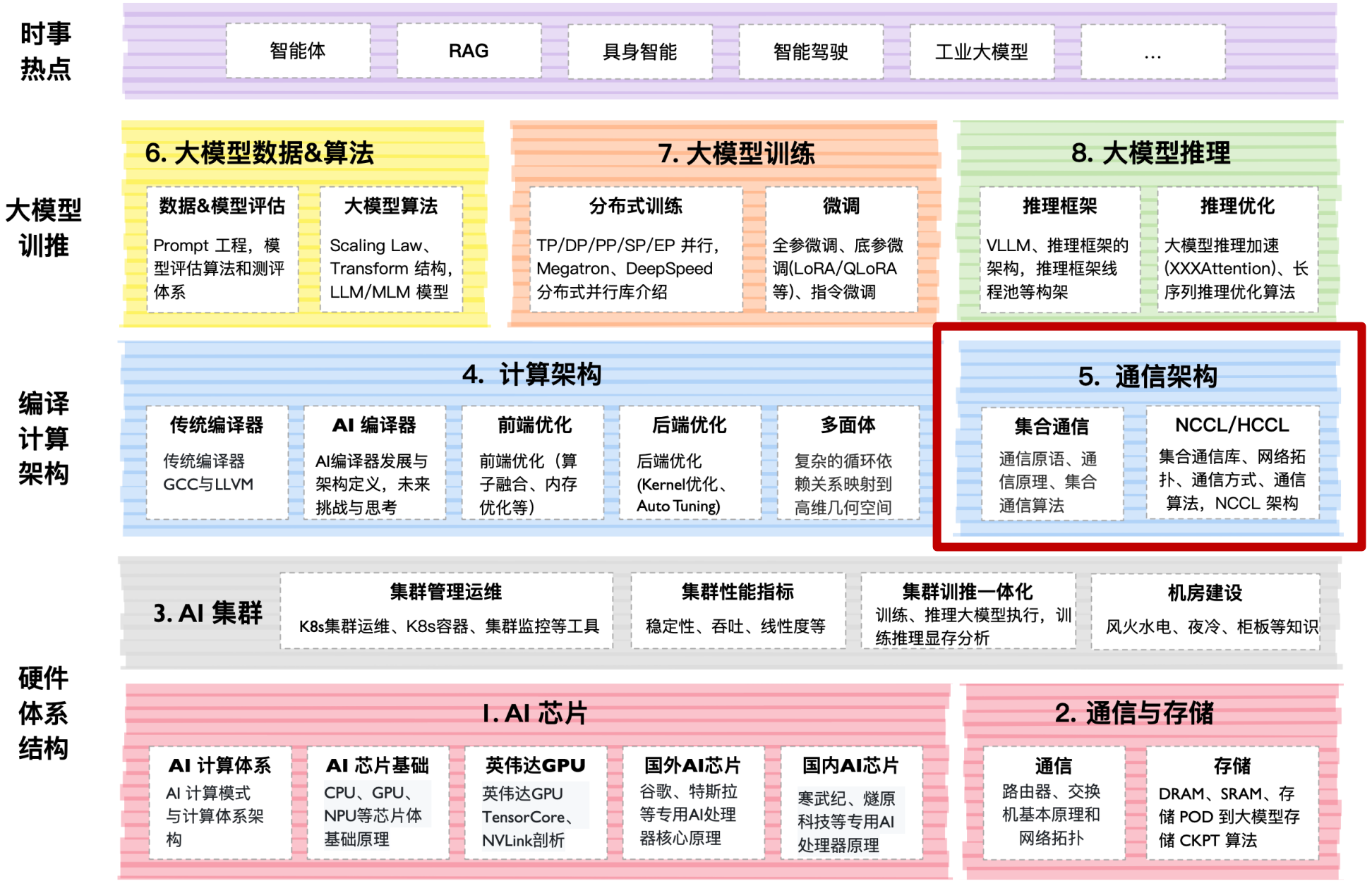
