# (5) --- 融合框架



## 0x00 摘要

我们需要一些问题来引导分析：

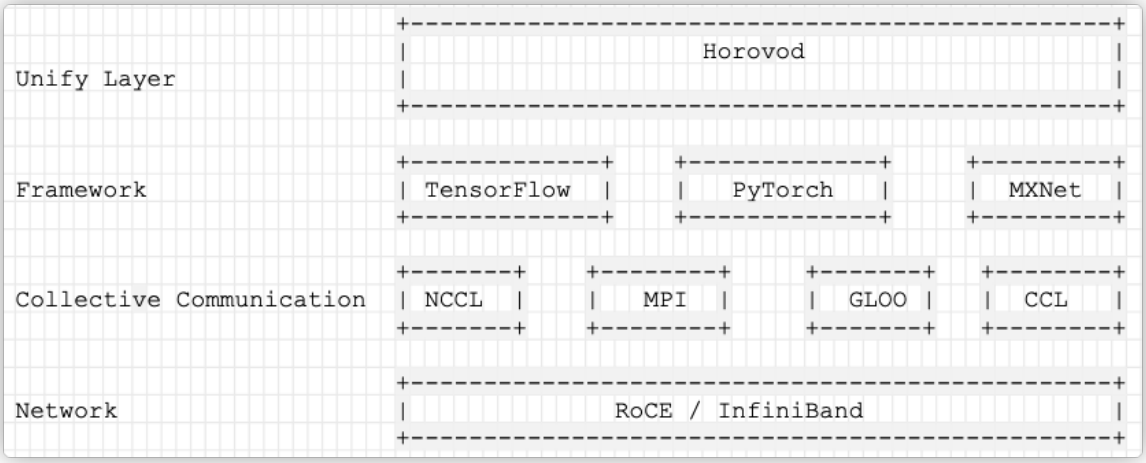
- Horovod 不依托于某个框架，自己通过MPI建立了一套分布式系统，完成了allreduce, allgather等collective operations通信工作，但是如何实现一个大统一的分布式通信框架？
- Horovod是一个库，怎么嵌入到各种深度学习框架之中？比如怎么嵌入到Tensorflow，PyTorch，MXNet，Keras？
- Horovod 因为需要兼容这么多学习框架，所以应该有自己的 OP 操作，在此基础上添加适配层，这样就可以达到兼容目的；
- 如何将梯度的同步通信完全抽象为框架无关的架构？
- 如何将通信和计算框架分离，这样，计算框架只需要直接调用hvd接口，如HorovodAllreduceOp来进行梯度求平均即可。

我们接下来看看 Horovod 如何融合。

# 0x01 架构图

我们通过架构图来看看。

以下是网上一位同学的架构图[带你了解当红炸子鸡Horovod分布式训练框架](#)，为了尽力保持风格统一，我重新绘制如下：




他分层思路如下：

- **统一层**：用来整合各个框架层，hvd将通信和计算框架分离之后，计算框架只需要直接调用hvd接口，如HorovodAllreduceOp 来进行梯度求平均即可。
- **框架层**：支持Tensorflow，PyTorch，MXNet，Keras四个热门的深度学习框架，对众多热门框架的训练支持是Horovod的优势之一。
- **多卡通信层（集合通信层）**：主要是集成一些通信框架，包括：NCCL, MPI, GLOO, CCL，主要就是完成前面说到的AllReduce的过程。
- **网络通信层**：主要是优化网络通信，提高集群间的通信效率。

MPI在Hovorod的角色比较特殊：

- 一方面Horovod内集成了基于MPI的AllReduce，类似于NCCL，都是用作梯度规约；
- 另一方面，MPI可以用来启动多个进程(Hovorod里用Rank表示)，实现并行计算；

# 0x02 统一层

我们现在知道，Horovod 内部实现（封装）了 allreduce 功能，借以实现梯度规约。

但是，hvd.allreduce又是如何实现对不同通信library的调用的呢？Horovod 使用一个统一层来完成。

首先，我们看看每个 rank 节点的运行机制，这样知道统一层的实现需要考虑哪些因素：

- 每个rank有两个thread：Execution thread 和 Background thread 。
- Execution thread 是用来做机器学习计算的。
- Background thread 是通讯和做allreduce的。
    - 后台线程中 有一个消息队列接收AllReduce，AllGather以及Broadcast等op的请求；
    - 后台线程会每隔一段时间轮询消息队列，拿到一批op之后，会对op中的tensor进行融合，再进行相应的操作。
    - 如果tensor在显存中，那么它会使用NCCL库执行。而如果是在内存中，则会使用MPI或者Gloo执行。

其次，统一层的实现是：

- 构建一个Operation 类体系，首先定义基类HVD OP，然后在此基础上定义子类AllReduceOP，并以此延伸出多个基于不同通信library的collective OP（就是适配层），比如说 GlooAllreduce 和

MPIAllReduce。

- 构建一个消息队列。所有的适配层 最后都是发出一些 Op + Tensor 的 Message 到队列中，后台初始化的时候会构建一个专门的线程（Background thread）专门消费这个队列。因此有一个同步消息的过程，相当于"某个 tensor"在所有节点上都就绪以后就可以开始计算了。
- Horovod 定义的这套HVD OP是跟具体深度学习框架无关的，Horovod 针对各个框架定义了不同的HVD OP实现。比如使用 TensorFlow时候，是无法直接插到TF Graph中执行的，所以还需要注册TF的HVD OP。

