大模型系列 - 集合通信

ZOMI

PyTorch 集合通信&计算并行

大模型业务全流程

数据 & 模型算法

2. 数据处理

- 开源数据
- 数据预处理
- 向量数据库

3. 模型算法

- LLM模型架构
- 多模态一切皆Tokens

模型训练 & 微调

4. 模型训练

- 混合精度
- 梯度检查
- 梯度累积
- …

5. 分布式并行

训练集群稳定性

6. 模型微调

- 全参微调
- 低参微调
- 指令微调

模型验证 & 推理部署

7. 模型验证

- 下游任务
- 测评标准
- bench mark

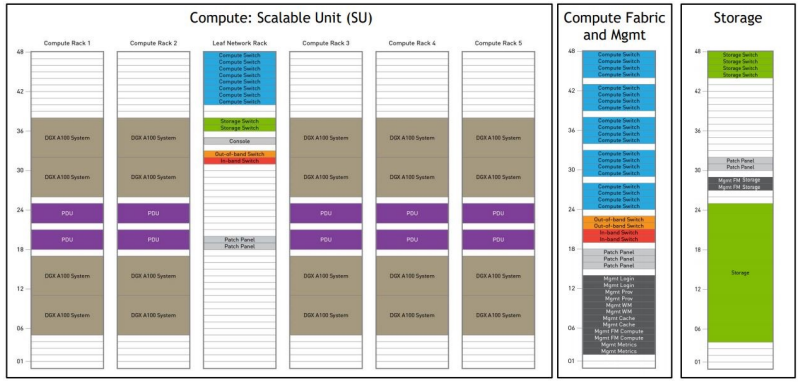
8. 推理与智能体

- 量化压缩
- 推理加速
- 9. Agent 智能体

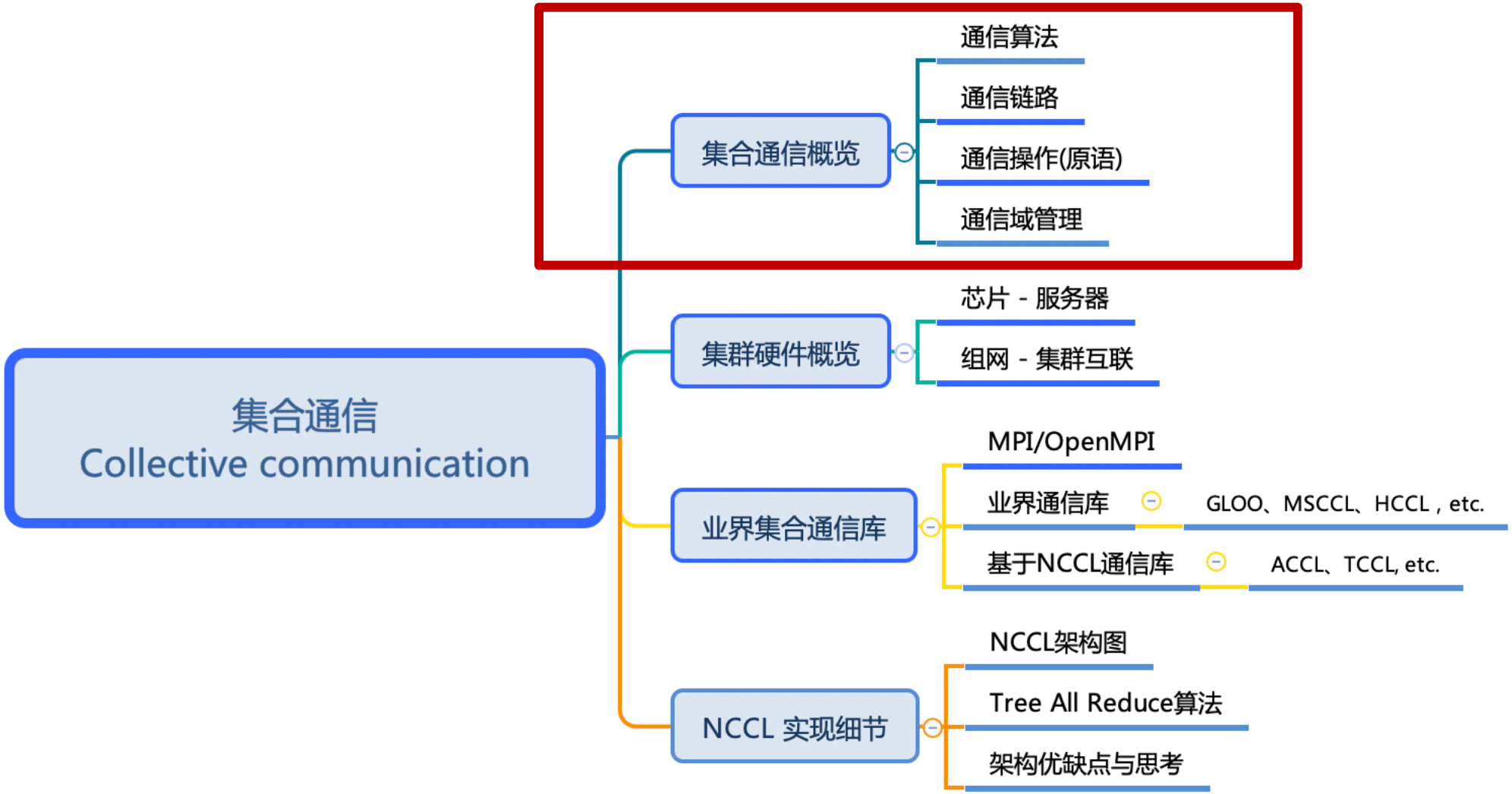
集群算力准备

1. AI 集群建设

- 计算、存储、网络
- AI 集群机房建设
- AI 集群上线与运维



思维导图 XMind


集合通信
Collective communication
集合通信概览
通信算法
通信链路
通信操作(原语)
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 实现细节
NCCL架构图
Tree All Reduce算法
架构优缺点与思考

