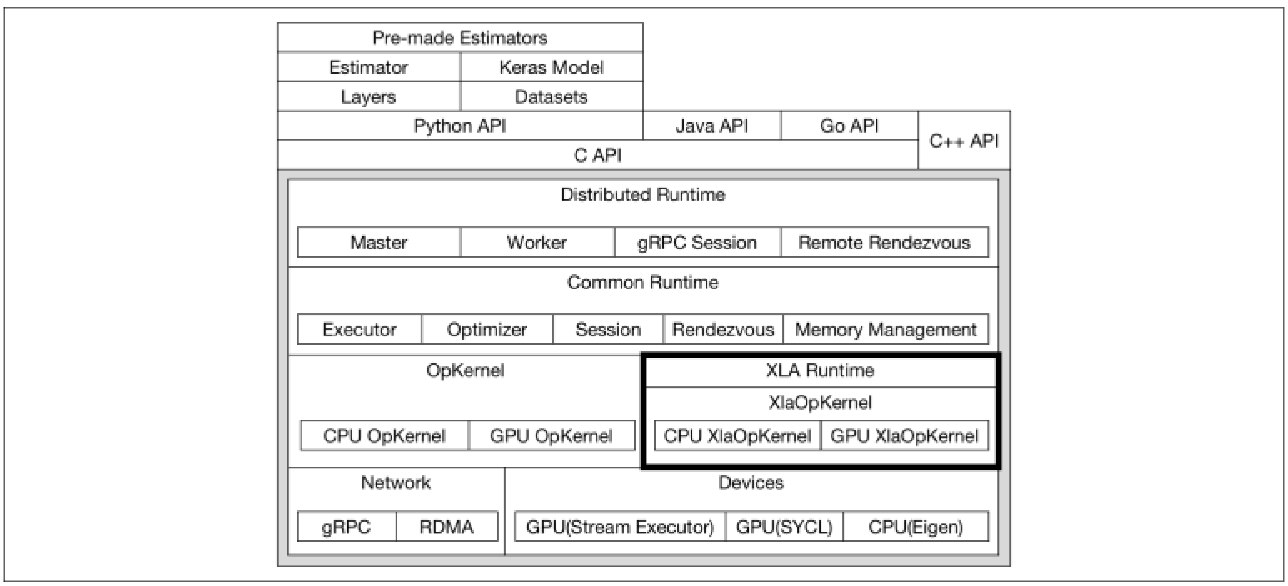
Tensorflow XLA Service Buffer优化详解

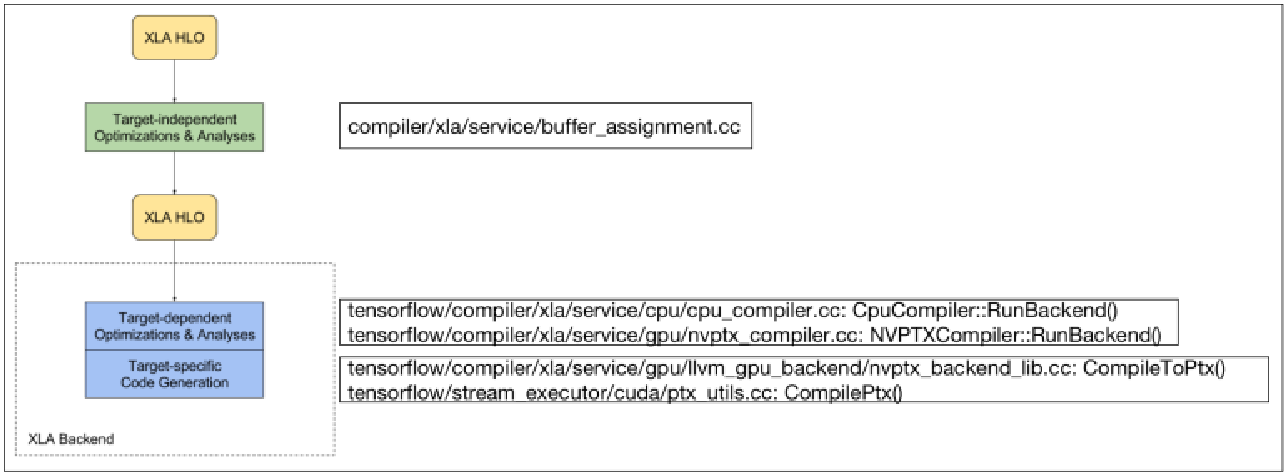
1820 Jiang XIAO

2020年3月8日 at pm6:16 (last edited 2020年6月20日 at am9:52)

下图是Tensorflow 架构图，以及XLA在Tensorflow中的位置



通过层层转换(参考Tensorflow XLA Service 详解 I)，Graph在进入XLA Service前已经被表达为HloModule的形式，而作为图编译器的核心，XLA Service就负责将HloModule表达的计算图编译为可以直接在不同硬件平台(Backend)执行的程序，而编译的核心，就是优化代码，包括设备无关的优化和设备相关的优化：



1. 优化HloModule所表示的计算图，并将其转化为LLVM HLO
2. 基于LLVM，生成硬件相关的二进制

作为通用的编译器框架，LLVM 会对LLVM HLO做大量的优化，生成高效的Target binary, 所以作为XLA的开发者，主要关注**阶段1**的优化: 设备无关的图优化算法。Compiler是XLA适配硬件的接口类，每个适配XLA的硬件都必须实现其中的方法。尤其是RunBackend()，参考Tensorflow XLA Service 详解 I 一文，该接口是进行图优化和编译的入口，统领整个优化和编译过程。

同时，XLA Service还实现了一组通用的优化方法(各种Schedule策略，各种Memory优化算法)供各个硬件平台的编译器使用，当然，主要是供给RunBackend()调用。以XLA的GPU平台的编译优化流程为例:

```
1.  NVPTXCompiler::RunBackend()
2.    hlo_schedule = GpuHloSchedule::Build(*module, *stream_assignment, pointer_size_)
3.    BufferAssigner::Run(hlo_schedule->ConsumeHloOrdering()...)
4.    entry_computation->Accept(&ir_emitter)
5.    CompileToPtx()
6.    CompilePtxOrGetCachedResult()
```