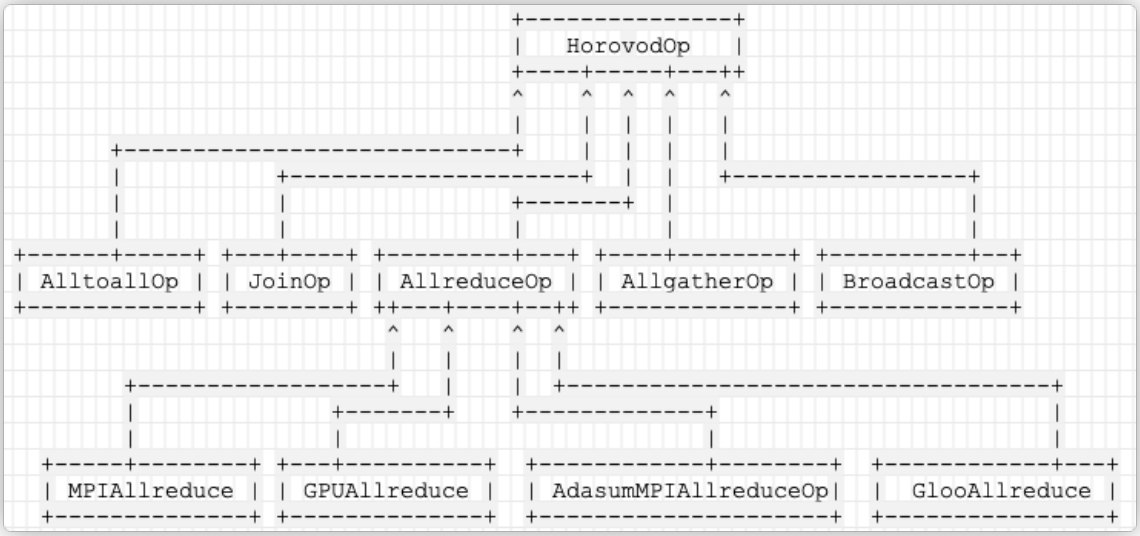
我们下面就逐一分析下这几个方面。

# 0x03 Horovod OP 类体系

Horovod OP 类体系如下：

- 首先定义基类HVD OP；
- 然后在次基础上定义子类AllReduceOP；
- 并以此延伸出多个基于不同通信library的collective OP，比如说 GlooAllreduce 和 MPIAllReduce；

逻辑如下：




## 3.1 基类 HorovodOp

HorovodOp 是所有类的基类，其主要作用是：

- 拥有 HorovodGlobalState，这样可以随时调用到总体state；
- NumElements 函数负责获取本 OP 拥有多少 tensor；
- 一个虚函数 Execute 用以被派生类实现，就是具体派生类需要实现的算法操作；

```
class HorovodOp {
public:
  HorovodOp::HorovodOp(HorovodGlobalState* global_state)
      : global_state_(global_state) {}

  int64_t HorovodOp::NumElements(std::vector<TensorTableEntry>& entries) {
    int64_t num_elements = 0;
    for (auto& e : entries) {
      num_elements += e.tensor->shape().num_elements();
    }
    return num_elements;
  }
```

```
  virtual Status Execute(std::vector<TensorTableEntry>& entries,
                         const Response& response) = 0;

protected:
  HorovodGlobalState* global_state_;
};
```

## 3.2 派生类 AllreduceOp

HorovodOp 的派生类有几个，其功能望文生义，比如：AllreduceOp ，AllgatherOp，BroadcastOp，AlltoallOp，JoinOp（弹性训练使用）。

我们以 AllreduceOp 为例，其定义如下，主要函数是：

- Execute 需要其派生类实现，就是具体进行算法操作；
- Enabled 需要其派生类实现；
- MemcpyInFusionBuffer： 用来拷贝 input Fusion tensor 多个entries；
- MemcpyOutFusionBuffer： 用来拷贝 output Fusion tensor 多个entries；
- MemcpyEntryInFusionBuffer： 用来拷贝 input Fusion tensor；
- MemcpyEntryOutFusionBuffer： 用来拷贝 output Fusion tensor；

```
class AllreduceOp : public HorovodOp {
public:
  virtual Status Execute(std::vector<TensorTableEntry>& entries,
                         const Response& response) = 0;

  virtual bool Enabled(const ParameterManager& param_manager,
                       const std::vector<TensorTableEntry>& entries,
                       const Response& response) const = 0;
protected:
  virtual void
  MemcpyInFusionBuffer(const std::vector<TensorTableEntry>& entries,
                       const void*& fused_input_data, void*& buffer_data,
                       size_t& buffer_len);
  ......
};
```

## 3.3 适配类 MPIAllreduce

接下来是具体的实现类，和具体通讯框架有关，比如：MPIAllreduce，GPUAllreduce，AdasumMPIAllreduceOp，GlooAllreduce。在 common/ops 中可以看到具体种类有 NCCL/Gloo/MPI 等等。

这些 op 由 op_manager 管理，op_manager 会根据优先级找到可以用来计算的 op 进行计算，比如：

- MPI 用的就是 MPI_Allreduce，具体 scatter-gather 和 all-gather openMPI 有现成的实现；
- NCCL 就直接调用 `ncclAllReduce`，比较新的 nccl 也支持跨节点的 allreduce 了，不用自己再套一层；