集合通信概览

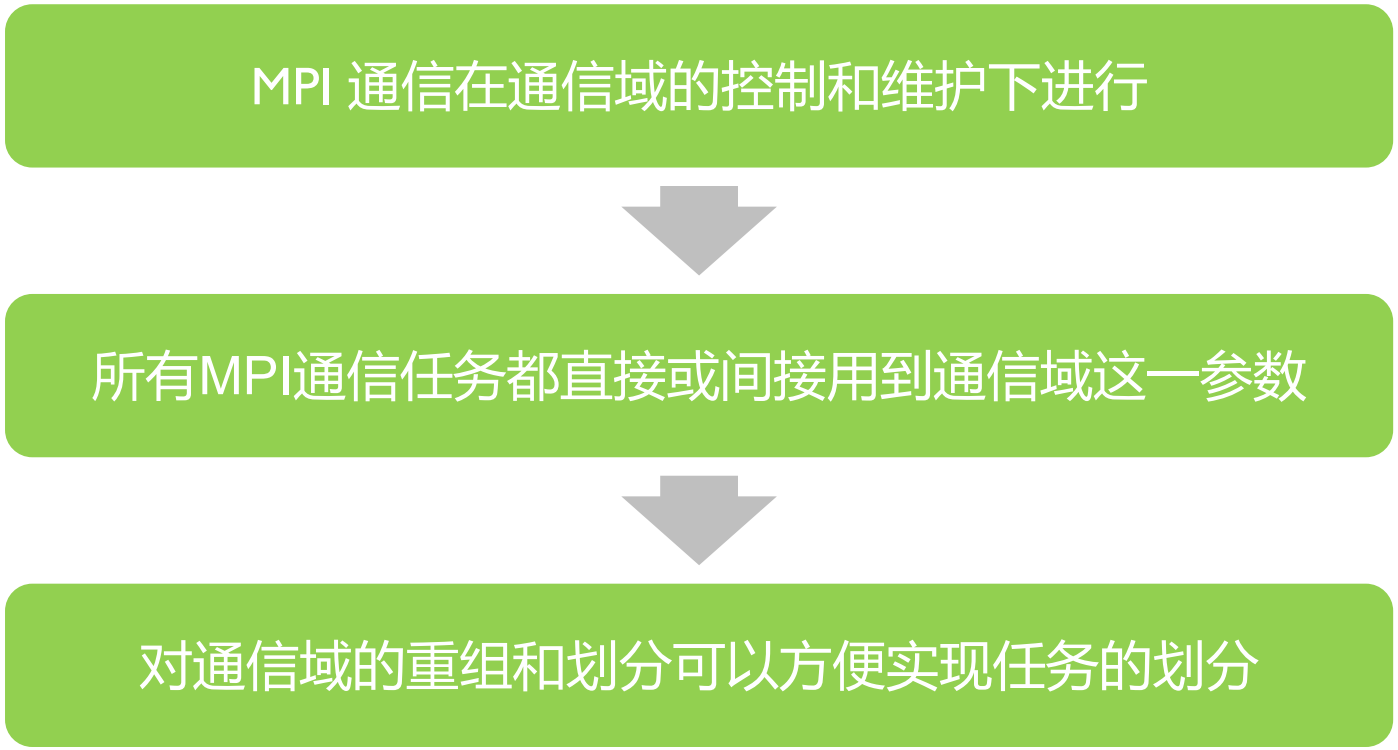
XCCL: XXXX Collective Communication Library

1. AI 与通信关系（AI 基础知识、训练推理、分布式并行）
2. XCCL 基本架构（HPC 通信架构 to XCCL 通信架构）
3. 集合通信原语（All Reduce, etc.）
4. 集合网络拓扑（Hypercube 、Ring 、Torus 、Fat-Tree 、Dragonfly & Dragonfly+）
5. PyTorch 集合通信与计算并行
   - 通信域、进程、进程组与 Rank 间关系
   - PyTorch 使用分布式功能调用集合通信
   - PyTorch 执行时计算与通信并行底层原理

01. 通信域与 Rank

什么是通信域

- 通信域是 MPI 重要概念：



什么是通信域

- 通信域是 MPI 重要概念：
  - 通信域（communicator）是一个综合的通信概念。其包括上下文（context），进程组（group），虚拟处理器拓扑（topology）。**其中进程组是比较重要的概念，表示通信域中所有进程的集合。一个通信域对应一个进程组。**

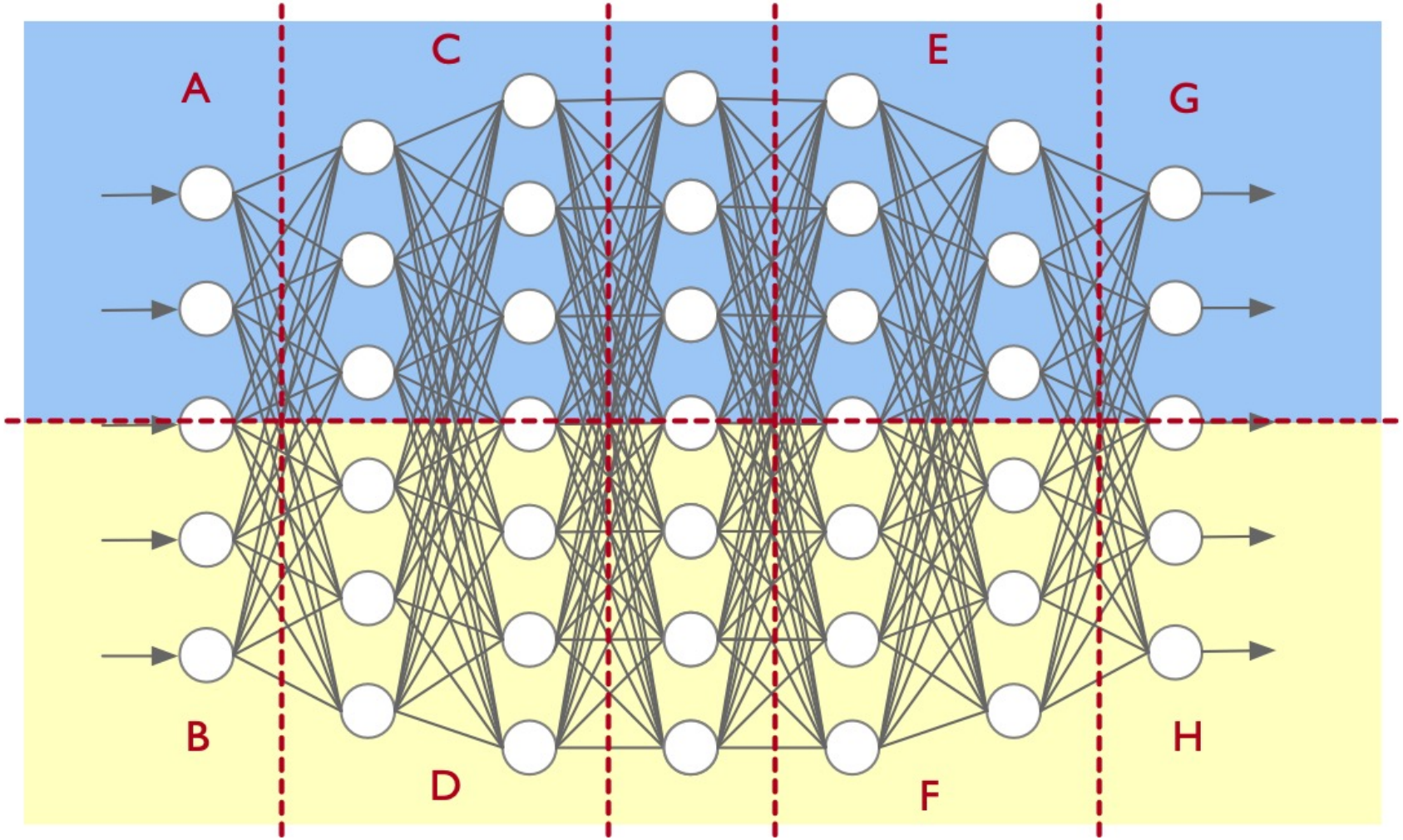
什么是进程与进程组

- 进程（process）与进程组（group）关系：
  - 每个进程客观上唯一的（一个进程对应一个 Process ID）；
  - 同一个进程可以属于多个进程组（每个进程在不同进程组中有个各自 Rank ID）；
  - 同一个进程可以属于不同的进程组（PID），因此也可以属于不同的通信域。