-1- 从XLA Service通用层中选择适合GPU的Schedule策略
-3- 基于Schedule策略，进行设备无关的Buffer优化，主要关注尽可能的减少Buffer的大小。注意，这里是设备无关的优化，是无法利用硬件Memory特性的。
-4- 将HloModule转化为LLVM IR
-5,6- 利用LLVM框架，将LLVM IR编译为二进制代码。

本文主要关注-3-，是XLA优化的核心。

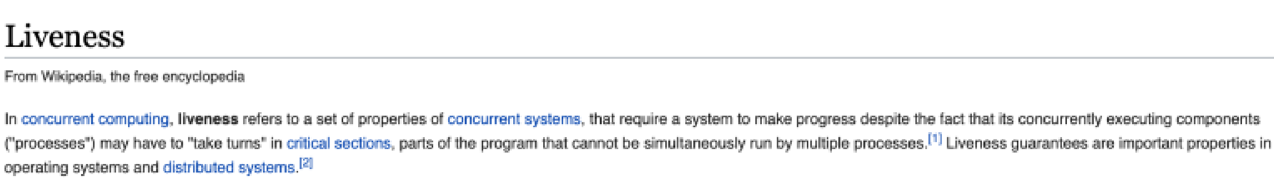
对BufferAssigner::Run()进一步分解。

```
NVPTXCompiler::RunBackend()
  hlo_schedule = GpuHloSchedule::Build(*module, *stream_assignment, pointer_size_)
  //this analysis figures out which temp buffers are required to run the computation
  BufferAssigner::Run(hlo_schedule->ConsumeHloOrdering()...)
    assigner.CreateAssignment(HloModule, hlo_ordering, buffer_size)
      liveness = BufferLiveness::Run()
      assignment = new BufferAssignment(module, liveness, ...)
      set<LogicalBuffer*> colocated_buffers
      set<BufferAllocation::Index> colocated_allocations
      vector<ColocatedBufferSet> colocated_buffer_sets
      BuildColocatedBufferSets(&colocated_buffer_sets)
```

```
12.         colorer_(assignment->liveness())
13.         AssignColocatedBufferSets(colocated_buffer_sets, assignment, &colocated_buffers
14.         GatherComputationsByAllocationType(module, &thread_local_computations, &global_
15.         for computation : global_computations:
16.           AssignBufferForComputation(computation, false, buffers_to_assign_sequentially
17.         AssignBuffersWithSequentialOrdering(buffers_to_assign_sequentially, ,assignment
18.         for computation : thread_local_computations:
19.           AssignBuffersForComputation()
20.         for buffer : assignment->liveness().maybe_live_out_buffers():
21.           if assignment->HasAllocation(buffer):
22.             assignment->GetMutableAssignedAllocation(buffer).set_mayby_live_out(true)
23.         assignment->CombineTempAllocations()
24.         return std::move(assignment)
25.     entry_computation->Accept(&ir_emitter)
26.     CompileToPtx()
27.     CompilePtxOrGetCachedResult()
```

-6- 进行BufferLiveness分析，分析整个HloModule的LogicalBuffer的干涉关系，为后续优化提供依据
-11- BuildColocatedBufferSets, 依据Bufferliveness的分析，将所有的LogicalBuffer分为几个Bufferset，并进行初步的Set融合，每个Bufferset内
参照注释, colocated buffer sets, 每个set都是一组可以共享BufferAllocation的LogicalBuffer, 共享Allocation，意味着共享同一块物理内存(GPU的显存)
-12- colorer_ 缺省被赋值为BufferLiveness::DefaultColorer(), 所有的LogicalBuffer实例的color都会被设置为0
-13- AssignColocatedBufferSets, 为Bufferset分配BufferAllocation, 每一个LogicalBufferSet 与其关联, 这里用到了buffer_size，这个函数是判断一个LogicalBuffer大小，LogicalBuffer的大小要和相应的Allocation一样，具体可以参考tf2xla/while_op.cc tf2xla/if_op.cc xla/client/builder.cc kConditional代码，可以看到明显的要求各个body的Shape要一致。通过TEST用例也能确认
-14- GatherComputationsByAllocationType，根据内含的LogicalBuffer的属性，将Allocation分为global和thread local两类，这部分是理解显存优化的关键，后文详细
-16- AssignBufferForComputation，关联Allocation和XlaComputation，此调用点只针对global,temp buffer被收集到buffers_to_assign_sequentially, 延后处理，

BufferLiveness::Run()


Liveness

From Wikipedia, the free encyclopedia

In concurrent computing, **liveness** refers to a set of properties of concurrent systems, that require a system to make progress despite the fact that its concurrently executing components ("processes") may have to "take turns" in critical sections, parts of the program that cannot be simultaneously run by multiple processes.[1] Liveness guarantees are important properties in operating systems and distributed systems.[2]

所以，BufferLiveness, 就是获取Buffer生命周期关系, 以便决定Buffer复用策略. 源码整体调用栈如下, 该阶段进一步也分3个过程: LogicalBufferAnalysis->TuplePointsToAnalysis->BufferLiveness

```
BufferAssigner::Run()
  assigner::CreateAssignment()
    liveness = BufferLiveness::Run(module, std::move(hlo_ordering)  //class BufferLiv
      liveness = new BufferLiveness()
      liveness->Analyze()
        points_to_analysis_ = TuplePointsToAnalysis::Run()
          logical_buffer_analysis = LogicalBufferAnalysis::Run()
            analysis = new LogicalBufferAnalysis()
            analysis.Analyze()
```

```
10.             return analysis
11.           analysis = new TuplePointsToAnalysis(logical_buffer_analysis)
12.           analysis.Analyze()
13.           return analysis
14.         maybe_live_out_buffers_ = points_to_analysis_->GetPointsToSet(root).CreateFla
15.       return liveness
16.     assignment(new BufferAssignment(module, std::move(liveness)
```