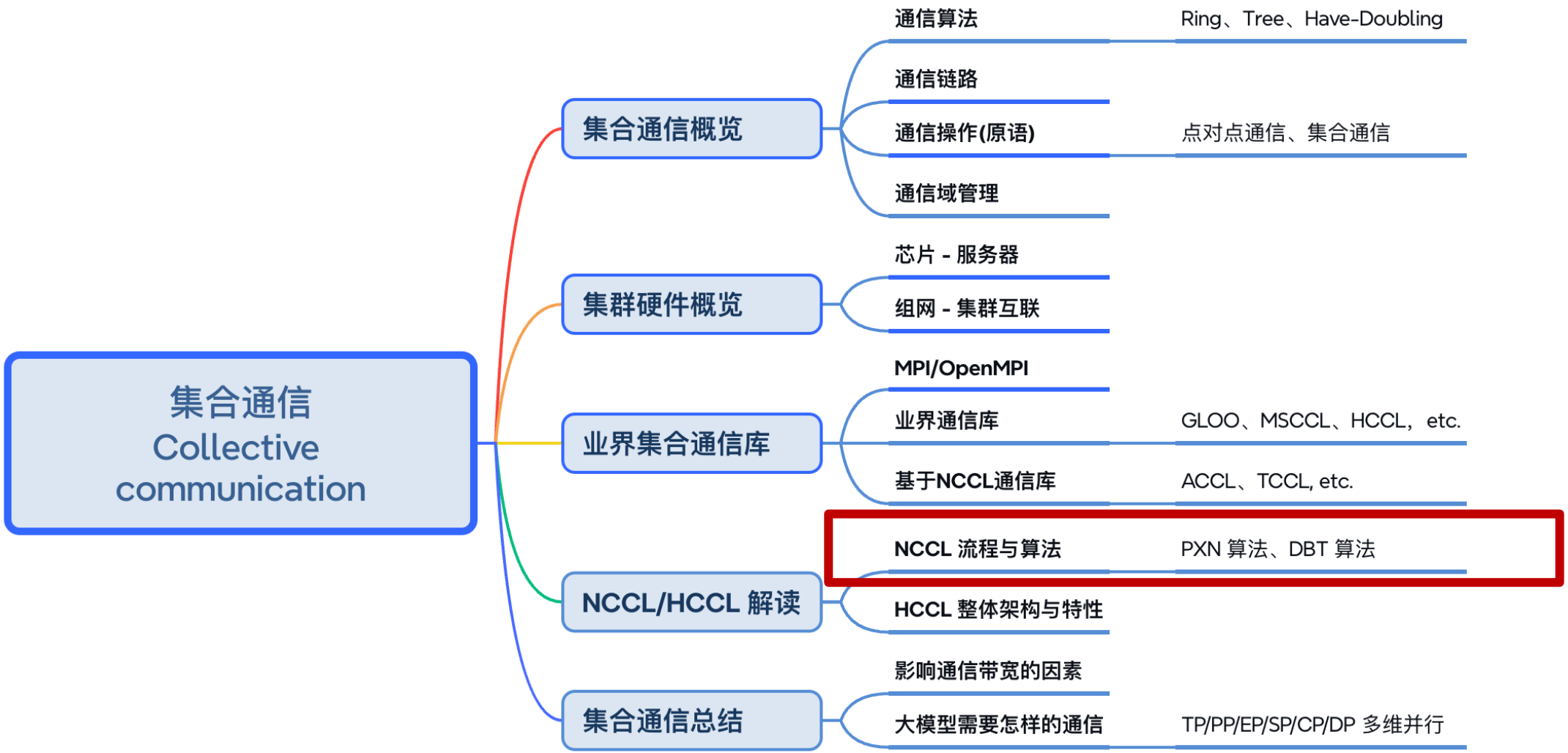