我们以 MPIAllreduce 为例进行说明，其定义如下：

```
class MPIAllreduce : public AllreduceOp {
public:
  MPIAllreduce(MPIContext* mpi_context, HorovodGlobalState* global_state);

  Status Execute(std::vector<TensorTableEntry>& entries, const Response&
response) override;

  bool Enabled(const ParameterManager& param_manager,
               const std::vector<TensorTableEntry>& entries,
               const Response& response) const override;

protected:
  MPIContext* mpi_context_;
};
```

具体 Execute 就是调用 MPI_Allreduce 来完成操作，比如：

- 从内存中拷贝到 fusion buffer；
- 调用 MPI_Allreduce 实现归并；
- 从 fusion buffer 拷贝出去；

```
Status MPIAllreduce::Execute(std::vector<TensorTableEntry>& entries, const
Response& response) {
  // Copy memory into the fusion buffer.
  ...
  MemcpyInFusionBuffer(entries, fused_input_data, buffer_data, buffer_len);
  ...

  // Do allreduce.
  timeline.ActivityStartAll(entries, MPI_ALLREDUCE);
  const void* sendbuf = entries.size() > 1 || fused_input_data == buffer_data
                        ? MPI_IN_PLACE : fused_input_data;
  int op = MPI_Allreduce(sendbuf, buffer_data,
                         (int) num_elements,
                         mpi_context_->GetMPIDataType(first_entry.tensor),
                         mpi_context_->GetMPISumOp(first_entry.tensor->dtype()),
                         mpi_context_-
>GetMPICommunicator(Communicator::GLOBAL));

  // Copy memory out of the fusion buffer.
  ...
  MemcpyOutFusionBuffer(buffer_data, entries);
  ...
}
```

## 3.4 后台线程如何使用

因为 Horovod 主要是由一个后台线程完成梯度操作，所以让我们看看这个后台线程之中如何调用到 Hovorod OP。

Horovod的工作流程比较简单：

- HorovodGlobalState 之中有一个消息队列接收AllReduce，AllGather以及Broadcast等op的请求。
- 有一个后台线程会每隔一段时间轮询消息队列，拿到一批op之后，会对op中的tensor进行融合，再进行相应的操作。

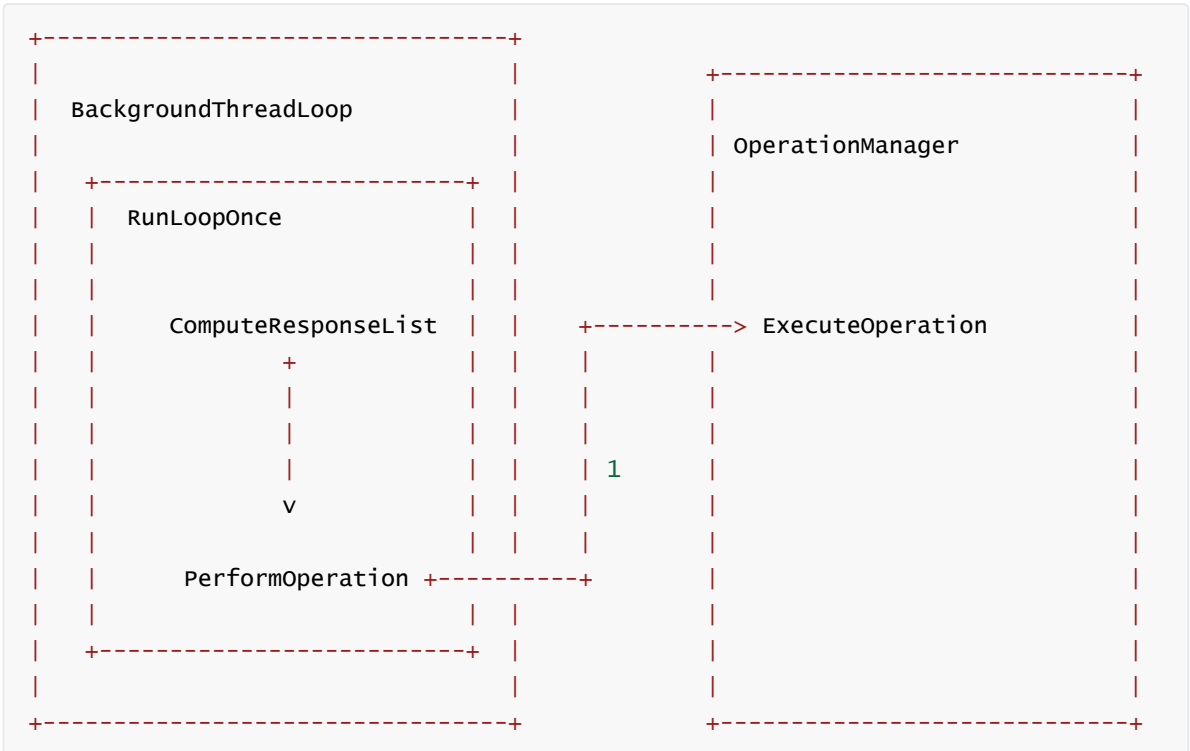
- 如果tensor在显存中，那么它会使用NCCL库执行。而如果是在内存中，则会使用MPI或者Gloo执行。

### 3.4.1 具体collective 操作

Horovod 的后台线程拿到需要融合的tensor 之后，会调用 PerformOperation 进行具体的collective 操作。在 PerformOperation 之中有调用

```
void PerformOperation(Response response, HorovodGlobalState& state) {
  ......
  Status status;
  try {
    // 进行collective的操作
    status = op_manager->ExecuteOperation(entries, response);
  } catch (const std::exception& ex) {
    status = Status::UnknownError(ex.what());
  }
  ......
}
```