通信域与进程组之间的关系

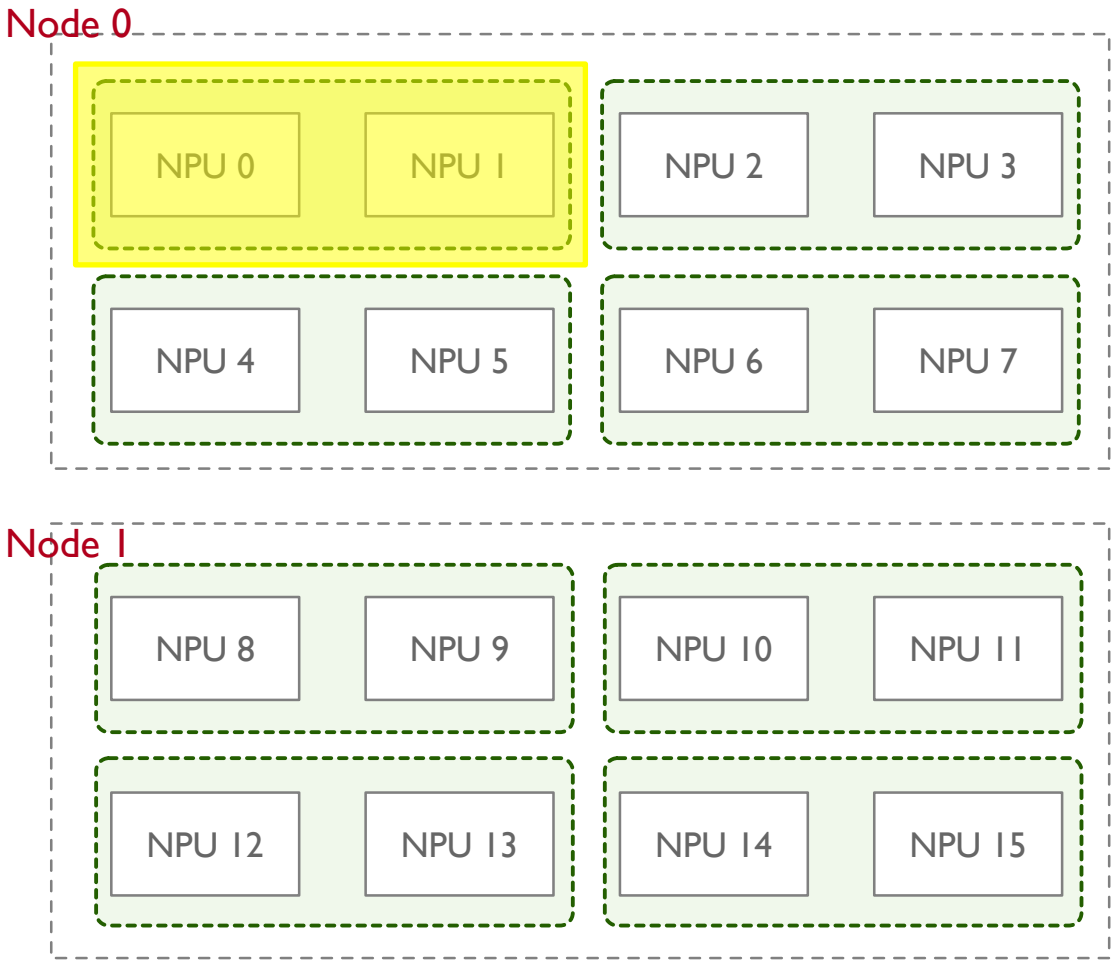
- 同一个进程，可以属于不同通信域；
- 同一个进程，可以同时参与不同通信域的通信，互不干扰。