大模型系列 – 集合通信库
NCCL API 解读
NVIDIA
ZOMI


时事热点
智能体
RAG
具身智能
智能驾驶
工业大模型
...
大模型训推
6. 大模型数据&算法
数据&模型评估
Prompt 工程，模型评估算法和测评体系
大模型算法
Scaling Law、Transform 结构，LLM/MLM 模型
7. 大模型训练
分布式训练
TP/DP/PP/SP/EP 并行，Megatron、DeepSpeed 分布式并行库介绍
微调
全参微调、底参微调(LoRA/QLoRA 等)、指令微调
8. 大模型推理
推理框架
VLLM、推理框架的架构，推理框架线程池等构架
推理优化
大模型推理加速(XXXAttention)、长序列推理优化算法
编译计算架构
4. 计算架构
传统编译器
传统编译器 GCC与LLVM
AI 编译器
AI编译器发展与架构定义，未来挑战与思考
前端优化
前端优化（算子融合、内存优化等）
后端优化
后端优化(Kernel优化、Auto Tuning)
多面体
复杂的循环依赖关系映射到高维几何空间
5. 通信架构
集合通信
通信原语、通信原理、集合通信算法
NCCL/HCCL
集合通信库、网络拓扑、通信方式、通信算法，NCCL 架构
硬件体系结构
3. AI 集群
集群管理运维
K8s集群运维、K8s容器、集群监控等工具
集群性能指标
稳定性、吞吐、线性度等
集群训推一体化
训练、推理大模型执行，训练推理显存分析
机房建设
风火水电、夜冷、柜板等知识
1. AI 芯片
AI 计算体系
AI 计算模式与计算体系架构
AI 芯片基础
CPU、GPU、NPU等芯片体基础原理
英伟达GPU
英伟达GPU TensorCore、NVLink剖析
国外AI芯片
谷歌、特斯拉等专用AI处理器核心原理
国内AI芯片
寒武纪、燧原科技等专用AI处理器原理
2. 通信与存储
通信
路由器、交换机基本原理和网络拓扑
存储
DRAM、SRAM、存储 POD 到大模型存储 CKPT 算法

思维导图 XMind


集合通信
Collective
communication
集合通信概览
通信算法
Ring、Tree、Have-Doubling
通信链路
通信操作(原语)
点对点通信、集合通信
通信域管理
集群硬件概览
芯片 - 服务器
组网 - 集群互联
业界集合通信库
MPI/OpenMPI
业界通信库
GLOO、MSCCL、HCCL，etc.
基于NCCL通信库
ACCL、TCCL, etc.
NCCL/HCCL 解读
NCCL 流程与算法
PXN 算法、DBT 算法
HCCL 整体架构与特性
集合通信总结
影响通信带宽的因素
大模型需要怎样的通信
TP/PP/EP/SP/CP/DP 多维并行

Question

1. 用英伟达跑大模型，在 PyTorch 等 AI 框架都集成了 NCCL，那么 NCCL 的 API 怎么使用?