-1- Memory优化入口
-3-16- 获取BufferLiveness, 对比-16-, 获取到的BufferLiveness会用于支撑Buffer优化决策，用OOP的方式优雅的实现pipeline
-8- 获取LogicalBufferAnalysis
-9- 根据HloInstruction的Shape, 构造一组相应的LogicalBuffer, Shape本身是一个树结构,表示一个HloInstruction的输出形状, 一个LogicalBuffer对应一个Shape的一个节点(Subshape), 表示一块逻辑buffer, 用pair<HloInstruction*, ShapeIndex>来索引一个LogicalBuffer。
-10- 作为Liveness分析的原材料-LogicalBuffer已经构造完毕, 存储下来, 并返回analysis, 准备进入下一阶段, 进行TuplePointsToAnalysis.
-11- 用存储有所有LogicalBuffer信息的LogicalBufferAnalysis实例构造TuplePointsToAnalysis实例, **Tuple, 用来描述Buffer树状结构的方式**, E = {%1, %2, {%3, %4}} 表示这样一个树:深为3, 深1的节点有一个, 树的根, 深为2的节点有3个, %1, %2, 没体现名字的节点暂且叫"X", 深为3的节点有2个, %3, %4, 这两个节点是上一层中"X"的子节点. **PointsTo, "指向",** 在前面的例子中, root的"PointsTo"就是%1, %2和"X", "X"的"PointsTo"就是%3和%4, 所以**TuplePointsToAnalysis就是分析整个计算图中的LogicalBuffer依赖关系并存储在PointsToSet中**，后面会详细分析
-12- 执行分析逻辑.
-13- 对计算图中的LogicalBuffer的依赖关系分析完毕, 存储下来, 并返回analysis, 准备进入下一阶段, 进行BufferLiveness
-14- 用TuplePointsToAnalysis实例获取root的alias_buffer, 也就是潜在的需要传出的HloModule的Buffer, 存储在maybe_live_out_buffers_中.
-15- 返回liveness实例, TuplePointsToAnalysis实例会存储LogicalBuffer的依赖关系, 但BufferLiveness并不会存储每一个LogicalBuffer的"liveness", 而是基于TuplePointsToAnalysis封装了一组判断特定LogicalBuffer的函数.
-16- 将BufferLiveness实例传入构造LogicalBuffer与BufferAllocation映射关系的BufferAssignment实例.

上面是整个BufferLiveness分析的主干流程, 下面介绍一些关键细节

LogicalBufferAnalysis

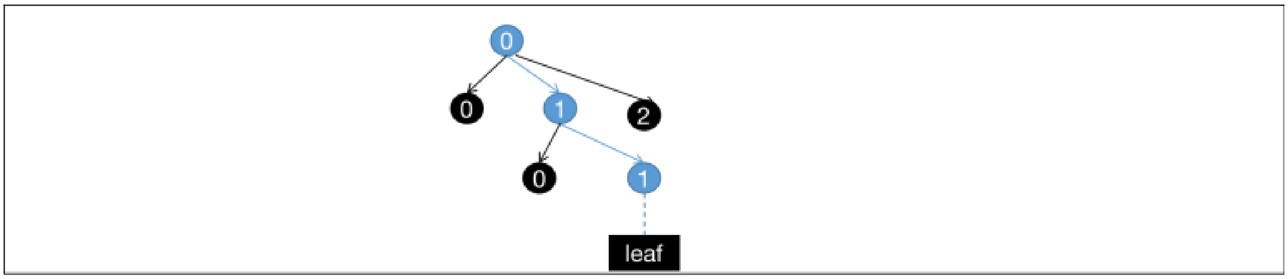
```
class LogicalBufferAnalysis : public DfsHloVisitorWithDefault
```

LogicalBufferAnalysis "**is a**" DfsHloVistorWithDefault, 后者是HLO层提供的以**visitor模式**(没错， 就是设计模式中的Visitor模式)遍历HloModule/HloComputation/HloInstruction的接口. 是XLA处理HLO优化的基础工具, 同时对于二次开发而言, 也是理解代码以及Debug的趁手工具. 对于一个"DfsHloVisitorWithDefault", 我们主要关心就是其各种"Handle"的实现, 同时, 也关心LogicalBufferAnalysis作为子类更加specific的部分.

```
//tensorflow/compiler/xla/service/logical_buffer_analysis.h
Status DefaultAction(HloInstruction* hlo_instruction) override;
Status HandleTuple(HloInstruction* tuple) override;
Status HandleGetTupleElement(HloInstruction* get_tuple_element) override;
Status HandleBitcast(HloInstruction* bitcast) override;
Status HandleDomain(HloInstruction* domain) override;
Status HandleCopy(HloInstruction* copy) override;
```

```
 8.   Status HandleRecvDone(HloInstruction* recv_done) override;
 9.   Status HandleSend(HloInstruction* send) override;
10.   Status HandleTupleSelect(HloInstruction* tuple_select) override;
```

-2- 缺省Handle, 遍历HloInstruction的每一个Subshape, 调用NewLogicalBuffer()为其构造LogicalBuffer, 每个LogicalBuffer都有LogicalBufferAnalysis内唯一的id, 由于此时一个Compilation Graph内也只有一个LogicalBufferAnalysis实例, 也就是说这个id是HloModule内唯一的. 构造的LogcialBuffer会统一存储在`'logical_buffers_'`和`'output_buffers_'`中, 二者的主要区别是索引方式不同, logical_buffers_以unique_ptr的方式存储原始的LogicalBuffer实例, 后者以std::pair<HloInstruction*, ShapeIndex>为Key, 进行Hash查找, 可快速进行LogicalBuffer检索。如果查看了ShapeIndex的实现细节，可以看到其Index并不是一个单独的"id"而是一个"absl::InlinedVector<int64, 2> indices_", 这里indices_中的每一个int64都代表HloInstruction对应的Shape多叉树的一个节点。 ShapeIndex只负责Index的部分， 至于"Leaf"是什么，由用户决定，这里，LogicalBufferAnalysis的Leaf就是LogicalBuffer，使用hash建立ShapeIndex和LogicalBuffer的关系。



