocation, buffer, chunk.offset, chunk.si
              allocation->AddAssignment(bufferm, offset, size)
          allocation->peak_buffers_ = ComputePeakMemoryLogicalBuffers(*allocatio
for computation in thread_local_computation:
  if computation IsFusionComputation():
    continue
  AssignBuffersForComputation(/*is_thread_local=*/true)
for buffer in assignment_.liveness().maybe_live_out_buffers()
```

```
253         if assignement->HasAllocation():
254           alloc = assignment->GetMutableAssignedAllocation()
255           alloc -> set_maybe_live_out(true)
256       assignment->CombineTempAllocations();
257         std::partition(allocations_.begin(), allocations_.end(),
258                        [](const BufferAllocation& allocation) {
259                          return !allocation.IsPreallocatedTempBuffer();
260                        });
261         allocations_.erase(first_temp_it, )
262         for combined:combined_allocation_map:
263           allocations_.push_back()
264         for index in allocations_.size():
265           allocation->set_index(index):
266             for buffer_offset_size in allocation->assigned_buffers_:
267               LogicalBuffer buffer = buffer_offset_size.first
268               allocation_index_for_buffer_[buffer] = index
269   ir_emitter_context();
270   IrEmitterUnnested ir_emitter
271   entry_computation->Accept(&ir_emitter)
272   ptx = CompileToPtx()
273   const std::vector<uint8> cubin = CompilePtxOrGetCachedResult()
274   ir_emitter.ConsumeThunkSequence(), std::move(stream_assignment), hlo_schedule->Thunk
275   auto* gpu_executable = new GpuExecutable()
276   gpu_executable->set_ir_module_string(ir_module_string_before_opt);
```

-2- 这个backend其实不是llvm的backend 而是HLO的backend?
-6- //buffer_assignment.cc，都是对元数据的管理，没有真的分配，包含了所有的内存管理代码，显存优化的入口，这里ConsumerHloOrdering()返回的是hlo_ordering
-8- 就是下文的buffer_size
-11- 对于一个tuple描述的shape, size的计算
-15- 遍历 module//buffer_liveness.cc –> DFS analysis, 构造一个liveness, 需要针对HloModule的TuplePointsToAnalysis::Run()->LogicalBufferAnalysis::Run()等一系列分析,
-19- 遍历 HloModule, 为每个HloInstruction构造相应的Logicalbuffer, 存入LogicalBufferAnalysis, 并返回其指针, `class LogicalBufferAnalysis : public DfsHloVisitorWithDefault`, 说明也是用来遍历的handler
-21- 遍历节点, 执行各种handler, 建立logicalbuffer和instruction及其subshape的关系, 并存储在logicalbuffer中
-22- 构造LogicalBuffer对象
-26- 所有非fusion的buffer构造完毕
-27- 这个循环收集所有的fusion_inst
-30- gather 该computation下的所有的fusion instruction
-32- 给fusion的inst分配logical buffer
-44- 构造BufferAssignment
-52- 当前只有这3种op可以colocated
-53- 将这个instruction对应的LogicalBuffer放到BufferSet
-54- 将BufferSet放到BufferSets
-60- colocate意味着最终分配的是同一块物理内存
-62- 使用邻接矩阵的方式表示一张图, 用int64标识一个节点. 遍历每一个node, 首先遍历其每一个neighbor, 这个neighbor不是相邻节点, 而是通过 cannot_merge_buffer_set 设置的, 比如id()不同&&… 他们用过的color就标记为不可用, 取一个neighbor都没用过的color, 最小未用, 给这个node, 最终的效果是, 每一个node和他的neighbors节点的颜色都不同, 但可能和其他节点相同, 相邻的不能共用buffer, 猜测应该是不能共用输入及输出, 前面的输出还是可以作为后面的输入的. 最终, 就是在这里解决内存冲突: 不相邻的标记为相同颜色, 说明后续可能可以合并. MergeColocatedBufferSets 遍历所有的BufferSet, 将能合并的set合并了，合并之前, 每个BufferSet都和Instruction/node关联, 合并之后, BufferSets中的一个Set里的buffer就不是和instruction强相关了, 可能是几个合在

一起的, 不过, 不存在把一个BufferSet拆成几个的情况, 两个bufferset, 要么能合并, 要么不能合并, 不会把一个BufferSet内的LogicalBuffer拆分.