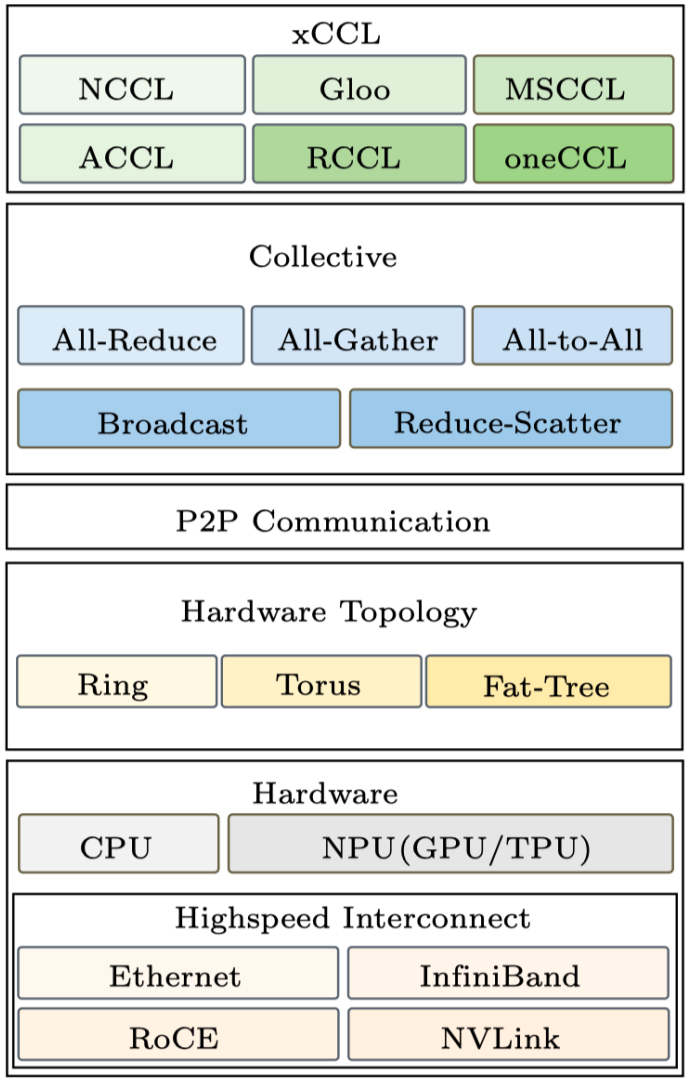
本节内容

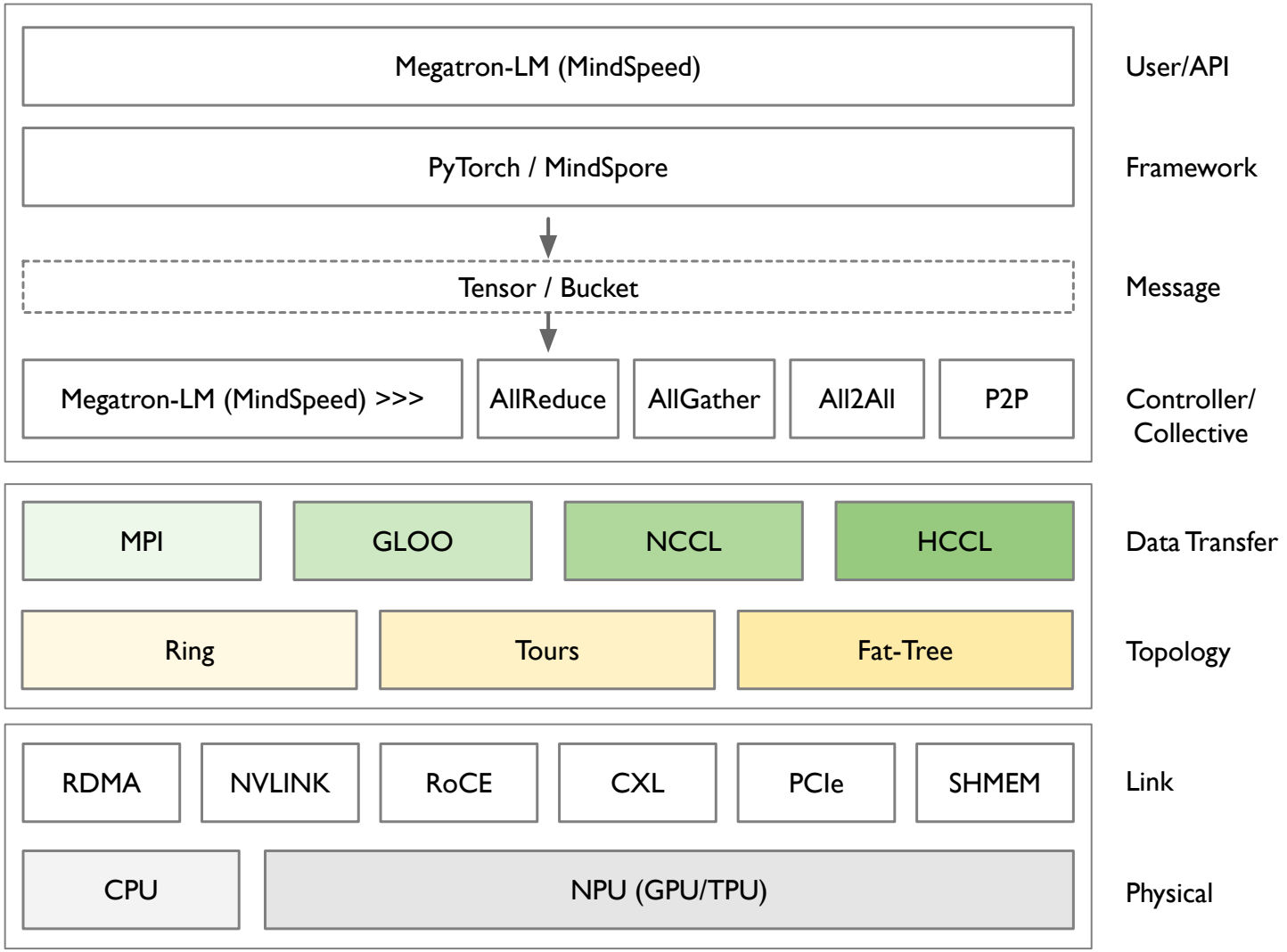
1. Communicator 生命周期
2. 错误处理和终止
3. 通信容错
4. 集合通信
5. 组间调用
6. 点对点通信