-3:11- 显然, 9个"non default"的Handle都是因为缺省DefaultAction的处理方式不适用于自己所负责HloOpcode, 里面的注释很清晰, 嗯, 如果已经理解TuplePointsToAnalysis的话. LogicalBufferAnalysis和TuplePointsToAnalysis分属不同文件主要是由于使用了Visitor的处理方式, 从单个HloInstruction或者单个Buffer的角度, 设计上一脉相承, 通常要一起分析. 这里, 从理解代码的角度, 可以重点关注以下几个Handle:
-3- HandleTuple, NewLogicalBuffer(tuple, /**index=*/{});, 分配以"{}"为索引的LogcialBuffer, "{}", 看起来并不符合"ShapeIndex"的含义, 看起来没有任何"形状"信息, 实际上, 这里只需了解: 以"{}"为索引是因为kTuple只需要一个"top-level buffer", 至于什么是"top-level buffer"呢? 我在TuplePointsToAnalysis中详细分析.
-5:6- HandleBitcast 和 HandleDomain, 不构造任何LogicalBuffer.
-7- HandleCopy, 单纯构造一个新LogicalBuffer, 同样, 一个"top-level buffer"
-8- HandleRecvDone, 两个LogicalBuffer, 位于{}的top-level buffer用于以buffer alias的形式存储接收的数据, 位于{1}的buffer用于存储交互token
-9- HandleSend, 3个LogicalBuffer, 位于{}的top-level buffer用于以buffer alias的形式存储即将发送的数据, 位于{1}的buffer用于存储交互context, 位于{2}的buffer`用于存储交互token

了解了Visitor的部分, 现在来看下Specific的部分.

```
Status LogicalBufferAnalysis::Analyze();
  // We filter out fusion computations, and get to them through fusion
  // instructions. This is because it's possible to have orphaned (unreachable)
  // fusion computations, and we don't want to try to assign buffers to those.
  std::vector<HloInstruction*> fusion_instructions;
  for (auto* computation : module_->MakeNonfusionComputations()):
    computation->Accept(this);
    for (auto* instruction : computation->instructions()):
      if (instruction->opcode() != HloOpcode::kFusion):
        continue;
      GatherFusionInstructions(instruction, &fusion_instructions);
  for (auto* instruction : fusion_instructions):
    instruction->fused_expression_root()->Accept(this);
```

上面面这个函数是执行遍历的入口, 其中对kFusion进行了filter out的处理, 注释写的很清楚, 对计算图进行"剪枝", 不给 unreachable fusion computations分配LogicalBuffer, 避免浪费内存, 这样的处理在XLA很多代码中都有遇到. 这里有一个经典问题, TF runtime前期进行图处理的时候, 已经将unreachable的节点去掉了, 为什么XLA这里还要处理一遍呢? 理解的关键点在Op->HloInstruction的映射, XLA用少数几个kOpcode就组合成了几百个Op类型, 虽然TF前期已经将unreachable op去掉了, 但XLA将Op组成的计算图转化为HloInstruction组成的计算图之后, 又会出现新的unreachable节点, 这里的节点不再是Op, 而是粒度更小HloInstruction. 但即便粒度不同, TF传统引擎和XLA执行的很多图优化算法是类似的, 而二者并没有在更高的层次上进行图优化算法的抽象, 导致了这种功能重复实现. 有些同学认为从OOP的角度, 这里存在优化空间, 我的理解是恰恰相反, 这里不但不需要进一步抽象引入依赖, XLA最好将自己完全从TF引擎解耦, 以外部插件的形式接入到TF核心层. 大家感兴趣的可以给我留言讨论.

TuplePointsToAnalysis

理解这个Analysis的主要难点在于理解"**TuplePointsTo**"的含义, 理解了含义, 其内部的设计逻辑就一目了然. 对于一个HloInstruction而言, "**PointsTo**"是表达的内容: 该HloInstruction所依赖的Buffer, "**Tuple**"是表达形式: Buffer和Buffer间的拓扑关系用"`{..., {...}}`"的形式表达.

源码中, 下面这段注释有很好的解释:

```
    // For the subshape at the given index (where index is defined as in
    // ShapeUtil::GetSubshape) this method returns the set of HLO instructions
    // which may produce the tuple subshape at that index. For example, given:
    //
    // %tuple1 = tuple(...)
    // %tuple2 = tuple(...)
    // %select = select(%tuple1, %tuple2)
    // %nested_tuple = tuple(%select, %tuple1)
    //
    // These are the values for tuple_sources() for the PointsToSet of
    // %nested_tuple:
    //
    // tuple_sources({}) = {%nested_tuple}
    // tuple_sources({0}) = {%tuple1, %tuple2}
    // tuple_sources({1}) = {%tuple1}
    //
    // tuple_sources() at the index of an array shape (not a tuple) returns the
    // empty set. The instructions in the set returned by tuple_sources
    // necessarily are either Tuple instructions, constants, or parameters.
```

转化成图示, 左图就是上例图示,其中蓝色箭头就是"PointsTo"的含义, 表示的buffer索引, 所有的箭头和它对应的buffer们就是kTuple的"PointsToSet", 个人认为, 这里使用kTupleSelect并不适合作为例子, 因为它本身不是大多数情况使用的DefaultAction, 而是和kTuple一样对输入的Buffer进行alias的操作, 所以简单画了右图, 希望能表达清楚。

