曦云®系列通用计算 GPU

MCCL 编程指南

CSPG-23004-020-F3_V04　　2024-06-14

沐曦专有和三级保密信息　　本文档受 NDA 管控

飞腾信息MetaX Confidential 2024-11-18 15:00:00

版权所有 © 沐曦集成电路（上海）有限公司

更新记录

版本	日期	更新说明
V04	2024-06-14	更新以下章节： 3.1 通讯器管理
V03	2024-03-15	新增曦云®系列 GPU 产品信息 更新以下章节： 4.1.1 环境变量
V02	2023-12-29	新增以下章节： 4.1.2 共享内存
V01	2023-10-16	正式版本首次发布

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

目录

1. 概述 1
2. 简介 2
2.1 系统架构 3
3. 编程模型 4
3.1 通讯器管理 4
3.2 集合通讯 6
3.3 组调用 8
3.4 点对点通讯 9
3.5 数据类型 9
3.6 返回值类型 10
3.7 归约操作类型 10
4. 附录 11
4.1 调试信息 11
4.1.1 环境变量 11
4.1.2 共享内存 11
4.2 术语/缩略语 12

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

图目录

图 2-1 系统架构..... 3

飞腾信息 MetaX Confidential
2024-11-18 15:00:00

1. 概述

本文档详细描述了使用曦云®系列 GPU MetaX Collective Communications Library（MCCL）的软件开发方法，旨在帮助开发人员利用曦云系列 GPU 提供的计算资源，快速构建自己的应用。

本文档主要适用于利用曦云系列 GPU MCCL 的软件开发人员。

飞腾信息 MetaX Confidential
2024-11-18 15:00:00

2. 简介

MetaX Collective Communications Library（MCCL）是一个提供 GPU 间通信原语的库。MCCL 使用方便，可以进行拓扑感知与计算，是专用于 GPU 间通信的通讯库。

MCCL 提供以下集合通信原语：

- AllReduce
- Broadcast
- Reduce
- AllGather
- ReduceScatter
- P2P Send/Receive

MCCL 是基于 MXMACA®，并通过 MXMACA 的内存复制和 MXMACA 核函数来实现，以进行局部归约。另一方面，MCCL 通过算法优化，以实现快速同步，并最大限度地提高带宽利用率。

开发人员在使用 MCCL 后，不再需要对通讯进行优化。MCCL 支持节点内和跨节点的多 GPU 上进行集合通讯。它支持多种技术，包括 PCIe、MetaXLink、InfiniBand Verbs 和 IP Sockets。

MCCL 编程简单，提供 C 语言标准的 API，可以通过多种编程语言调用。MCCL 符合由 MPI 定义的 API 标准。MCCL 增加了 `stream` 参数，可以与 MXMACA 进行同步。MCCL 兼容多种 GPU 并行化模型：

- 所有 GPU 的单线程控制
- 多线程，例如，每个 GPU 使用一个线程
- 多进程，例如 MPI

MCCL 在深度学习框架中得到了很好的应用，其中 AllReduce 被大量用于神经网络训练。MCCL 提供的多 GPU 多节点通信可以有效扩展神经网络训练。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1 系统架构