PTD 并行在集群里面与模型的关系

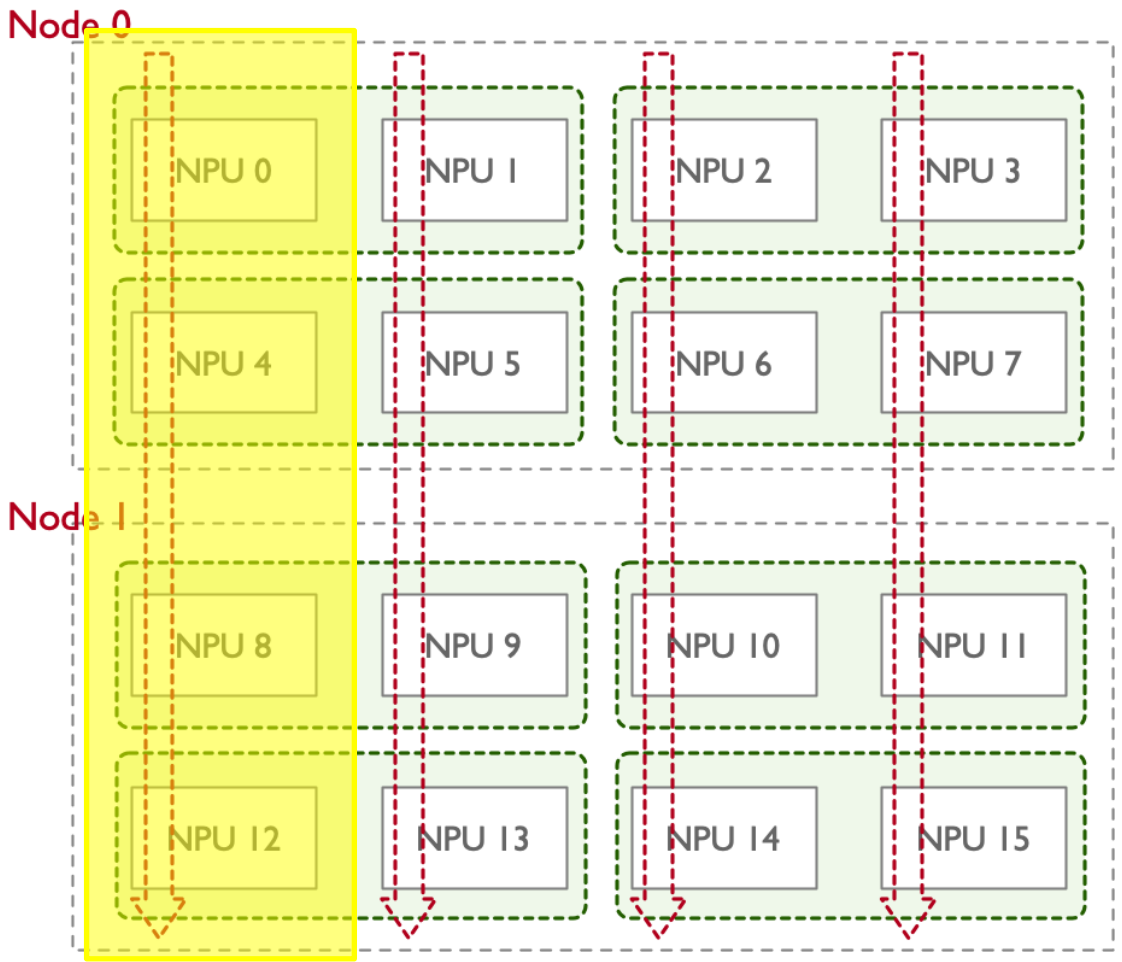


TP && PP 通信域

- Tensor Parallel

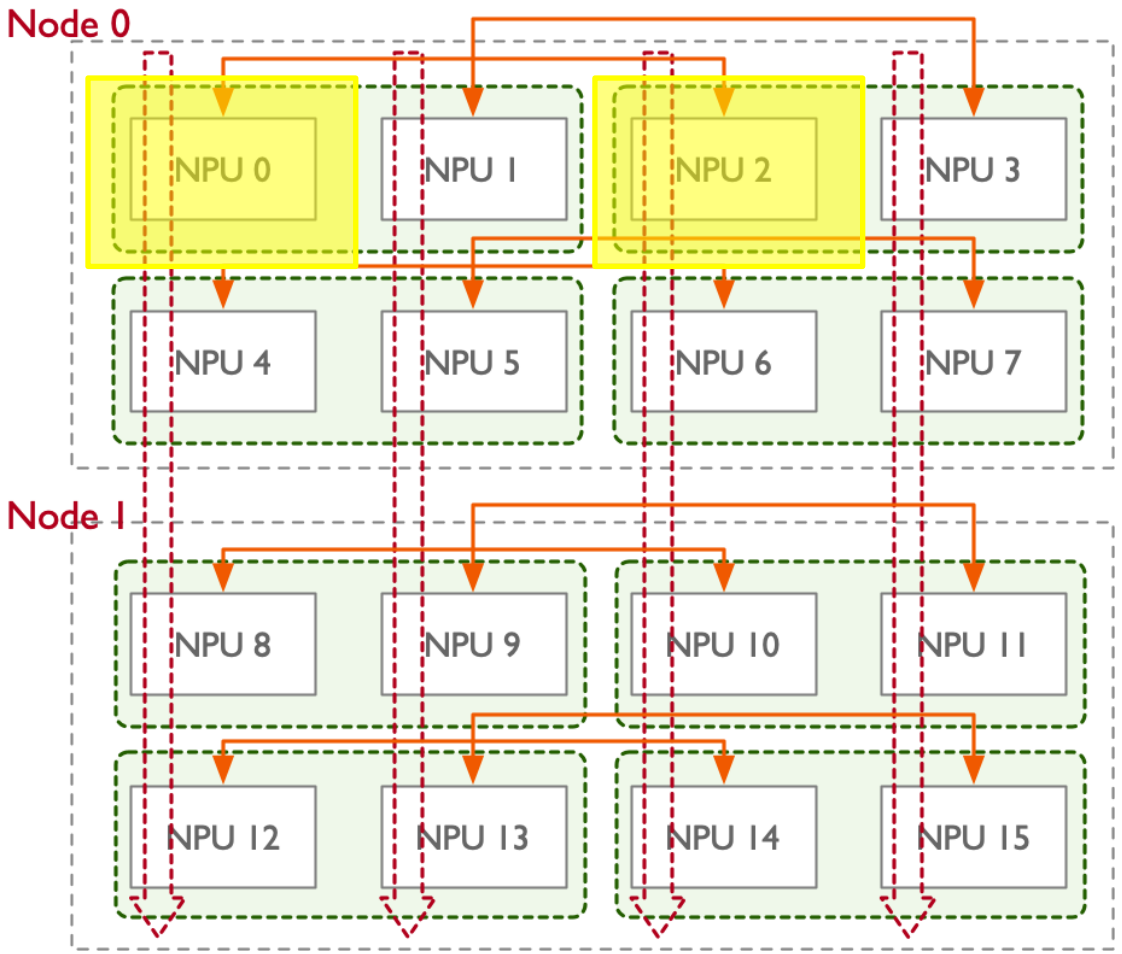


- Pipeline Parallel

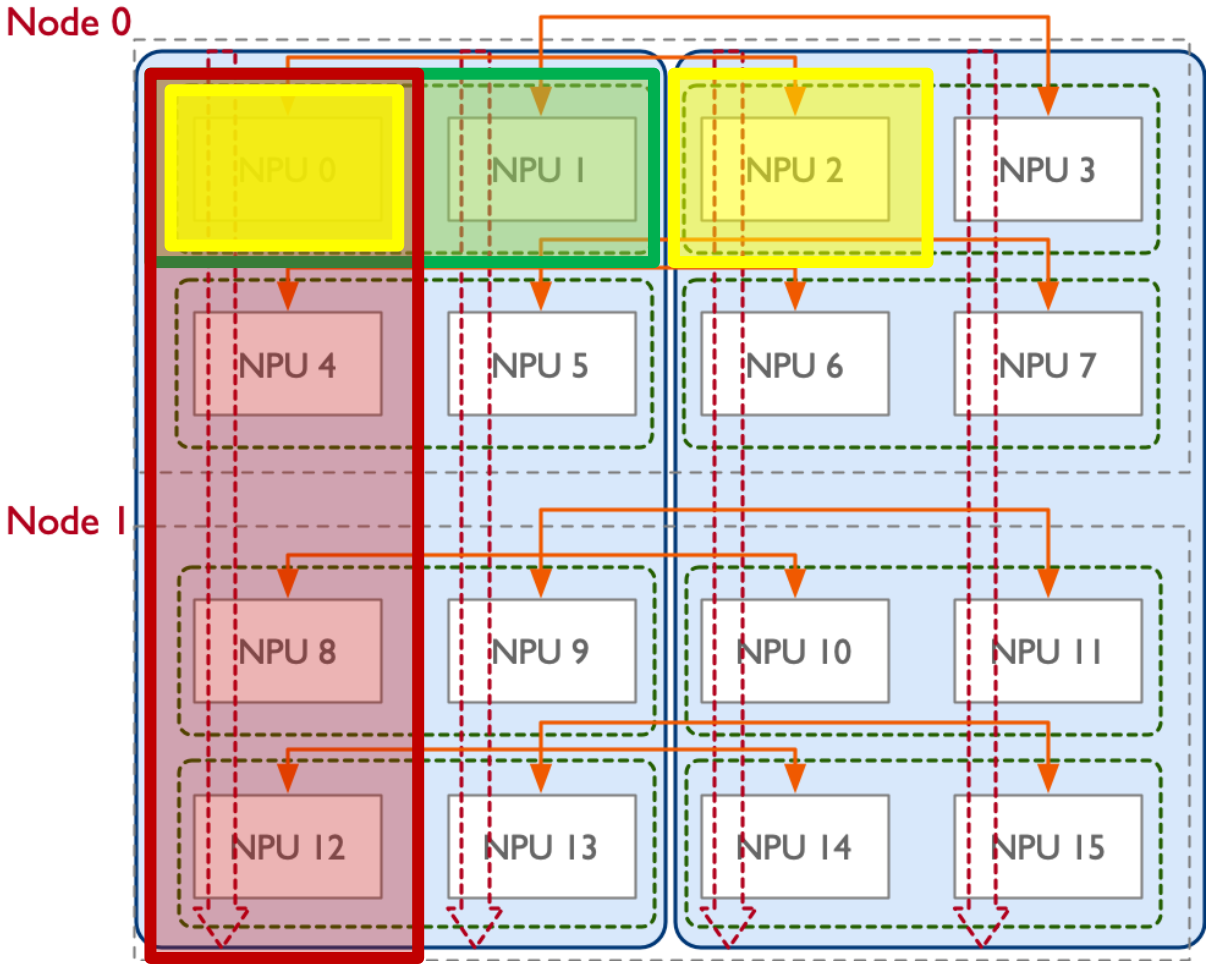


DP && MP 通信域

- Data Parallel



- Hybrid Parallel



02. PyTorch

通信调用

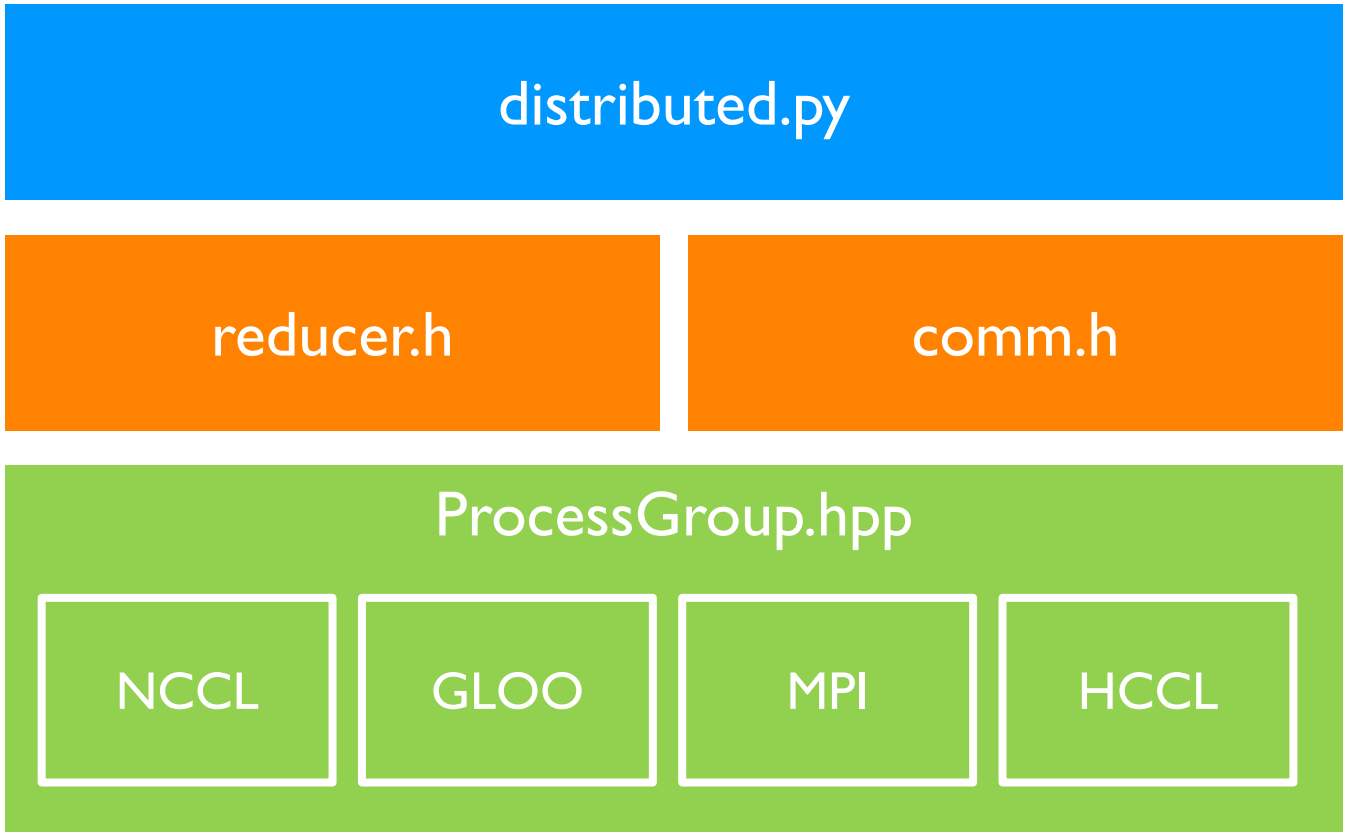
PyTorch 分布式训练依赖

- PyTorch 分布式训练通信依赖 torch.distributed 模块实现，其提供 P2P 和 CC 两种通信方式：
  - Point-2-Point Communication：提供 send 和 recv 语义，用于任务间通信；
  - Collective Communication：提供 scatter/broadcast/gather/reduce/all reduce/all gather 通信操作；

- 不同 backend 在提供的通信语义上具有一定的差异性

PyTorch 分布式训练依赖

- PyTorch 用户直接感知的是 distributed.py 中定义的数据类型 DDP :class:torch.nn.parallel.DistributedDataParallel`。 而 NCCL/MPI/Gloo 等分布式通信库直接对接的 ProcessGroup 结构。



后端通信库支持度

	GLOO		MPI		NCCL		HCCL	
Device	CPU	GPU	CPU	GPU	CPU	GPU	CPU	NPU
send	✓	✗	✓	?	✗	✓	✗	✓
recv	✓	✗	✓	?	✗	✓	✗	✓
broadcast	✓	✓	✓	?	✗	✓	✗	✓
all_reduce	✓	✓	✓	?	✗	✓	✗	✓