逻辑如下：




### 3.4.2 调用不同类型的OP

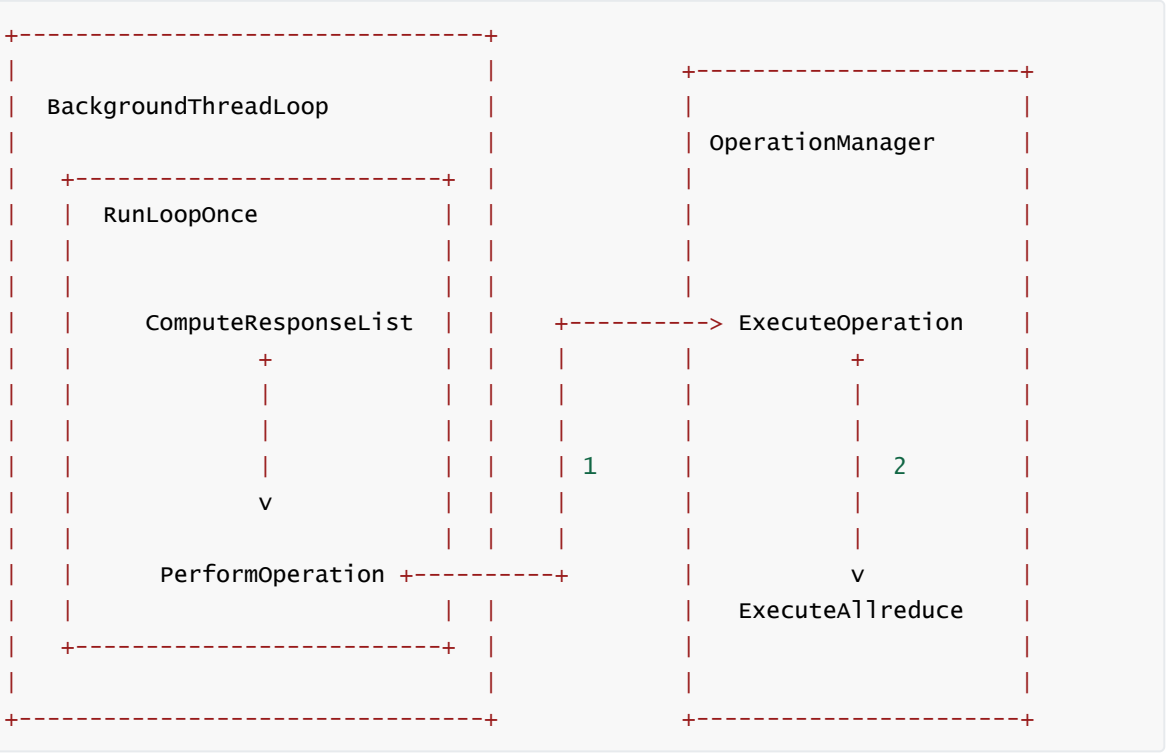
然后 status = op_manager->ExecuteOperation(entries, response) 会调用不同的 op->Execute(entries, response) 执行reduce 运算。

比如 ALLREDUCE 就调用了 ExecuteAllreduce(entries, response)。

```
Status OperationManager::ExecuteOperation(std::vector<TensorTableEntry>&
entries,
                                          const Response& response) const {
  if (response.response_type() == Response::ALLREDUCE) {
    return ExecuteAllreduce(entries, response); // 这里
  } else if (response.response_type() == Response::ALLGATHER) {
```

```
    return ExecuteAllgather(entries, response);
  } else if (response.response_type() == Response::BROADCAST) {
    return ExecuteBroadcast(entries, response);
  } else if (response.response_type() == Response::ALLTOALL) {
    return ExecuteAlltoall(entries, response);
  } else if (response.response_type() == Response::JOIN) {
    return ExecuteJoin(entries, response);
  }
  .....
}
```

逻辑如下：




### 3.4.3 取一个适配层

具体就是从 allreduce_ops_ 之中选取一个合适的 op，调用其Execute。

```
Status OperationManager::ExecuteAllreduce(std::vector<TensorTableEntry>&
entries,
                                          const Response& response) const {
  for (auto& op : allreduce_ops_) {
    if (op->Enabled(*param_manager_, entries, response)) {
      return op->Execute(entries, response);
    }
  }
}
```

allreduce_ops_ 是从哪里来的？在 OperationManager 构建函数中有。

```
allreduce_ops_(std::move(allreduce_ops)),
```