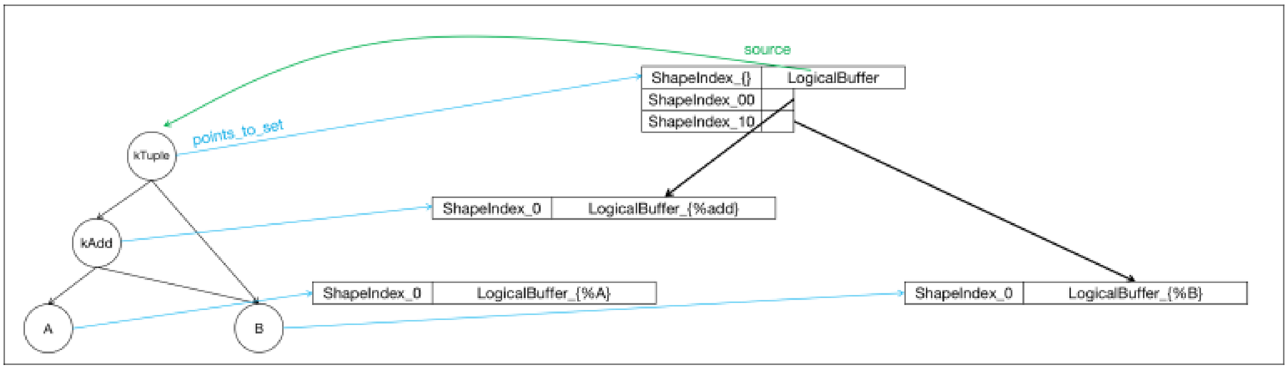
介绍了"TuplePointsTo"的含义，接下来从源码角度分析其实现细节。

```
// A class describing the source(s) of the Buffer(s) contained in the output of
// a particular HLO instruction. The structure of PointsToSet mirrors the
// structure of the instruction's shape, which may be an arbitrary tree (eg, a
// nested tuple). Each node in this tree corresponds to a single buffer in the
// instruction's output and contains the set of Buffers which might define
// the corresponding buffer.
class PointsToSet {
 private:
  struct Elem {
    BufferList buffers;
    SourceSet tuple_sources;
  };
  ShapeTree<Elem> tree_;
};
// DFS visitor that performs tuple points-to analysis. This analysis determines
// the potential sources of each buffer in each instruction's output.
class TuplePointsToAnalysis : public DfsHloVisitorWithDefault {
  // Information kept per instruction
  struct PerInstruction {
    std::unique_ptr<PointsToSet> points_to_set;
    // Empircally, ~92% of instructions have 1
    // instruction_defined_buffer, and 99% have 0 or 1
    BufferDefinitionVector instruction_defined_buffers;
  };

  // The logical buffers for this module.
  const std::unique_ptr<LogicalBufferAnalysis> logical_buffer_analysis_;

  // A map from instruction->unique_id() to
  absl::flat_hash_map<int, std::unique_ptr<PerInstruction>> per_instruction_;

  // A map from LogicalBuffer->id() to alias information about that logical
  // buffer
  std::vector<BufferAliasVector> logical_buffer_aliases_;
```

同样, TuplePointsToAnalysis也是DfsHloVisitorWithDefault, 而对于所有的Visitor, 我们都主要关注其Handle实现, 细心的同学可能发现了, 这里的Handle名与LogicalBufferAnalysis实现的一模一样, 事实上, TuplePointsToAnalysis的几个Handle实现其功能依赖于LogicalBufferAnalysis中对应的Handle的实现, 所以在看这部分代码时, 如果TuplePointsToAnalysis的某个Handle不理解, 可以去LogcialBufferAnalysis中的同名Handle看一下.

```
Status DefaultAction(HloInstruction* hlo_instruction) override;
Status HandleTuple(HloInstruction* tuple) override;
Status HandleGetTupleElement(HloInstruction* get_tuple_element) override;
Status HandleBitcast(HloInstruction* bitcast) override;
Status HandleDomain(HloInstruction* domain) override;
Status HandleCopy(HloInstruction* copy) override;
Status HandleRecvDone(HloInstruction* recv_done) override;
Status HandleSend(HloInstruction* send) override;
Status HandleTupleSelect(HloInstruction* tuple_select) override;
Status HandleAddDependency(HloInstruction* add_dependency) override;
```

-1- DefaultAction, LogicalBufferAnalysis的DefaultAction()根据Shape构造LogicalBuffer实例, 这里的DefaultAction()取其结果, 将每一个LogicalBuffer都存储在PointToSet的buffers中, 此外, 对于TupleShape的HloInstruction, 还将该HloInstruction作为其上述buffers的"source". 至于什么样XlaOp会将Shape设置为Tuple, 有这些: Infeed, InfeedWitHToken, RecvFromHost, RecvWithToken, SendToHost, SendWithToken, 都是用于大量传输数据的XlaOp. 构造PointToSet实例是作为TuplePointsToAnalysis作为DfsHloVisitorWithDefault的主要工作.
-2- 在LogicalBufferAnalysis的HandleTuple()基础上, 构造PointsToSet实例, 注意到, 1. Handle内获取LogcialBuffer的索引与LogicalBufferAnalysis中存入LogcialBuffer的索引一致, 2. 对于Tuple来说, 它的PointsToSet就是每一个operand的PointsToSet集合, 包括buffers和buffers对应的source. 这一点后面在BufferLiveness中也会用到.
-3:10- 在LogicalBufferAnalysis相应Handle的基础上, 构造相应的PointToSet实例



共性部分分析完, 同样, 下面来看下TuplePointsToSet的Specific部分. 如前文所述, Analyze()方法是入口.

```
1.  TuplePointsToAnalysis::Analyze()
2.    std::vector<HloInstruction*> fusion_instructions
3.    for computation : module_->MakeNonfusionComputations():
4.      computation->Accept(this)
5.      PopulateDefinedBuffersAndAliases(computation->instructions())
6.        for instruction : instructions:
7.          PerInstruction* pi = PerInst(instruction)
8.          GatherBuffersDefinedByInstruction(instruction, &pi->instruction_defined_buffe
9.        points_to_set = GetPointsToSet(instruction)
10.       points_to_set.ForEachElement(
11.         [this, &instruction](const ShapeIndex& index,const PointsToSet::BufferList& p
12.           for (const LogicalBuffer* buffer : pointed_to_buffers):
13.             logical_buffer_aliases_[buffer->id()].emplace_back(instruction, index)
14.         })
15.     for instruction : computation->instructions():
16.       if instruction->opcode() == HloOpcode::kFusion:
17.         GatherFusionInstructions(instruction, &fusion_instructions)
18.   // Run points-to analysis on fusion instructions in 'computation'.
19.   for instruction : fusion_instructions:
20.     instruction->fused_expression_root()->Accept(this))
21.     PopulateDefinedBuffersAndAliases(instruction->fused_instructions())
```

-3- 这里和LogicalBufferAnalysis类似, 同样进行的"剪枝"处理.
-5- 除了通过Accept()执行Handle, TuplePointsToAnalysis还会对Handle处理的结果通过PopulateDefinedBuffersAndAlias()进一步处理, 形成两个存储结构:
`'TuplePointsToSet::PerInstruction::instruction_defined_buffers'` 和
`'TuplePointsToAnalysis::logical_buffer_aliases_'`, 一个instruction_defined_buffers是通过GatherBuffersDefinedByInstruction()构造的, 里面依然是一个对所有SubShape遍历操作, 经过之前的过程, 不是可以轻松的获得一个HloInstruction对应的LogicalBuffer, 为什么这里还要遍历一次呢? 如果您有这样的疑问, 建议看