01. NCCL 基本概念

Basic Concept

XCCL 在 AI 系统中的位置





集合通信基本概念

1. **Devices and Streams 设备和 Sreams 流：**
   - NCCL 可以在多个 GPU 间进行通信。
   - 同时 NCCL 支持与 CUDA 流（Streams）集成，实现异步通信和并行计算。
2. **Synchronization 同步：**
   - 分布式计算中，同步是至关重要。
   - NCCL 提供各种同步原语，e.g. barrier 同步，以确保进程在执行集合通信时一致状态。
3. **Performance Optimization 性能优化：**
   1. 提供集体通信合并、数据传输批量处理、信道拆分、数据拆分、通信与计算并行等优化的方式。

集合通信基本概念

1. Collective Operations 集合通信操作：

- NCCL 支持多种集合通信操作，可以在多个 GPU 或节点间进行数据同步和合并。
- e.g. 广播（Broadcast）、规约（Reduction）、聚合（Aggregation）、AllReduce、AllGather 等。

2. Processes and Groups 进程和组：

- NCCL 中进程（Processes）表示参与通信的计算节点。
- 进程可以组织成组（Groups），以便在组内进行集合通信。

3. Communicators 通信者：

- 指定参与通信的进程组和具体的通信操作，其定义了一组能够互相发消息的进程。
- 进程中每个进程会被分配一个序号，称作 Rank，进程间显性地通过指定 Rank 来进行通信。

https://github.com/NVIDIA/nccl

02. Communicator 生命周期

创建单个 Communicator

```
ncclResult_t ncclCommInitAll(ncclComm_t* comm, int ndev, const int* devlist) {
  ncclUniqueId Id;
  ncclGetUniqueId(&Id);
  ncclGroupStart();
  for (int i=0; i<ndev; i++) {
    cudaSetDevice(devlist[i]);
    ncclCommInitRank(comm+i, ndev, Id, i);
  }
  ncclGroupEnd();
}
```

为每个CUDA设备指定一个unique rank

创建communicator对象，每个 comminutor 关联固定 Rank

创建单个 Communicator

```
ncclResult_t ncclCommInitAll(ncclComm_t* comm, int ndev, const int* devlist) {
  ncclUniqueId Id;
  ncclGetUniqueId(&Id);
  ncclGroupStart();
  for (int i=0; i<ndev; i++) {
    cudaSetDevice(devlist[i]);
    ncclCommInitRank(comm+i, ndev, Id, i);
  }
  ncclGroupEnd();
}
```

ncclGetUniqueId() 创建 unique ID，通过广播给所有相关线程和进程

Unique ID 被所有进程和线程共享，让他们进行同步

创建多个 Communicator

- ncclCommSplit 对已有 Communicator进行划分，将其划分成多个sub-partitions；

```
int rank;
ncclCommUserRank(comm, &rank);
ncclCommSplit(comm, 0, rank, &newcomm, NULL);
```

- 或者复制一个现有的Communicator，甚至是创建一个拥有更少的ranks的单个communitor。

```
int rank, nranks;
ncclCommUserRank(comm, &rank);
ncclCommCount(comm, &nranks);
ncclCommSplit(comm, rank/(nranks/2), rank%(nranks/2), &newcomm, NULL);
```