所以我们看看allreduce_ops 如何构建。

### 3.4.4 适配层构建

在 CreateOperationManager 之中对 allreduce_ops 进行添加。

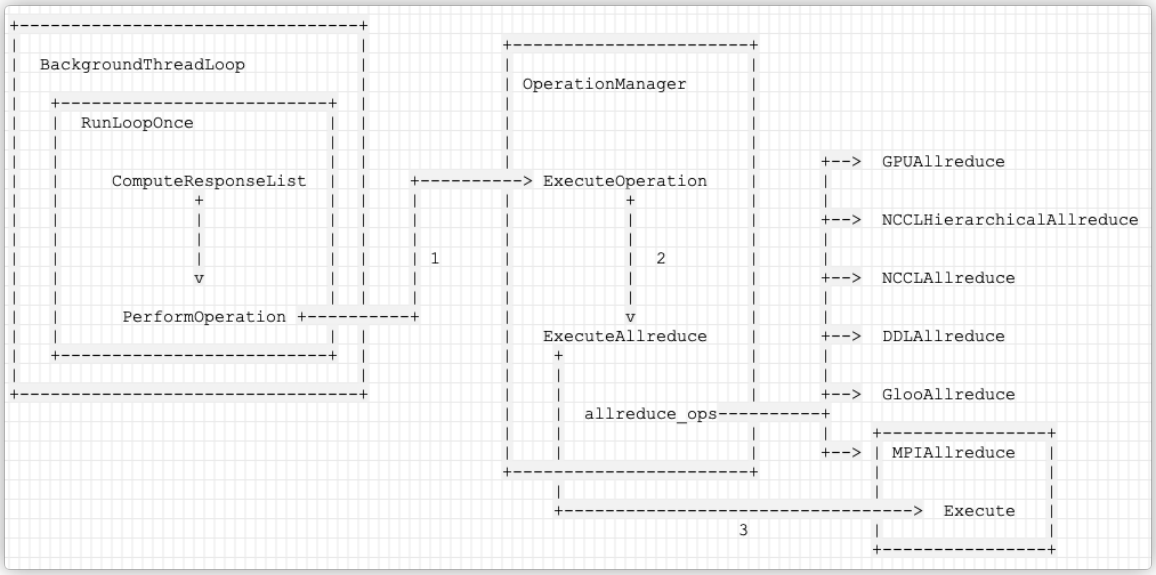
可以看到，添加的类型大致如下：

- MPI_GPUAllreduce
- NCCLHierarchicalAllreduce
- NCCLAllreduce
- DDLAllreduce
- GlooAllreduce
- GPUAllreduce
- MPIAllreduce
- ……

```
OperationManager* CreateOperationManager(HorovodGlobalState& state) {
  // Order of these operations is very important. Operations will be checked
  // sequentially from the first to the last. The first 'Enabled' operation will
  // be executed.
  std::vector<std::shared_ptr<AllreduceOp>> allreduce_ops;
  std::vector<std::shared_ptr<AllgatherOp>> allgather_ops;
  std::vector<std::shared_ptr<BroadcastOp>> broadcast_ops;
  std::vector<std::shared_ptr<AllreduceOp>> adasum_ops;
  std::vector<std::shared_ptr<AlltoallOp>> alltoall_ops;

#if HAVE_MPI && HAVE_GPU // 如果配置了MPI
  if (mpi_context.IsEnabled()) {
#if HOROVOD_GPU_ALLREDUCE == 'M'
    allreduce_ops.push_back(std::shared_ptr<AllreduceOp>(
        new MPI_GPUAllreduce(&mpi_context, &gpu_context, &state)));

    allreduce_ops.push_back(
        std::shared_ptr<AllreduceOp>(new NCCLHierarchicalAllreduce(
            &nccl_context, &mpi_context, &gpu_context, &state)));

#elif HAVE_DDL && HOROVOD_GPU_ALLREDUCE == 'D'  //如果配置了DDL
    allreduce_ops.push_back(std::shared_ptr<AllreduceOp>(
        new DDLAllreduce(&ddl_context, &gpu_context, &state)));
#endif

#if HAVE_NCCL && HOROVOD_GPU_ALLREDUCE == 'N'//如果配置了NCCL
  allreduce_ops.push_back(std::shared_ptr<AllreduceOp>(
      new NCCLAllreduce(&nccl_context, &gpu_context, &state)));
#endif

.......
```

因此我们知道，如何使用这些 Operation。

流程如下：




回顾下每个 rank 节点的运行机制，每个rank有两个thread：

- Execution thread 是用来做机器学习计算的。
- background thread 是负责通讯和allreduce。

到目前为止，我们其实分析的是第二部分：background thread 是负责通讯和allreduce。

下面我们要看看第一部分的某些环节，即 Tensorflow 这样的框架是如何把 tensor & op 发送给 后台线程。

# 0x04 与通讯框架融合

Horovod 定义的这套HVD OP是跟具体深度学习框架无关的，比如使用 TensorFlow时候，是无法直接insert到TF Graph中执行的，所以还需要注册TF的OP。

Horovod 针对各个框架定义了不同的实现。

针对 TensorFlow 模型分布式训练，Horovod 开发了 TensorFlow ops 来实现 Tensorflow tensor 的 AllReduce。而且这些 op 可以融入 TensorFlow 的计算图中，利用 TensorFlow graph 的 runtime 实现计算与通信的 overlapping，从而提高通信效率。

以 TensorFlow 模型的 AllReduce 分布式训练为例，Horovod 开发了 allreduce ops 嵌入 TensorFlow 的反向计算图中，从而获取 TensorFlow 反向计算的梯度并进行梯度汇合。allreduce ops 可以通过调用 gloo 提供的 allreduce API 来实现梯度汇合的。

比如 在 horovod/tensorflow/mpi_ops.cc 之中，就针对 tensorflow 定义了 **HorovodAllreduceOp**。

## 4.1 TensorFlow 定义Op

对于 TensorFlow，可以自定义 Operation，即如果现有的库没有涵盖你想要的操作, 你可以自己定制一个。

为了使定制的 Op 能够兼容原有的库，你必须做以下工作:

- 在一个 C++ 文件中注册新 Op. Op 的注册与实现是相互独立的. 在其注册时描述了 Op 该如何执行. 例如, 注册 Op 时定义了 Op 的名字, 并指定了它的输入和输出.
- 使用 C++ 实现 Op. 每一个实现称之为一个 "kernel", 可以存在多个 kernel, 以适配不同的架构 (CPU, GPU 等)或不同的输入/输出类型.
- 创建一个 Python 包装器（wrapper）. 这个包装器是创建 Op 的公开 API. 当注册 Op 时, 会自动生成一个默认 默认的包装器. 既可以直接使用默认包装器, 也可以添加一个新的包装器.
- (可选) 写一个函数计算 Op 的梯度.