下HandleTuple()等TuplePointsToAnaysis的"定制"Handle, 对于 **kTuple** 等几个HloOpcode, 它的PointsTo的source可不是本身, 而是其operand的PointsTo的source, 显然, 这类HloInstruction没有自己的"defined buffers", 而后续进行allocation assignment的前提条件就是"buffer is defined by instruction"， 这里即是挑出会被assigned的"buffer defined by instruction", 此外，代码的实现还顺便做了合法性检查: **kTuple** etc类的HloInstruction的所有buffers的source, 都不会是自己. 至于logical_buffer_aliases_, 可以理解为一个反向检索, Alias, 别名, 也是这个意思, 可以方便用户根据LogicalBuffer->id, 快速的索引到对应的HloInstruction和ShapeIndex.

BufferLiveness

经过LogicalBufferAnalysis构造原材料(LogicalBuffer), TuplePointsToAnalysis构造原材料之间的关系(PointsToSet), 大部分苦活已经完成, BufferLiveness进过封装就可以对外提供判断一个LogcialBuffer的Liveness的接口, 主要以下几个:

```
  // Returns true if the live range of the buffer containing the output of 'a'
  // may overlap with the live range of the buffer of 'b'. If instruction 'a'
  // interferes with instruction 'b' then they cannot share the same buffer.
  bool MayInterfere(const LogicalBuffer& a, const LogicalBuffer& b) const;

  // Returns true if the buffer for the given instruction may be live out of the
  // module. That is, the instruction's buffer may be included in the output of
  // the entry computation.
  bool MaybeLiveOut(const LogicalBuffer& buffer) const;

  // Returns the complete set of buffers that may be live out of the module.
  const PointsToSet::BufferSet& maybe_live_out_buffers() const;
  // Returns the underlying points-to analysis used for this liveness analysis.
  const TuplePointsToAnalysis& points_to_analysis() const;
```

这里还是关注几个小细节.

```
1.  Status BufferLiveness::Analyze() {
2.    points_to_analysis_ = TuplePointsToAnalysis::Run(module_)
3.    for computation : module_->computations():
4.      if computation->IsFusionComputation():
5.        continue;
6.      // Gather all instructions whose buffers might alias other instructions into
7.      // the set aliased_buffers_.  This includes those contained as a tuple
8.      // element in other instruction's output.
9.      for instruction : computation->instructions():
10.       for LogicalBuffer* aliased_buffer : points_to_analysis_->GetPointsToSet(instruc
11.         if (aliased_buffer->instruction() != instruction):
12.           aliased_buffers_.insert(aliased_buffer);
13.     if computation == module_->entry_computation():
14.       HloInstruction* root = computation->root_instruction();
15.       maybe_live_out_buffers_ = points_to_analysis_->GetPointsToSet(root).CreateFlatt
```

-11- 如果一个HloInstruction->PointsToSet中->LogicalBuffer的source不是该HloInstruction, 就表示该HloInstruction是类Tuple的, 没有实质的LogicalBuffer, 也不是任何LogicalBuffer的source, 将其存储在alias_buffers_
-15- live_out_buffer就是生命周期长于entry_computation(或者理解为HloModule)的Buffer, 就是承载整个entry_computation输出数据的Buffer, root作为整个程序的总入口和总出口, 其PointsToSet的LogicalBuffers就是整个程序总输出Buffer不难理解, 但这里又为什么叫"maybe"? 代码的注释是, 后续进行执行流优化的时候, 会对hlo_ordering调整, 那么就有可能root不再作为最后的HloInstruction, 所以叫"maybe". 但根据我对XLA的理解, 没有

想到哪些优化会导致root被提前, 也许只是这个文件的程序员给自己留个后路, 毕竟万一有呢, 当然, 更可能是我忽略了一些细节, 理解不深(TODO).

下面两个是BufferLiveness和核心方法, 在后续的优化被频繁调用, 逻辑简单, 就不展开

```
  // Returns true if the live range of the buffer containing the output of 'a'
  // may overlap with the live range of the buffer of 'b'. If instruction 'a'
  // interferes with instruction 'b' then they cannot share the same buffer.
  bool MayInterfere(const LogicalBuffer& a, const LogicalBuffer& b) const;
  // Returns true if the buffer for the given instruction may be live out of the
  // module. That is, the instruction's buffer may be included in the output of
  // the entry computation.
  bool MaybeLiveOut(const LogicalBuffer& buffer) const;
```

BuildColocatedBufferSets()

这个方法的核心是将可以共享的Memory Chunk的LogicalBuffer挑出来，存在同一个BufferSet里

```
// Builds sets of buffers in 'colocated_buffer_sets' which should be colocated
// in the same allocation (currently just supports kWhile, kCall, and
// kConditional and input output aliasing).
void BufferAssigner::BuildColocatedBufferSets(
```

这句注释能看出几点信息:
1. 相同的buffer set会共享同一个allocation，即共享同一块物理内存
2. 当前只针对kWhile，kCall，和kConditional，而这三个HloOpcode的共同点是都有called_computations — 是连接entry_computation和其他computation的"铰链"
3. 该函数只会进行computation-wide 的合并，即：不会有一个ColocatedBufferSets会cross-computation ColocatedBufferSet。这一点在很多地方都能体现。

```
BuildColocatedBufferSets()
  module->input_output_alias_config().ForEachAlias({...})
  for computation : module->MakeComputationPostOrder():
    if computation->IsFusionComputation():
      continue
    for instruction : computation->MakeInstructionPostOrder():
      if kWhile:
        ...
      else if kCall:
        ...
      else if kConditional:
        ...

  MergeColocatedBufferSets()
    is_readonly_entry_parameter = [](){}
    set_can_be_merged                        //这是一个vector<bool>
    for i : colocated_buffer_sets.size():
      for buffer : colocated_buffer_sets[i]:
        if buffer_liveness.MaybeLiveOut(buffer) || is_readonly_entry_parameter(buffer
          set_can_be_merged[i] = false
          break

    cannot_merge_buffer_sets = [](){};
    interference_map
    for i : colocated_buffer_sets.size():
```

```
26.       for j : colocated_buffer_sets.size():
27.         if cannot_merge_buffer_sets(i, j):
28.           interference_map[i].push_back(j)
29.           interference_map[j].push_back(i)
30.     assigned_colors = ColorInterferenceGraph(interference_map)
31.       c_sort
32.       assigne_colors
33.       for node : nodes:
34.         for
35.         for
36.         assigned_colors[node] = color
37.         return assigned_colors
38.     for
39.       new_colocated_buffer_sets
```