reduce	✓	✗	✓	?	✗	✓	✗	✓
all_gather	✓	✗	✓	?	✗	✓	✗	✓
gather	✓	✗	✓	?	✗	✓	✗	✓
scatter	✓	✗	✓	?	✗	✓	✗	✓
reduce_scatter	✗	✗	✗	✗	✗	✓	✗	✓
all_to_all	✗	✗	✓	?	✗	✓	✗	✓
barrier	✓	✗	✓	?	✗	✓	✗	✓

P2P Communication 操作 Step1

- **Step1 初始化：**
  - PyTorch 分布式通信操作（原语）使用前，需要对分布式模块进行初始化
  - PyTorch 分布式模块通过 torch.distributed.init_process_group 来完成

```
3
4   def init_process(rank_id, size, fn, backend='gloo'):
5       """ Initialize the distributed environment. """
6       os.environ['MASTER_ADDR'] = '127.0.0.1'
7       os.environ['MASTER_PORT'] = '29500'
8       dist.init_process_group(backend, rank=rank_id, world_size=size)
9       fn(rank_id, size)
10
```

P2P Communication 操作 Step2

- **Step2 通信逻辑：**
  - 通过 rank_id 来区分当前应该执行哪一个 rank 业务逻辑；
  - PyTorch 通过 torch.distributed.send() 来实现 tensor 发送，其中 send 是同步函数 isend 是异步函数；
  - PyTorch 中通过 torch.distributed.recv() 来实现 tensor 接收，其中 recv 是同步函数 irecv 是异步函数；