-70- interference_map表示vector的vector，表征的是colocated_buffer_sets中两个set的关系，而从这里看, 每个set里存储的一组logicalbuffer其实都是一个node使用, 这一点其实也可以从colocated_buffer_sets的构造过程可以看出: 3种op每种op都会先添加一组上logical_buffer到colocated_buffer_set, 再把set放到sets中
-72- 将 interference_map中表征的有冲突的node用不同的color表示
-77- num_sets是最大的color值, 即能将nodes划分为互斥最小集合的数量
-80- 将不冲突的, 即color相同的node都放一起
-87- AssignColocatedBufferSets, 代码里的AssignXXX都是构造/分配BufferAllocation的意思, 比如这个 AssignColocatedBufferSets, 就是根据BufferSets分配BufferAllocation, 遍历colocated_buffer_sets 里的每一个Set里的每一个LogicalBuffer, 将一个Set的LogicalBuffer都放在一个BufferAllocation中, 一个Allocation里的LogicalBuffer不一定一样大, 将二者关联的时候会传入buffer_size的, 此时, offset都还是0
-92- Module->computation->instruction的嵌套关系
-94- 根据assigner构造时传入的函数对象计算, □ 这里是遍历所有的buffersize, 将相同的合并到一个allocation中
-95- 为 buffer 分配allocation
-96- **每个allocation内的buffer或buffer_size是固定的?**
-99- 将这个buffer 赋给已有BufferAllocation对象, 这里 offset是0, size就是buffer_size, 是第一次存储这两个值
-106- 第一个顺序执行的点
-107- 至此, 就完成了初始的内存分配代码, 接下来 AssignBufferForComputation(), 就是将 Computation都关联到Allocation, 首先是 global_computations, 遍历所有的Instruction->points_to_analysis->GetBuffersDefinedByInstruction(), 将所有的Buffer按照从大到小排序, 之所以从大到小排序, 应该和 BufferAllocation的Size 要比里面的所有 Buffer的size 都大的原因, 这样就能充分利用 allocation, 毕竟越少越好, 前面的用大块 buffer 搞了大的, 后面就可以乘凉, NewAllocation 本身就会将自己链入Assignment, 总之, 这里会通过一系列判断, 将大部分buffer存入尽可能少的Allocation中. global_computations 进行Assign的过程中, 有一些buffer被加入了sequential的范围，这部分buffer在AssignBuffersWithSequtialOrdering, 同样, 这里Assign也是找Allocation的意思, 这里会用到堆模拟器, 会计算很多offset, size之类的, 将buffer加入到Heap中, `NewEmptyAllocation(result.heap_size,` 就能看出, 此外, 由于根据hlo_ordering, stream_no> 1时不会所有的都是顺序的, 所以, 会存在根据hlo_ordering最终有多个chunk的情况, 更具体的, 一个computation对应多个LogicalBuffer, 并进行一个simulate, 得到一组chunk, 和一个allocation.
-112- 有关于顺序执行的代码
-116- 将传入的computation下的每一个instruction的每一个LogicalBuffer都入队
-118- 保存传入的computation下每一个instruction及其位置
-130- 根据注释, :// Output tuple table may be allocated at run-time, so make sure we don’t // overwrite them.
-140- 处理在sorted_buffer中, 但不在 colocated_buffers 中的, 前文的 colocated_buffers 只有3种HloOpCode可以用
-155- 获取每一个已有的Allocation
-161- 说明该buffer已经找到了 allocation
-162- 说明之前没有找到能用的
-163- 找到最接近的大于的key
-172- 从小到大(即从后向前) 找合适的allocation存储buffer
-186- 不是liveout的buffer, 即temp的buffer, 先存着, 没有相应的Allocation, 这里存的是comutation和buffer们的映射关系
-189- 实在不行了, 分配个新的Allocation
-193- 计算一个buffer的 live out 或 interfere 等
-199- buffer 顺序执行, 重点关注
-202- 这里用的LazyBestFitHeap, ChoostBestHeapAlgorithm
-208- 默认构造一个buffer只有0, 通过default_colorer(), 还有其他逻辑导致可能没有0吗?-213- out/trace.log:2019-08-02 10:44:04.437956: I tensorflow/compiler/xla/service/buffer_assignment.cc: 1262] Running per-

computation heap simulation
-215- 这里的是LogicalBuffer->BufferValue里的color, 不是前文用来合并BufferSet的color, 是通过buffer_liveness.h 中的 DefaultColorer赋值的. 这部分buffers都有color, 但没有被colocate合并过的, 也没有被合并单奥已有allocation中的temp buffer
-216- single_colored_set里的buffer是可以合并的.
-237- 分析时候可以共享内存
-243- 用整个优化过的堆大小来定义新的allocation, 一个allocation里是可以存放很多不同offset和size的buffer的. allocation有自己的size, 全文来看, 同样的color分配一个allocation. 那直接取最大的不就好了。 所以最终, Assignment里存储的tmp buffer是最小数量的Allocation和他的size, 用color区分 + 之前3种已经处理好的Op, + 其他
-245- 会存入allocation
-256- 把多个allocation里的buffer存到同一个combined_allocation里
-257- 划分的原则
-270- XLA HLO Visitor机制
-271- 使用上述代码构造的buffer来构造Thunk

Related:
Tensorflow XLA Service Buffer优化详解
Tensorflow XLA Service 详解 II
Tensorflow XLA Service 详解 I
Tensorflow XLA Client | HloModuleProto 详解
Tensorflow XlaOpKernel | tf2xla 机制详解
Tensorflow JIT 技术详解
Tensorflow JIT/XLA UML
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 , XLA , 技术

Tensorflow XLA Service 详解 I
Tensorflow-Horovod安装部署checklist

One comment on “Tensorflow XLA Service 详解 II”

Pingback: Tensorflow OptimizationPassRegistry机制详解 - sketch2sky

Leave a Reply