-2- 处理Alias的buffer，用ShapeIndex来索引，确保相同Set内的LogicalBuffer的Shape相同，相应的Buffersize也相同
-7,9,11- 将kWhile，kCall，kConditional的buffer都按照”相同ShapeIndex的Buffer归属到相同的Bufferset”的原则构造colocated_set
-15- MergeColocatedBufferSets, 前面都是构造Buffersets，这里尝试能否将多个BufferSet进一步合并，之前每个BufferSet的构造都是独立的，
-23 cannot_merge_buffer_sets, 判断两个Bufferset能否合并，是两个bufferset能合并的”充要条件”
-24- interference_map, 标记Bufferset之间的”cannot be merged”的关系
-24:29- 使用邻接矩阵法表示两个BufferSet是否有干涉。比如，以下面的inference_map为例:

```
0:1,2
1:0,3
2:0
3:1
```

该map表示BufferSet0和BufferSet1，BufferSet2 都是cannot be merged，BufferSet3同理
-30:39- ColorInterferenceGraph，为inference_map里的set标记color，相同color的表示可以merge，用color是为了计算最终需要多少Bufferset。最终就会生成merge好的colocated_buffersets，注意，目前为止，这里还只有kWhile，kConditional，kCall的Bufferset。一开始进行按照冲突的set的数量进行排序，优先处理冲突最多的。进

优”，按照冲突的多寡排序，会保证冲突多的color值比较小，color值越大，冲突越小，而按照相同的color合并，前面的能合并的少，后面的能合并的多。举个例子:

```
1:{4,5}
2:{4,5}
3:{4}
4:{1,2,3}
5:{1,2}
```

问，4，5能否合并

答，这个其实看情况，如果3 的color比1，2小，那么4和5就不会是同一个color，也就不会合并，而如果不是这样，那么4和5就是相同的color，也就可以合并4的color和1，2怎么确定呢？这里将整体排序，按照冲突程度排序，冲突越大，color越小在本例中，4的color最小，5和4的color一样将冲突大的往前排，先分配好color，这样对于4和5这种情况，会使4和5在一个set中

AssignColocatedBufferSets()

在Computation内部，为ColotatedBufferset分配BufferAllocation，此时，ColocatedBufferset只有Alias，kCall，kWhile，kConditional几个HloOpcode

```
// For each buffer set in 'colocated_buffer_sets', assigns all buffers in the
// same set to the same buffer allocation in 'assignment'.
void AssignColocatedBufferSets(
```

```
1.  AssignColocatedBufferSets()                    //Assign, 赋值, Logical->Allocatio
2.    for colocated_buffer_set : colocated_buffer_sets:
3.      BufferAllocation* allocation = nullptr
4.      for buffer : colocated_buffer_set:
5.        if instruction->opcode() == HloOpcode::kParameter && computation == computation
6.          entry_parameter...
7.        else if instruction->opcode() == HloOpcode::kConstant:
8.          is_constant = true
9.      for buffer : colocated_buffer_set:
10.       if  allocation == nullptr:
11.         // TODO(b/32491382) Avoid current trivial solution of using new
12.         // allocations for each colocated buffer set. When liveness has
13.         // module-level scope, we can allow buffers to be shared across
14.         // computations (in some cases).
15.         allocation = assignment->NewAllocation(*buffer, buffer_size)
16.         if is_constant:
17.           allocation->set_constant(true)
18.         colocated_allocations->insert(allocation->index())
19.       else:
20.         assignment->AddAssignment(...offset = 0, buffer_size)
21.       colocated_buffers->insert(buffer)
22.     if entry_parameter_number >= 0:
23.       parameter_has_alias = ...
24.       allocation->set_entry_computation_paramter()
```

-2- 遍历 colocated_buffer_sets 中的每一个colocated_buffer_set中的每一个Logical buffer
-3- BufferAllocation* 每个colocated_buffer_set共享一个BufferAllocation
-5,7- 如果一个ColocatedBufferSet中有一个用于kParameter和kConstant的LogicalBuffer，那么整个BufferAllocation都要配置相应的域
-9- 遍历每一个 colocated_buffer_set, 如果是Set中的第一个LogicalBuffer，就通过NewAllocation创建BufferAllocation实例并保存在assignment中，否则就将LogicalBuffer关联已经构造好的BufferAllocation实例中。这里注意，这里的offset都是0，意味着同一个ColocatedBufferSet中的LogicalBuffer都是每块Memory的0地址开始使用，这也符合ColocatedBufferSet的定义，同一个ColocatedBufferSet内的任意两个Buffer都在时间上不干涉的，配置同一个地址，没有任何问题。注释: 由于当前BufferLiveness的分析都是在HloComputation内部，所以基于BufferLiveness的ColocatedBufferSets分析和BufferAllocation的构造，也不会computation-level scope的，这块还有优化空间，可以进一步压缩BufferAllocation的需求。

GatherComputationsByAllocationType()

HloComputation 在概念上类比"函数"，这个接口按照其返回reference和value的不同，将其分为global和thread-local两类，划分标准十分粗暴，还有很大的优化空间:

1. 如果一个HloComputation实例返回reference，那么该实例就是global，其内部的所有BufferAllocation，无论是否是"局部变量"，均当做"全局变量"处理(set_thread_local(false))，一同分配，优化，生命周期也和全局变量一样。体现在BufferAllocation的属性上，就是is_thread_local == false。这种划分方法会导致global的HloComputation内的生命周期较实际需要的状况被延长了，造成浪费，但能大幅简化优化逻辑。
2. 如果一个HloComputation实例返回value，那么该实例就是thread-local，其内部的所有BufferAllocation，必须全部是"局部变量"(set_thread_local(true))，简化了Buffer生命周期的管理，随着HloComputation的执行结束，"局部变量"会被回收。也正是由于栈变量可以反复的申请释放，也就无需像global一样进行全局的优化。全部是"局部变量"的约束，显得僵硬，但能大幅简化优化逻辑。

可以类比线程来理解这两个概念: global_computations –> 主线程，thread_local_computations -—> 子线程。但为什么要做这个划分呢？笔者认为可以从以下2点来理解:

1. 当前的软件层是HLO: High Level Optimization，是硬件无关的，而无论什么执行硬件，都有"执行流"的概念，也有"主执行流"的概念，只不过具体实现不同，比如CPU中有主线程和子线程，GPU中有CUDA Stream。对这些硬件公共的执行流进行抽象，就是HLO的工作，所以，这里既没有说进程，线程，CUDA Stream，而只是说global和thread_local，是一个高度抽象的执行流
2. HLO要做显存优化，一方面要知道两个Buffer是否存在必需的依赖关系(BufferLiveness)，如果没有依赖关系，就要考察两个Buffer是否在同一个执行流，简单的理解，在同一个执行流的Buffer，由于存在先后顺序，通常是可以复用的。

```
// Walk the call graph of the HLO module and place each computation into either
// thread_local_computations or global_computations depending upon whether the
// computation requires thread-local allocations or global allocations. The
// elements in thread_local_computations and global_computations are in post
// order (if computation A has an instruction which calls computation B, then A
// will appear after B in the vector).  --> thread_local 就是per thread 变量，线程不安全
GatherComputationsByAllocationType()
  while !worklist.empty():
    if is_thread_local && in_global_set ||
      ...
    if is_thread_local:
      thread_local_set...
    else:
      global_set...
    for instruction : computation->instructions():
      for subcomputation : instruction->called_computation():
        switch instruction->opcode():
          case HloOpcode::kCall:
          case HloOpcode::kConditional:
          case HloOpcode::kWhile:
            worklist.push_back(std::make_pair(subcomputation, false))
            break;
          case HloOpcode::kAllReduce:
          case HloOpcode::kMap:
          case HloOpcode::kReduce:
          case HloOpcode::kReduceWindow:
          case HloOpcode::kScatter:
          case HloOpcode::kSelectAndScatter:
          case HloOpcode::kSort:
          case HloOpcode::kFusion:
            worklist.push_back(std::make_pair(subcomputation,true))
          default:
            return InternalError()

    for computation : module->MakeComputationPostOrder():
      if thread_local_set.contains(computation):
```

```
37.         thread_local_computation->push_back(computation)
38.       else if global_set.contains(computation):
39.         global_computations->push_back(computation)
```

-26- 只处理有called_computation的情况
-28:30- 如果instruction是kCall, kConditional, kWhile, 就是global，与AssignColocatedBufferSets()相呼应
-32:39- 对于kAllreduce，kMap，kReduce，kReduceWindow，kScatter，kSelectAndScatter，kSort，kFusion都会新开thread_local

AssignBuffersForComputation()

为剩下的LogicalBuffer找Allocation，hlo_opcode.h有97个HloCode，之前主要是围绕Alias，kCall，kWhile，kConditional准备Buffer，现在要处理剩下的HloOpcode，尽可能复用之前已经分配的BufferAllocation，复用的算法就是已经分配好global_computation和thread_local_computation

1. global_computation是"main stream"，是顺序的，尽可能复用已有的BufferAllocation
2. thread_local_computation 要分配自己的

```
// 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
  for instruction : computation->instructions():
    for buffer : assignment->points_to_analysis().GetBufferDefinedByInstruction()
      sorted_buffers.push_back()
  for instruction : computation->MakeInstructionPostOrder():
    post_order_position.emplace(instruction, position)
    position++
  absl::c_sort(sorted_buffers,...)
  for buffer : sorted_buffer:
    if colocated_buffers.contains(buffer):
      continue
    if instruction->opcode() == HloOpcode::kConstant:
      BufferAllocation* allocation = assignment->NewAllocation(*buffer, buffer_size)
      allocation->set_constant(true)
      continue
    if is_entry_parameter:
      ...
      continue
    if is_thread_local:
      allocation->set_is_thread_local(true)
      continue
    //global
    if (buffer->IsTopLevel() && !buffer->IsTuple()):
      ...
    ...
    //temp
    if (!assignment->HasAllocation(*buffer) && has_sequential_order &&
        !liveness.MaybeLiveOut(*buffer)):
    (*buffers_to_assign_sequentially)[computation].insert(buffer)

    if (!assignment->HasAllocation(*buffer)):
      BufferAllocation* allocation = assignment->NewAllocation(*buffer, buffer_size)
```

-14- 如果是ColocatedBuffer，那么continue，因为在GatherComputationsByAllocationType()已经为这类(Alias, kCall, kWhile)分配了BufferAllocation(而且还是offset为0的)
-16- 处理kContant，分配新BufferAllocation
-20- 处理kParameter，分配新BufferAllocation
-23- 对于thread local，分配新的BufferAllocation
-24:27- 对于global，尽可能复用
-28- 对于一直没有assign的temp buffer，先登记，延后处理
-33- 剩下依然没有办法复用的，分配新的BufferAllocation

AssignBuffersWithSequentialOrdering()

对于前文规划好的一组顺序分配的BufferAllocation(global)，利用HeapSimulation算法，将它们压缩到一块完整的BufferAllocation中，主要针对 global 和 temp

```
// 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.
AssignBuffersWithSequentialOrdering()
  AssignBuffersFromHeapSimulator()
    BufferAllocation* allocation = assignment->NewEmptyAllocation(result.heap_size, color);
```

CombineTempAllocations()

优化的最后一步，优化Temp的BufferAllocation

```
// Combines allocations of temporary buffers of the same color into one big
// BufferAllocation.
CombineTempAllocations():
```

Related:
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 Forums

技术 Tensorflow，XLA，技术

BERT Training By Example