P2P Communication 操作 Step2

```
13  def run(rank_id, size):
14      tensor = torch.zeros(1)
15      if rank_id == 0:
16          tensor += 1
17          dist.send(tensor=tensor, dst=1)
18          print('after send, Rank ', rank_id, ' has data ', tensor[0])
19          dist.recv(tensor=tensor, src=1)
20          print('after recv, Rank ', rank_id, ' has data ', tensor[0])
21      else:
22          dist.recv(tensor=tensor, src=0)
23          print('after recv, Rank ', rank_id, ' has data ', tensor[0])
24          tensor += 1
25          dist.send(tensor=tensor, dst=0)
26          print('after send, Rank ', rank_id, ' has data ', tensor[0])
27
```

P2P Communication 操作 Step3

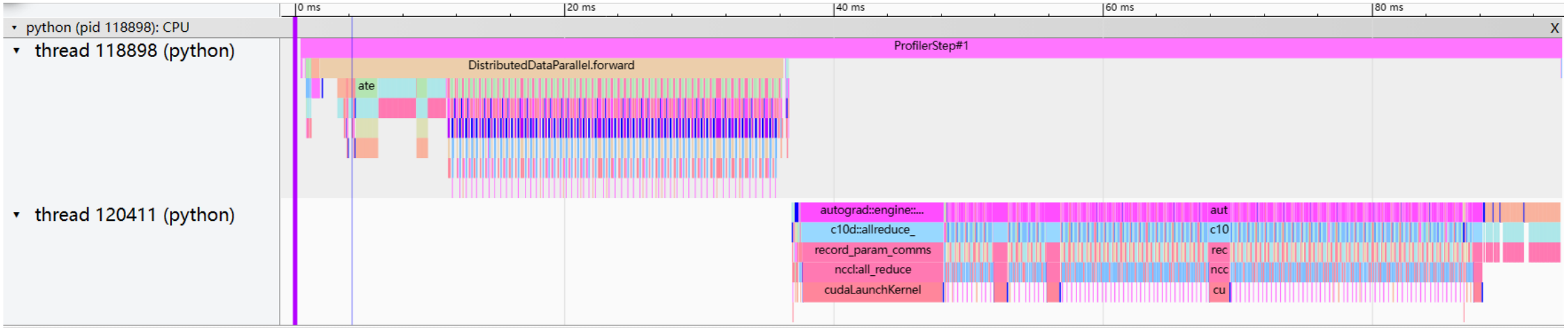
- **Step3 任务启动：**
  - 使用 torch.multiprocessing 来启动多进程，其是对 python 库中 multiprocessing 封装；
  - multiprocessing.set_start_method 用于创建 child process 方式，可选值为 fork、spawn 和 forkserver。
  - 使用 spawn，child process 仅会继承 parent process 的必要 resource，file descriptor 和 handle 均不会继承。

P2P Communication 操作 Step3

```
31  if __name__ == "__main__":
32      size = 2
33      processes = []
34      multiprocessing.set_start_method("spawn")
35      for rank in range(size):
36          p = mp.Process(target=init_process, args=(rank, size, run))
37          p.start()
38          processes.append(p)
39
40      for p in processes:
41          p.join()
```

03. PyTorch 计算与通信并行

Stream & Event 基本概念



7832 items selected. Slices (7831) | Instant Event (1)

Name	Wall Duration	Self time	Average Wall Duration	Occurrences	Event(s)	Link
ProfilerStep#1	96.521 ms	58.020 ms	96.521 ms	1	Outgoing flow	ac2g
DistributedDataParallel.forward	34.452 ms	7.193 ms	34.452 ms	1	Outgoing flow	ac2g
c10d::allreduce_	14.961 ms	0.048 ms	2.992 ms	5	Outgoing flow	ac2g
nccl:all_reduce	14.767 ms	0.360 ms	2.953 ms	5	Outgoing flow	ac2g
record_param_comms	15.205 ms	0.202 ms	1.689 ms	9	Outgoing flow	ac2g
aten::unflatten_dense_tensors	2.455 ms	0.674 ms	1.228 ms	2	Outgoing flow	ac2g
Optimizer.zero_grad#SGD.zero_grad	0.592 ms	0.592 ms	0.592 ms	1	Outgoing flow	ac2g
aten::flatten_dense_tensors	0.964 ms	0.388 ms	0.482 ms	2	Outgoing flow	ac2g
autograd::engine::evaluate_function: AddmmBackward0	0.282 ms	0.029 ms	0.282 ms	1	Outgoing flow	ac2g
autograd::engine::evaluate_function: ConvolutionBackward0	13.116 ms	0.830 ms	0.247 ms	53	Outgoing flow	ac2g
ConvolutionBackward0	11.755 ms	0.383 ms	0.222 ms	53	Outgoing flow	ac2g

Stream 基本概念

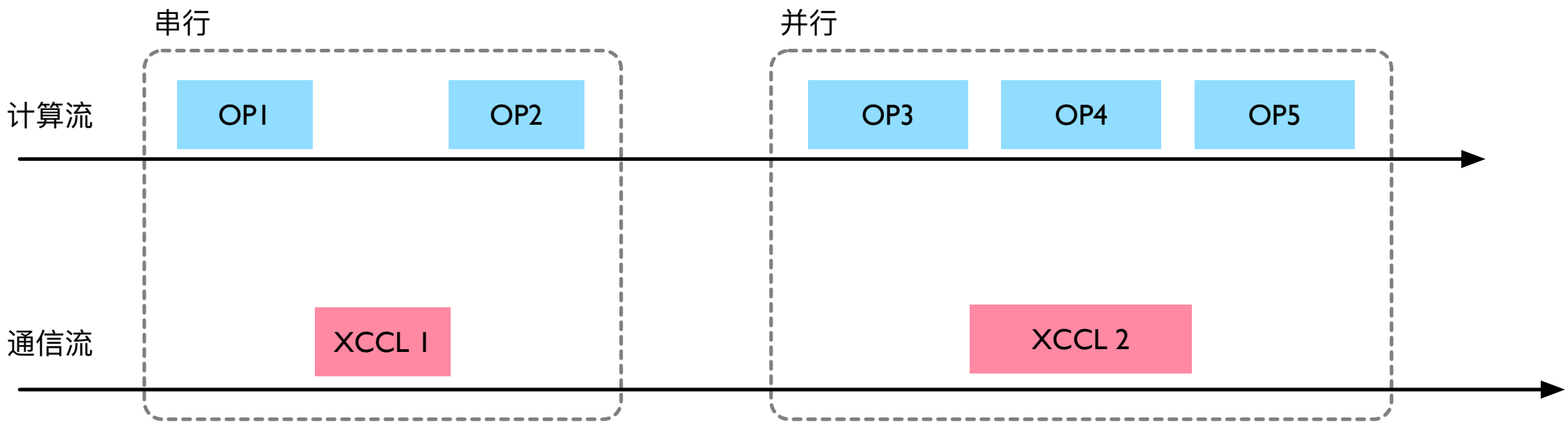
- 使用 CANN/CUDA 的程序一般需要处理海量数据，内存带宽经常会成为主要瓶颈。
- 通过 Stream CANN/CUDA 的程序可以有效地将内存读取和数值运算并行，提升数据吞吐量。
- CANN/CUDA Stream 是指一堆异步 Kernel 操作，其按照 host 代码调用顺序执行在 device 上。