- (可选) 写一个函数, 描述 Op 的输入和输出 shape. 该函数能够允许从 Op 推断 shape.
- 测试 Op, 通常使用 Pyhton。如果你定义了梯度，你可以使用Python的GradientChecker来测试它。

## 4.2 Horovod 实现 --- HorovodAllreduceOp

HorovodAllreduceOp 就是一种TF Async OP，然后其内部实现中调用了 HVD OP，这是比较巧妙的**组合模式**。显然继承了TP Aysnc OP的HorovodAllReduce 是可以插入到TF Graph里面，然后被正常执行的。

添加新的OP需要3步，我们具体看看。

### 4.2.1 定义 Op 的接口

第一步是定义Op 的接口，使用REGISTER_OP()向 TensorFlow 系统注册来定义 Op 的接口，该OP就是HorovodAllreduceOp。

```
// 1. 定义 Op 的接口
//    REGISTER_OP()向 TensorFlow 系统注册来定义 Op 的接口，该OP就是HorovodAllreduceOp.
//    在注册时，指定 Op 的名称: REGISTER_OP("HorovodAllreduce")
//                      输入(类型和名称): Input("tensor: T")
//                      输出(类型和名称): Output("sum: T")
//                      和所需要任何 属性的文档说明Doc(R"doc(...)doc");
//
//    该 Op 接受一个 T 类型 tensor 作为输入，T 类型可以是{int32, int64, float32,
float64}
//          输出一个 T 类型 tensor sum，sum是在所有的MPI进程中求和

REGISTER_OP("HorovodAllreduce")
    .Attr("T: {int32, int64, float16, float32, float64}")
    .Attr("reduce_op: int")
    .Attr("prescale_factor: float")
    .Attr("postscale_factor: float")
    .Attr("ignore_name_scope: bool = False")
    .Input("tensor: T")
    .Output("sum: T")
    .SetShapeFn([](shape_inference::InferenceContext* c) {
      c->set_output(0, c->input(0));
      return Status::OK();
    });
```

### 4.2.2 为 Op 实现 kernel

第二步是为 Op 实现 kernel。在定义接口之后, 每一个实现称之为一个 "kernel"，提供一个或多个 Op 的实现，即可以存在多个 kernel。

HorovodAllreduceOp 类继承 AsyncOpKernel，覆盖 其ComputeAsync() 方法。ComputeAsync()方法提供一个类型为 OpKernelContext* 的参数 context, 用于访问一些有用的信息, 例如输入和输出的 tensor。

在 ComputeAsync 里，会把这一 AllReduce 的请求入队。可以看到，在 TensorFlow 支持的实现上，Horovod 与百度大同小异。都是自定义了 AllReduce Op，在 Op 中把请求入队。

```
// 2. 为 Op 实现 kernel。
//    在定义接口之后，每一个实现称之为一个 "kernel"，提供一个或多个 Op 的实现，即可以存在多个
kernel。
```

```
//    为这些 kernel 的每一个创建一个对应的类，继承 AsyncOpKernel，覆盖 ComputeAsync 方法。
//    ComputeAsync 方法提供一个类型为 OpKernelContext* 的参数 context，用于访问一些有用的信息，例如输入和输出的 tensor

class HorovodAllreduceOp : public AsyncOpKernel {
public:
  // 防止类构造函数的隐式自动转换，只能显示调用该构造函数
  explicit HorovodAllreduceOp(OpKernelConstruction* context)
      : AsyncOpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("reduce_op", &reduce_op_));
    OP_REQUIRES_OK(context, context->GetAttr("prescale_factor",
&prescale_factor_));
    OP_REQUIRES_OK(context, context->GetAttr("postscale_factor",
&postscale_factor_));
    OP_REQUIRES_OK(context, context->GetAttr("ignore_name_scope",
&ignore_name_scope_));
  }

  // 重写ComputeAsync()方法
  void ComputeAsync(OpKernelContext* context, DoneCallback done) override {
    OP_REQUIRES_OK_ASYNC(context, ConvertStatus(common::CheckInitialized()),
                         done);

    auto node_name = name();
    if (ignore_name_scope_) {
      auto pos = node_name.find_last_of('/');
      if (pos != std::string::npos) {
        node_name = node_name.substr(pos + 1);
      }
    }
    auto device = GetDeviceID(context);
    auto tensor = context->input(0);
    horovod::common::ReduceOp reduce_op = static_cast<horovod::common::ReduceOp>
(reduce_op_);
    Tensor* output;
    OP_REQUIRES_OK_ASYNC(
        context, context->allocate_output(0, tensor.shape(), &output), done);
    // ReadyEvent makes sure input tensor is ready, and output is allocated.
    // shared_ptr 是一个标准的共享所有权的智能指针，允许多个指针指向同一个对象
    auto ready_event = std::shared_ptr<common::ReadyEvent>
(RecordReadyEvent(context));
    // 模板函数 std::make_shared 可以返回一个指定类型的 std::shared_ptr
    auto hvd_context = std::make_shared<TFOpContext>(context);
    auto hvd_tensor = std::make_shared<TFTensor>(tensor);
    auto hvd_output = std::make_shared<TFTensor>(*output);

    // 将张量的Allreduce操作OP加入队列
    auto enqueue_result = EnqueueTensorAllreduce(
        hvd_context, hvd_tensor, hvd_output, ready_event, node_name, device,
        [context, done](const common::Status& status) {
          context->SetStatus(ConvertStatus(status));
          done();
        }, reduce_op, (double) prescale_factor_, (double) postscale_factor_);
    OP_REQUIRES_OK_ASYNC(context, ConvertStatus(enqueue_result), done);
  }

private:
```

```
  int reduce_op_;
  // Using float since TF does not support double OP attributes
  float prescale_factor_;
  float postscale_factor_;
  bool ignore_name_scope_;
};
```