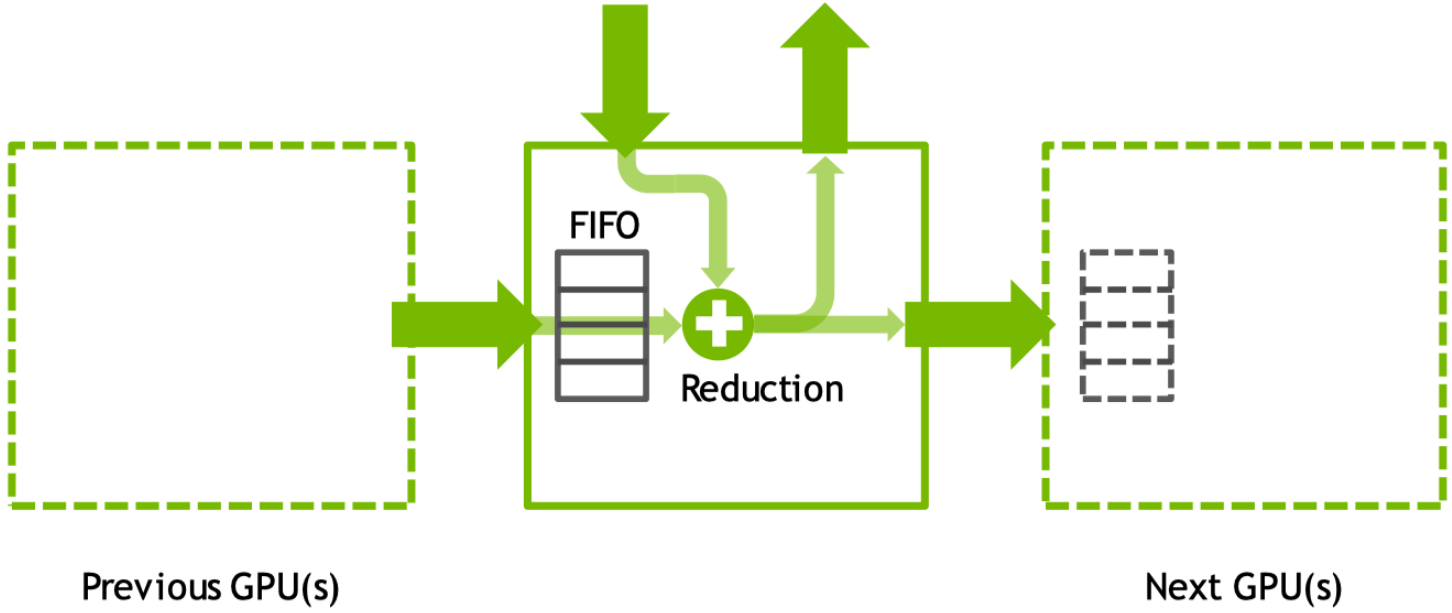
创建多个 Communicator

- 函数需要原始 communicator 中所有 ranks 一起调用，即便是某些 rank 将不再是这个 communicator 一部分，也需要将 color 参数设置为 NCCL_SPLIT_NOCOLOR 来调用。

```
int rank;
ncclCommUserRank(comm, &rank);
ncclCommSplit(comm, rank<2 ? 0 : NCCL_SPLIT_NOCOLOR, rank, &newcomm, NULL);
```

使用多个 Communicator

- 使用多个NCCL Communicator需要小心的进行同步，否则会造成死锁。NCCL kernel会因为等待数据到来而阻塞，在此期间任何CUDA操作都会导致设备同步，意味着需要等待所有的NCCL kernel完成。
- 即NCCL kernel在等待数据到来期间如果有任何CUDA operation进入了队列就会导致死锁，因为NCCL也会执行CUDA调用，而NCCL的CUDA调用会进入队列中等待前一个CUDA操作执行完毕。



Finalizing a communicator

- ncclCommFinalize 会把一个communicator从ncclSuccess状态转变为ncclInProgress状态，开始完成background中的各种操作并与其他ranks之间进行同步。
- ncclCommFinalize会将所有未完成的操作以及与该communicaotr所属的网络相关的资源会被flushed并被释放掉。一旦所有的NCCL操作都完成了，communicator会将状态转换成ncclSuccess。可以通过ncclCommGetAsyncError来查询状态。

ncclCommFinalize

```
ncclResult_t ncclCommFinalize(ncclComm_t comm)
```

Finalize a communicator object *comm*. When the communicator is marked as nonblocking, *ncclCommFinalize* is a nonblocking function. Successful return from it will set communicator state as *ncclInProgress* and indicates the communicator is under finalization where all uncompleted operations and the network-related resources are being flushed and freed. Once all NCCL operations are complete, the communicator will transition to the *ncclSuccess* state. Users can query that state with *ncclCommGetAsyncError*.