Event 基本概念

- Event 是 CUDA 编程中用于时间测量和 Stream 同步一种机制。
  - 允许 CUDA Stream 中创建标记点，记录特定操作 Start and End time，从而测量这些 Kernel 执行时间;
  - Event 可以被视为一种轻量级同步机制，不影响 GPU 执行速度，其本身不执行计算只是用来记录时间。

Stream & Event 通信计算并行

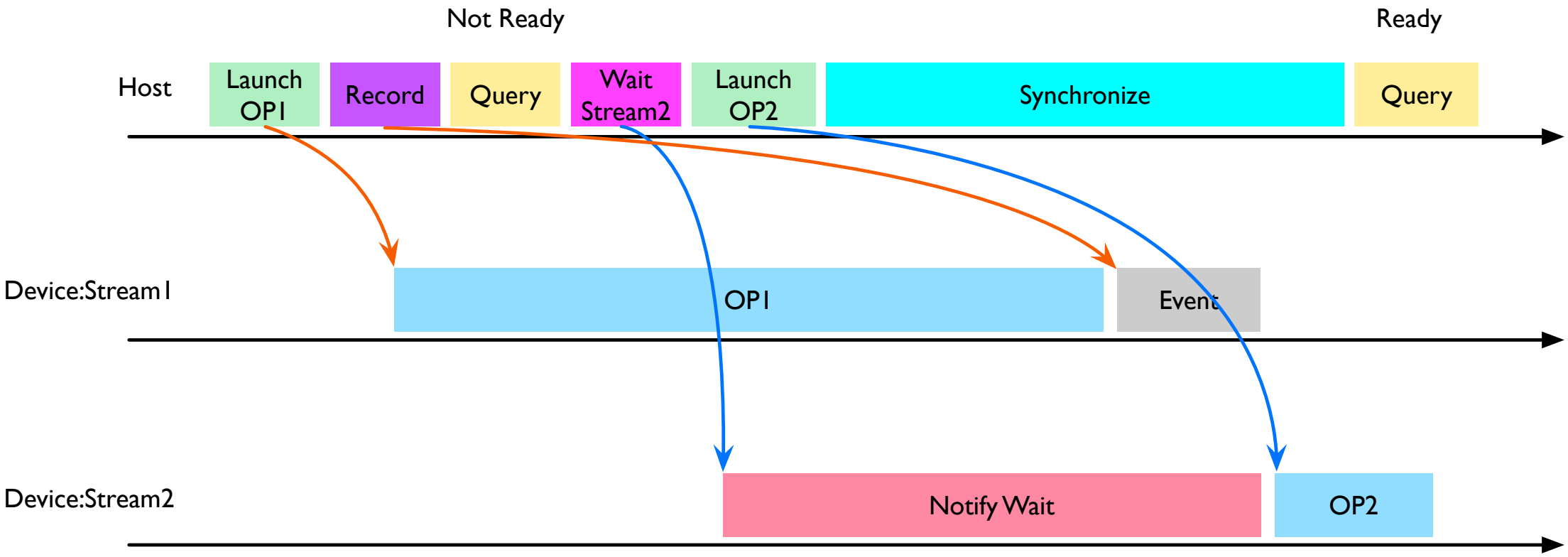
- PyTorch 通信与计算并行，主要通过 Stream（并行能力）与Event（时序控制）这两个提供的底层能力来实现：



Stream & Event 通信计算并行

- 分为 host 下发与 device 执行，二者间异步：
  - 第一个 query 结果 not ready，因为 OP1 还未执行完毕，event 在等待
  - OP2 下发 Stream2，由于 OP2 前下发了 wait stream2
  - 那么 wait Stream2 后下发任务都必须要等 event 完成后才执行
  - synchronize() 是同步接口，所以 host 阻塞直到 event 完成后才返回
  - 之后再调用 query，返回的则是 ready