### 4.2.3 注册OP到 TensorFlow 系统

第三步是注册OP到 TensorFlow 系统。

```
// 3. 注册OP到 TensorFlow 系统
//    注册时可以指定该 kernel 运行时的多个约束条件. 例如可以指定一个 kernel 在 CPU 上运行，另一个在 GPU 上运行
REGISTER_KERNEL_BUILDER(Name("HorovodAllreduce").Device(DEVICE_CPU),
                        HorovodAllreduceOp);
// 如果执行了GPU
#if HOROVOD_GPU_ALLREDUCE
REGISTER_KERNEL_BUILDER(Name("HorovodAllreduce").Device(DEVICE_GPU),
                        HorovodAllreduceOp);
#endif
```

### 4.2.4 注意点

具体可以参考 add new op，里面规范了 Tensorflow 自定义算子的实现。

> 请注意，生成的函数将获得一个蛇形名称（以符合 PEP8）。因此，如果您的操作在 C++ 文件中命名为 ZeroOut，则 Python 函数将称为 zero_out。

C++ 的定义是驼峰的，生成出来的 python 函数是下划线小写的，所以最后对应的是，适配Op的代码在 horovod/tensorflow 目录下面。

| C++ | Python |
| --- | --- |
| HorovodAllgather | horovod_allgather |
| HorovodAllreduce | horovod_allreduce |
| HorovodBroadcast | horovod_broadcast |

**所以，在 python 世界中，当 _DistributedOptimizer 调用 compute_gradients 来优化的时候，会通过 _allreduce 来调用到 MPI_LIB.horovod_allreduce，也就是调用到 HorovodAllreduceOp** 这里。

具体 _DistributedOptimizer 如何调用到 _allreduce，我们在后续文章中会讲解。

```
def _allreduce(tensor, name=None, op=Sum):
    if name is None and not _executing_eagerly():
        name = 'HorovodAllreduce_%s' % _normalize_name(tensor.name)
    return MPI_LIB.horovod_allreduce(tensor, name=name, reduce_op=op)
```

## 4.3 如何使用

### 4.3.1 EnqueueTensorAllreduce

HorovodAllreduceOp 类会调用 EnqueueTensorAllreduce() 方法，将张量的Allreduce操作OP加入HorovodGlobalState的队列中。

EnqueueTensorAllreduce 位于：/horovod/common/operations.cc。

具体方法就是构建contexts，callbacks等各种支撑数据，然后调用 EnqueueTensorAllreduces 进行处理。

```
// Contexts and controller must be initialized and the background thread
// must be running before this function is called.
Status EnqueueTensorAllreduce(std::shared_ptr<OpContext> context,
                              std::shared_ptr<Tensor> tensor,
                              std::shared_ptr<Tensor> output,
                              std::shared_ptr<ReadyEvent> ready_event,
                              std::string name, const int device,
                              StatusCallback callback,
                              ReduceOp reduce_op,
                              double prescale_factor,
                              double postscale_factor) {
  // Wrap inputs in std::vector and pass onto multi tensor implementation
  std::vector<std::shared_ptr<OpContext>> contexts;
  std::vector<std::shared_ptr<Tensor>> tensors;
  std::vector<std::shared_ptr<Tensor>> outputs;
  std::vector<std::shared_ptr<ReadyEvent>> ready_events;
  std::vector<std::string> names;
  std::vector<StatusCallback> callbacks;

  contexts.emplace_back(std::move(context));
  tensors.emplace_back(std::move(tensor));
  outputs.emplace_back(std::move(output));
  ready_events.emplace_back(std::move(ready_event));
  names.emplace_back(std::move(name));
  callbacks.emplace_back(std::move(callback));

  return EnqueueTensorAllreduces(contexts, tensors, outputs, ready_events,
                                 names, device, callbacks, reduce_op,
                                 prescale_factor, postscale_factor);
}
```

### 4.3.2 提交命令

EnqueueTensorAllreduces 主要就是调用 AddToTensorQueueMulti 向 tensor queue 提交操作，方法逻辑为：

- 把需要 reduce 的 tensor 组装成一个Request。
- 针对每个 tensor，会创建对应 TensorTableEntry，用于保存tensor 的权重，message 主要是一些 元信息 metadata。
- 把 request 和 TensorTableEntry往 GlobalState 的 tensor_queue 里面塞，这是一个进程内共享的全局对象维护的一个队列。
- 等待后台线程去读取这些allreduce 的请求。后台进程，会一直在执行一个循环 `RunLoopOnce`。在其中，后台线程会利用 MPIController 来处理入队的请求。 MPIController 可以理解为是协调不