Destroying a communicator

- 一个Communicator被Finalize之后，下一步就是释放掉它所拥有的全部资源，包括它自己也会被释放。ncclCommDestroy函数会释放掉一个Communicator的本地所属资源。
- 调用 ncclCommDestroy 时候如果对应的Communicator的状态是ncclSuccess，那么可以保证这个调用是非阻塞，否则这个调用可能会被阻塞。会释放掉一个Communicator的所有资源然后返回，对应的Communicator也不能再被使用。

ncclCommDestroy

```
ncclResult_t ncclCommDestroy(ncclComm_t comm)
```

Destroy a communicator object *comm*. *ncclCommDestroy* only frees the local resources that are allocated to the communicator object *comm* if *ncclCommFinalize* was previously called on the communicator; otherwise, *ncclCommDestroy* will call ncclCommFinalize internally. If *ncclCommFinalize* is called by users, users should guarantee that the state of the communicator becomes *ncclSuccess* before calling *ncclCommDestroy*. In all cases, the communicator should no longer be accessed after ncclCommDestroy returns. It is recommended that users call *ncclCommFinalize* and then *ncclCommDestroy*. This function is an intra-node collective call, which all ranks on the same node should call to avoid a hang.

03. Error handling

错误处理

异步错误及其处理

```
int ncclStreamSynchronize(cudaStream_t stream, ncclComm_t comm) {
  cudaError_t cudaErr;
  ncclResult_t ncclErr, ncclAsyncErr;
  while (1) {
   cudaErr = cudaStreamQuery(stream);
   if (cudaErr == cudaSuccess)
     return 0;

   if (cudaErr != cudaErrorNotReady) {
     printf("CUDA Error : cudaStreamQuery returned %d\n", cudaErr);
     return 1;
   }

   ncclErr = ncclCommGetAsyncError(comm, &ncclAsyncErr);
   if (ncclErr != ncclSuccess) {
     printf("NCCL Error : ncclCommGetAsyncError returned %d\n", ncclErr);
     return 1;
   }

   if (ncclAsyncErr != ncclSuccess) {
     // An asynchronous error happened. Stop the operation and destroy
     // the communicator
     ncclErr = ncclCommAbort(comm);
     if (ncclErr != ncclSuccess)
       printf("NCCL Error : ncclCommDestroy returned %d\n", ncclErr);
     // Caller may abort or try to create a new communicator.
     return 2;
   }

   // We might want to let other threads (including NCCL threads) use the CPU.
   sched_yield();
  }
}
```

- 所有的NCCL调用都会返回一个NCCL错误码（error code）。
- 如果某个NCCL调用返回的error code不是ncclSuccess或者ncclInternalError，并且NCCL_DEBUG设置为WARN，NCCL会打印出human-readable的信息来解释内部发生了什么导致该错误的发生。
- 如果NCCL_DEBUG设置为INFO，NCCL也会打印出导致该错误的调用栈，以便帮助用户定位和修复问题。

异步错误及其处理

```
int ncclStreamSynchronize(cudaStream_t stream, ncclComm_t comm) {
  cudaError_t cudaErr;
  ncclResult_t ncclErr, ncclAsyncErr;
  while (1) {
   cudaErr = cudaStreamQuery(stream);
   if (cudaErr == cudaSuccess)
     return 0;

   if (cudaErr != cudaErrorNotReady) {
     printf("CUDA Error : cudaStreamQuery returned %d\n", cudaErr);
     return 1;
   }

   ncclErr = ncclCommGetAsyncError(comm, &ncclAsyncErr);
   if (ncclErr != ncclSuccess) {
     printf("NCCL Error : ncclCommGetAsyncError returned %d\n", ncclErr);
     return 1;
   }

   if (ncclAsyncErr != ncclSuccess) {
     // An asynchronous error happened. Stop the operation and destroy
     // the communicator
     ncclErr = ncclCommAbort(comm);
     if (ncclErr != ncclSuccess)
       printf("NCCL Error : ncclCommDestroy returned %d\n", ncclErr);
     // Caller may abort or try to create a new communicator.
     return 2;
   }

   // We might want to let other threads (including NCCL threads) use the CPU.
   sched_yield();
  }
}
```