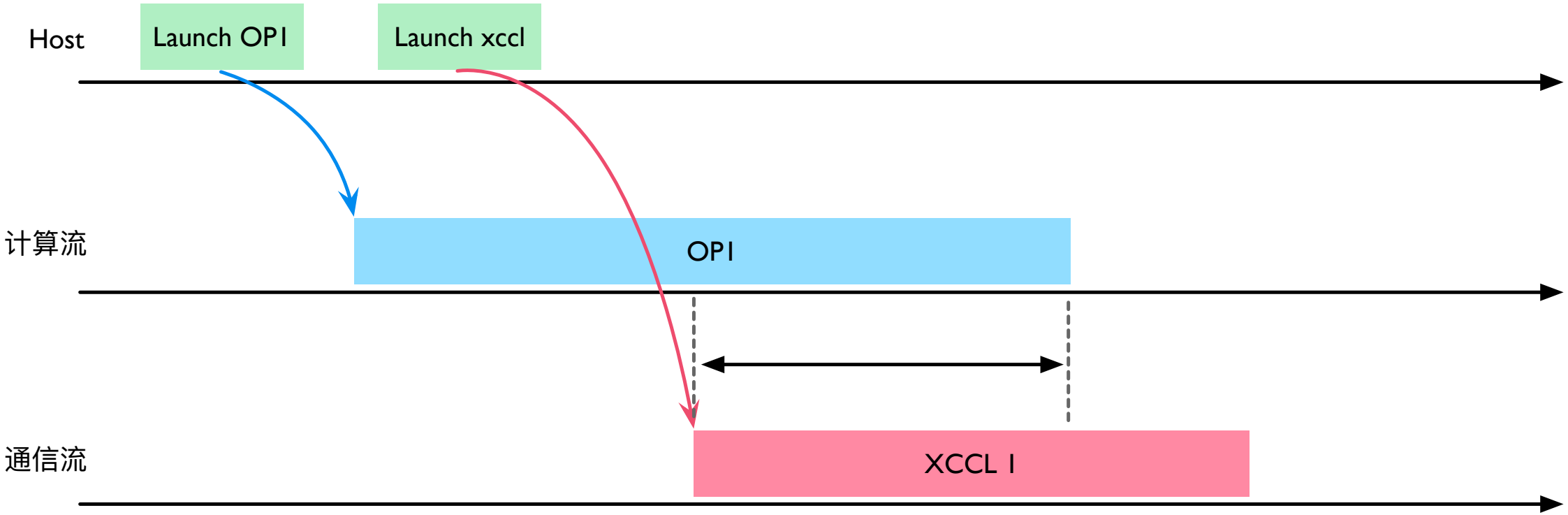
Stream & Event 通信计算并行



Stream & Event 通信计算并行

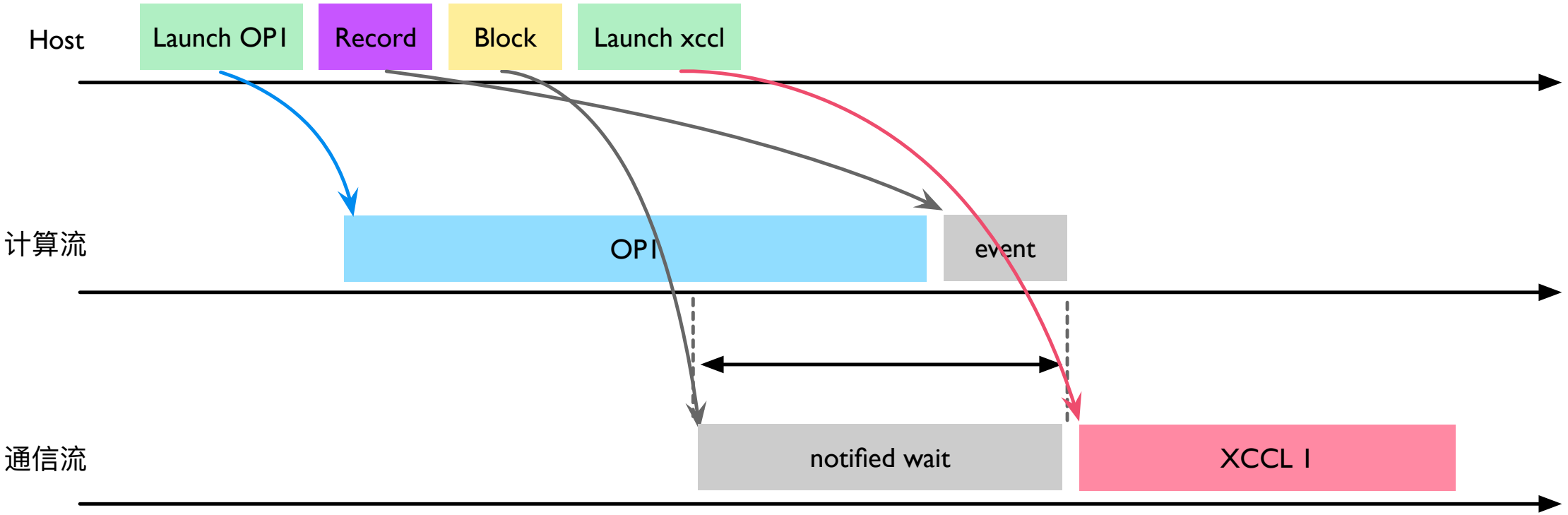
- ProcessGroupXCCL 中集合通信接口会调用 ProcessGroupXCCL::collective 接口，该接口接收一个函数 FN：XCCL 集合通信接口调用函数。FN 下发到集合通信流（xcclStreams） 上集合通信操作。
- 设 Inputs/outputs 都是默认计算流上 Tensor，那么：OP1 输出作为 HCCL1 输入 Tensor，此时该 Tensor 内存归属与计算流上内存池，PyTorch下发是异步，会出现 OP1 写内存，而 HCCL1 在读内存。
- 需要进行集合通信的 Tensor 其内存，是在 Stream 上内存池上管理（PyTorch 内存池与 Stream 绑定）。

Stream & Event 通信计算并行



Stream & Event 通信计算并行

- collective() 通过 syncStream() 函数解决异步问题：计算流下发 event，在通信流下发一个 notify wait 等待 OP1 完成。Tensor 先写后读，消除并发读写问题。



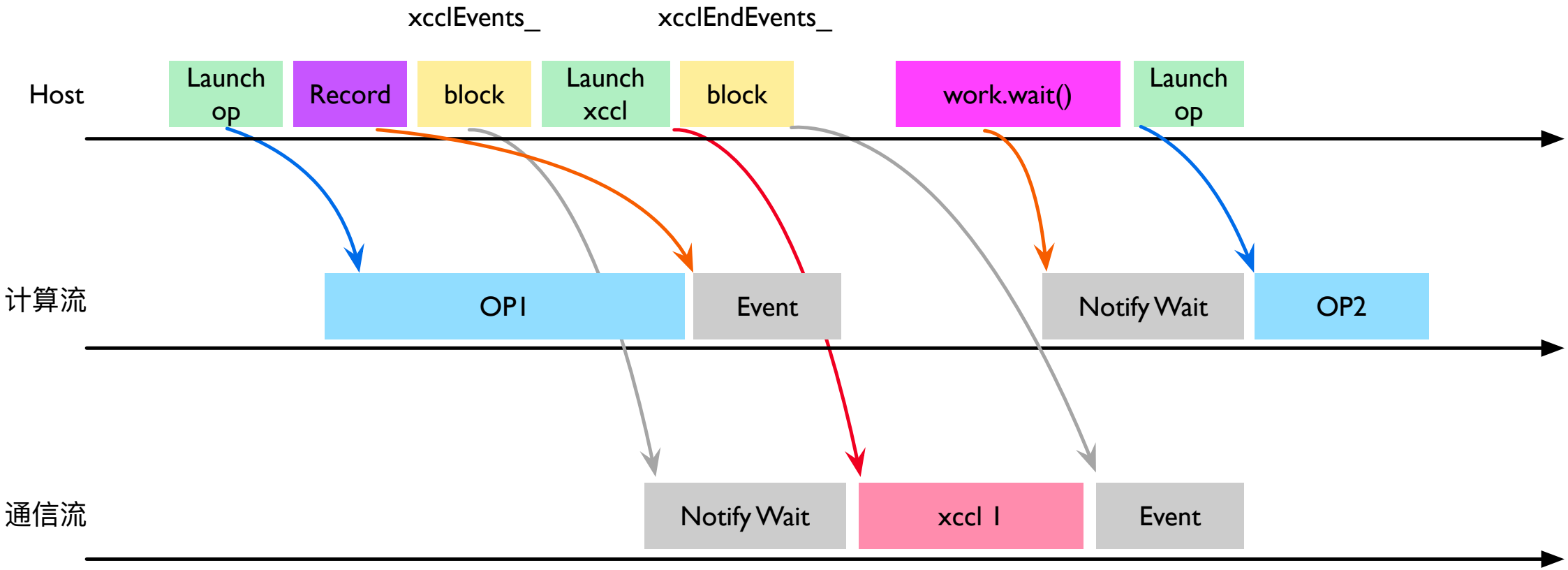
反过来思考

- 需要集合通信的输出作为计算流上算子的输入，怎么办？

Stream & Event 通信计算并行

- work.wait() 则做好时序控制，wait() 函数调用到 ProcessGroupXCCL::WorkXCCL::synchronizeStreams()，通过 block 使得 currentStream 转成计算流而非通信流。
- 那么 record 是在哪里被调用的？collective() 中通过 xcclEndEvents_ 在 xcclStream 上调用 record，用于 work.wait() 函数执行 block。

Stream & Event 通信计算并行



小结与思考

小结与思考

了解完本内容后：

1. AI 神经网络模型学习/训练阶段为什么要通信（AI 基础知识、训练推理、分布式并行）
2. XCCL 在 AI 系统中的位置（HPC 通信架构 to XCCL 通信架构）
3. 集合通信原语（All Reduce, etc.）
4. 集合网络拓扑（Hypercube 、 Ring 、 Torus 、 Fat-Tree 、 Dragonfly & Dragonfly+）
5. PyTorch 集合通信与计算并行
   - 通信域、进程、进程组与 Rank 间关系
   - PyTorch 使用分布式功能调用集合通信
   - PyTorch 执行时计算与通信并行底层原理

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/AISystem