同的 Rank 进程，处理请求的对象。这个抽象是百度所不具备的，主要是为了支持 Facebook gloo 等其他的集合计算库。因此 Horovod 也有 GlooController 等等实现。

具体代码如下：

```
Status EnqueueTensorAllreduces(std::vector<std::shared_ptr<OpContext>>&
contexts,
                               std::vector<std::shared_ptr<Tensor>>& tensors,
                               std::vector<std::shared_ptr<Tensor>>& outputs,
                               std::vector<std::shared_ptr<ReadyEvent>>&
ready_events,
                               std::vector<std::string>& names,
                               const int device,
                               std::vector<StatusCallback>& callbacks,
                               ReduceOp reduce_op,
                               double prescale_factor,
                               double postscale_factor) {
  Status status;

  ......

  std::vector<Request> messages;
  std::vector<TensorTableEntry> entries;
  messages.reserve(tensors.size());
  entries.reserve(tensors.size());

  for (int n = 0; n < tensors.size(); ++n) { // 遍历需要 reduce 的 tensor
    // 把tensor组装成一个Request
    Request message;
    message.set_request_rank(horovod_global.controller->GetRank());
    message.set_tensor_name(names[n]);
    message.set_tensor_type(tensors[n]->dtype());
    message.set_device(device);
    message.set_prescale_factor(prescale_factor);
    message.set_postscale_factor(postscale_factor);

    if (reduce_op == ReduceOp::ADASUM) {
      message.set_request_type(Request::ADASUM);
    } else {
      message.set_request_type(Request::ALLREDUCE);
    }

    message.set_tensor_shape(tensors[n]->shape().to_vector());
    messages.push_back(std::move(message));

    TensorTableEntry e;
    e.tensor_name = names[n];
    e.context = std::move(contexts[n]);
    // input and output can be the same, only move when safe
    if (tensors[n] != outputs[n]) {
      e.tensor = std::move(tensors[n]);
      e.output = std::move(outputs[n]);
    } else {
      e.tensor = tensors[n];
      e.output = outputs[n];
    }
    e.ready_event = std::move(ready_events[n]);
```

```
    e.device = device;
    e.callback = std::move(callbacks[n]);

    // 针对每个 tensor，会创建对应 TensorTableEntry，用于保存tensor 的权重，message 主要是一些 元信息 metadata
    entries.push_back(std::move(e));
  }

  std::string tensors_enqueued;
  for (const auto& n : names) {
    tensors_enqueued += n + "; ";
  }

  // Only create groups larger than 1 tensor, unless disable_group_fusion is requested.
  // In that case, even single tensor groups are created to enforce disabling fusion.
  if (tensors.size() > 1 || horovod_global.disable_group_fusion) {
    auto group_id = horovod_global.group_table.RegisterGroup(std::move(names));
    for (auto& message : messages) {
      message.set_group_id(group_id);
    }
  }

  // 往 GlobalState 的 tensor_queue 里面添加
  status = horovod_global.tensor_queue.AddToTensorQueueMulti(entries, messages);

  return status;
}
```

### 4.3.3 TensorQueue

Tensor 和 op 主要是添加到 TensorQueue，具体就是调用 如下：

```
status = horovod_global.tensor_queue.AddToTensorQueueMulti(entries, messages);
```

AddToTensorQueue 和 AddToTensorQueueMulti 函数基本逻辑类似，只不过后者是处理多个 message，具体如下：

- 将MPI Request message请求加入 horovod_global.message_queue；
- 将TensorTableEntry e 加入horovod_global.tensor_table ；

```
// Add a TensorTableEntry as well as its message to the queue.
Status TensorQueue::AddToTensorQueue(TensorTableEntry& e, Request& message) {
  std::lock_guard<std::mutex> guard(mutex_);
  if (tensor_table_.find(e.tensor_name) != tensor_table_.end()) {
    return DUPLICATE_NAME_ERROR;
  }
  tensor_table_.emplace(e.tensor_name, std::move(e));
  message_queue_.push(std::move(message));
  return Status::OK();
}

Status TensorQueue::AddToTensorQueueMulti(std::vector<TensorTableEntry>& entries,
                                          std::vector<Request>& messages) {
```

```
  std::lock_guard<std::mutex> guard(mutex_);

  for (int i = 0; i < entries.size(); ++i) {
    if (tensor_table_.find(entries[i].tensor_name) != tensor_table_.end()) {
      return DUPLICATE_NAME_ERROR;
    }
    tensor_table_.emplace(entries[i].tensor_name, std::move(entries[i]));
    message_queue_.push(std::move(messages[i]));
  }
  return Status::OK();
}
```

这样就添加到了 message queue，我们的逻辑也完成了。

# 0x05 总结

总结Horovod的梯度同步更新以及AllReduce操作的全过程如下：

- 首先HVD定义TF异步的AllReduce OP，通过wrap optimizer将AllReduce OP插入到TF execution Graph中；
- OP内部主要就是把All Reduce需要的信息打包成Request，发送给coordinator（Rank0）；
- 由Rank0协调所有Rank的request，并在所有Rank都Ready后，发送Response让各个Rank执行AllReduce操作。

具体如下图：