- 一些communicator 错误，尤其是网络错误，会通过ncclCommGetAsyncError函数报告。
- 发生异步错误的操作通常将难以继续进行下去并且该操作将永远不能完成。因此当异步错误发生的时候，对应的操作应该被丢弃且通信器应该被destroy，可以通过ncclCommAbort来进行上述操作。
- 如果是等待NCCL操作完成的时候发生了异步错误，我们的应用中应该调用ncclCommGetAsyncError函数来destroy对应的communicator。

04.Fault Tolerance

容错

```
bool globalFlag;
bool abortFlag = false;
ncclConfig_t config = NCCL_CONFIG_INITIALIZER;
config.blocking = 0;
CHECK(ncclCommInitRankConfig(&comm, nRanks, id, myRank, &config));
do {
  CHECK(ncclCommGetAsyncError(comm, &state));
} while(state == ncclInProgress && checkTimeout() != true);

if (checkTimeout() == true || state != ncclSuccess) abortFlag = true;

/* sync abortFlag among all healthy ranks. */
reportErrorGlobally(abortFlag, &globalFlag);

if (globalFlag) {
  /* time is out or initialization failed: every rank needs to abort and restart. */
  ncclCommAbort(comm);
  /* restart NCCL; this is a user implemented function, it might include
   * resource cleanup and ncclCommInitRankConfig() to create new communicators. */
  restartNCCL(&comm);
}

/* nonblocking communicator split. */
CHECK(ncclCommSplit(comm, color, key, &childComm, &config));
do {
  CHECK(ncclCommGetAsyncError(comm, &state));
} while(state == ncclInProgress && checkTimeout() != true);

if (checkTimeout() == true || state != ncclSuccess) abortFlag = true;

/* sync abortFlag among all healthy ranks. */
reportErrorGlobally(abortFlag, &globalFlag);

if (globalFlag) {
  ncclCommAbort(comm);
  /* if chilComm is not NCCL_COMM_NULL, user should abort child communicator
   * here as well for resource reclamation. */
  if (childComm != NCCL_COMM_NULL) ncclCommAbort(childComm);
  restartNCCL(&comm);
}
/* application workload */
```

- NCCL提供了许多feature来让我们的应用从严重的错误中恢复到正常，比如网络连接失败、节点宕机以及进程挂掉等。当这样的错误发生的时候，应用应该能够调用 ncclCommAbort 方法来释放相应的 communicator 的资源，然后创建一个新的 communicator 继续之前的任务。为了保证 ncclCommAbout 能够在任何时间点被调用，所有的NCCL调用都可以是非阻塞（异步）的操作。

- 为了正确的弃用，当一个communicator中任何一个rank出错了的时候，所有其他的rank都需要调用 ncclCommAbort 来启用它们自己的 NCCL communicator。用户可以实现方法来决定什么时候以及是否弃用这些 communicator 并重新开始当前的NCCL操作。

05. 集合通信

Collective Operations

集合操作

- 集体操作（Collective Operation）需要被每个 rank 都调用，从而形成一次集体的操作。如果失败的话可能会导致其他rank陷入无尽的等待。

06. 组调用

Group Calls

Group functions

- 组函数可以将多个调用合并成一个。有三种用法，它们也可以组合起来：
  - 使用一个线程管理多个GPU
  - 聚合通信算子，从而提升性能
  - 合并多个send/receive类型的点对点操作。

07. 点对点通信

Point-to-point

Thank you

把AI系统带入每个开发者、每个家庭、每个组织，构建万物互联的智能世界

Bring AI System to every person, home and organization for a fully connected, intelligent world.

ZOMI

Course chenzomi12.github.io

GitHub github.com/chenzomi12/AIFoundation