曦云系列 GPU MCCL 的系统整体架构如下图所示：

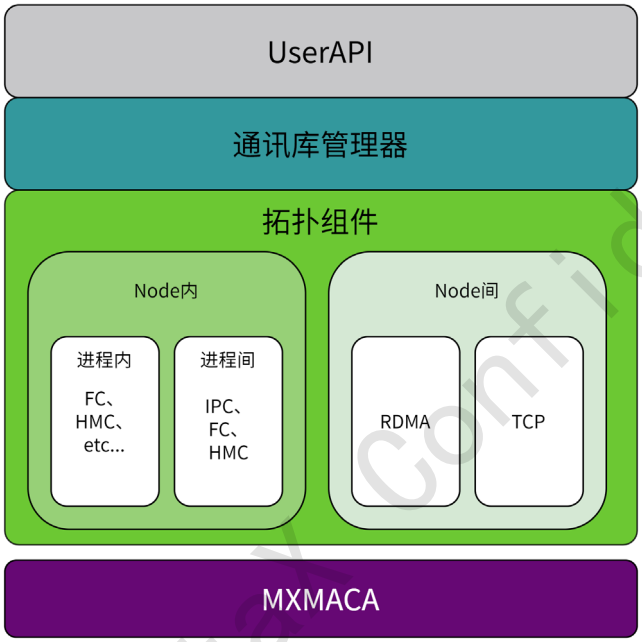

图 2-1 系统架构

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

3. 编程模型

3.1 通讯器管理

mcclGetVersion

```
mcclResult_t mcclGetVersion(int *version);
```

获取当前 MCCL 版本号。

mcclGetUniqueId

```
mcclResult_t mcclGetUniqueId(mcclUniqueId* uniqueId);
```

获取 ID 号，用于在多进程场景中初始化的 ID，由任意一个进程获取，并通过 MPI 或 socket 发送给其它进程用于初始化。多节点环境同样适用。

mcclCommInitRank

```
mcclResult_t mcclCommInitRank(mcclComm_t* comm, int nranks, mcclUniqueId commId, int rank);
```

创建通讯器，`nranks` 为通讯器总数，`commId` 为使用 `mcclGetUniqueId` 获取到的 ID，`rank` 为此通讯器的编号，其最小值为 `0`，最大值为 `nranks-1`。此通讯器所使用的设备为当前线程正在使用的设备。每个通讯器不能使用相同的设备。

mcclCommInitAll

```
mcclResult_t mcclCommInitAll(mcclComm_t* comm, int ndev, const int* devlist);
```

用于在单进程场景中一次性创建所有通讯器。`devlist` 包含了每个 rank 使用的设备。

mcclCommDestroy

```
mcclResult_t mcclCommDestroy(mcclComm_t comm);
```

销毁通讯器，结束集合通讯，用于正常通讯流程的退出。

mcclCommAbort

```
mcclResult_t mcclCommAbort(mcclComm_t comm);
```

中止通讯器，结束集合通讯，常用于从致命异常（如网络故障、节点故障或进程故障）中恢复通讯。当异常出现时，调用此 API，然后创建新的通信器重新通讯。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

mcclGetErrorString

```
const char* mcclGetErrorString(mcclResult_t result);
```

获取返回值的字符串。

mcclCommGetAsyncError

```
mcclResult_t mcclCommGetAsyncError(mcclComm_t comm, mcclResult_t
*asyncError);
```

获取异步操作的返回值。

mcclCommCount

```
mcclResult_t mcclCommCount(const mcclComm_t comm, int* count);
```

获取通讯器数量。

mcclCommMcDevice

```
mcclResult_t mcclCommMcDevice(const mcclComm_t comm, int* device);
```

获取当前通讯器所使用的 device。

mcclCommUserRank

```
mcclResult_t mcclCommUserRank(const mcclComm_t comm, int* rank);
```

获取当前通讯器的 rank 编号。

示例

单进程

```
int nranks = 4;
mcclComm_t comms[nranks];
int devs[nranks] = {0, 1, 2, 3};
mcclCommInitAll(comms, nranks, devs);
//...
//OPs...
//...
// finalizing MCCL
for (int i = 0; i < nranks; ++i)
   mcclCommDestroy(comms[i]);
```

多进程

```
mcclUniqueId id;
mcclComm_t comm;
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
//initializing MPI
MPICHECK(MPI_Init(&argc, &argv));
MPICHECK(MPI_Comm_rank(MPI_COMM_WORLD, &myRank));
MPICHECK(MPI_Comm_size(MPI_COMM_WORLD, &nRanks));

//get MCCL unique ID at rank 0 and broadcast it to all others
if (myRank == 0) mcclGetUniqueId(&id);
MPICHECK(MPI_Bcast((void *)&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD));

mcSetDevice(myRank);
mcclCommInitRank(&comm, nRanks, id, myRank);

//Other Ops.

//finalizing MCCL
mcclCommDestroy(comm);
```

3.2 集合通讯

所有 rank 都必须参与集合通讯，即每一种操作必须被所有 rank 调用，否则其它 rank 将等待。

mcclAllReduce

```
mcclResult_t mcclAllReduce(const void* sendbuff, void* recvbuff, size_t count, mcclDataType_t datatype, mcclRedOp_t op, mcclComm_t comm, mcStream_t stream);
```

在所有 rank 上都执行归约计算，最终所有 rank 将得到相同的结果，并保存在 `recvbuff` 中。`sendbuff` 为输入源。`count` 为数据的个数，`datatype` 为数据类型。`op` 为操作类型。`comm` 为通信器对象，`stream` 为 MXMACA 的 stream。

mcclBroadcast

```
mcclResult_t mcclBroadcast(const void* sendbuff, void* recvbuff, size_t count, mcclDataType_t datatype, int root, mcclComm_t comm, mcStream_t stream);

mcclResult_t mcclBcast(void* buff, size_t count, mcclDataType_t datatype, int root, mcclComm_t comm, mcStream_t stream);
```

`mcclBroadcast` 可将 `sendbuff` 中的数据复制到所有 rank 的 `recvbuff` 中（包括发送者的 `recvbuff`）。`count` 为数据的个数，`datatype` 为数据类型。`root` 为发送者的 rank 编号。`comm` 为通信器对象，`stream` 为 MXMACA 的 stream。对于接收者，`sendbuff` 可为空。

`mcclBcast` 与 `mcclBroadcast` 相同，若发送者的 `sendbuff` 与 `recvbuff` 相同，则可以用

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

mcclBcast 代替 mcclBroadcast。

mcclReduce

```
mcclResult_t mcclReduce(const void* sendbuff, void* recvbuff, size_t count, mcclDataType_t datatype, mcclRedOp_t op, int root, mcclComm_t comm, mcStream_t stream);
```

在 rank 为 root 的通讯器上执行归约计算。结果保存在 root 的 recvbuff 中。因此对于其它 rank，recvbuff 可为空。对于发送方，root 为发送的目标 rank。root 为执行归约的通讯器 rank 号。comm 为通信器对象，stream 为 MXMACA 的 stream。

mcclAllGather

```
mcclResult_t mcclAllGather(const void* sendbuff, void* recvbuff, size_t sendcount, mcclDataType_t datatype, mcclComm_t comm, mcStream_t stream);
```

将所有 rank 的 sendbuff 中的数据按 rank 顺序排列保存到每一个 rank 的 recvbuff 中，最终每个 rank 得到相同的结果。sendcount 为 sendbuff 的数据个数，recvbuff 的数据个数将会是 sendcount*N（N 为所有 rank 的个数）。调用者必须自己保证 recvbuff 有足够的空间。datatype 为数据类型，comm 为通信器对象，stream 为 MXMACA 的 stream。

mcclReduceScatter

```
mcclResult_t mcclReduceScatter(const void* sendbuff, void* recvbuff, size_t recvcount, mcclDataType_t datatype, mcclRedOp_t op, mcclComm_t comm, mcStream_t stream);
```

对所有 rank 的 sendbuff 进行归约，然后将结果分成 N 份，并按 rank 顺序分发给所有通讯器的 recvbuff 中。recvcount 为接收数据的个数，发送数据的个数是 recvcount*N。op 为操作类型。comm 为通信器对象，stream 为 MXMACA 的 stream。

示例

```
mcclGroupStart();
for (int i = 0; i < nranks; ++i) {
    mcclAllReduce((const void*)sendbuff[i], (void*)recvbuff[i],
                  size, mcclFloat, mcclSum, comms[i], s[i]);
}
mcclGroupEnd();

mcclGroupStart();
for (int i = 0; i < nranks; ++i) {
    mcclBroadcast((const void*)sendbuff[i], (void*)recvbuff[i],
                  size, mcclFloat, 0, comms[i], s[i]);
}
mcclGroupEnd();
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
mcclGroupStart();
for (int i = 0; i < nranks; ++i) {
    mcclReduce((const void*)sendbuff[i], (void*)recvbuff[i],
                        size, mcclFloat, mcclSum, 0, comms[i], s[i]);
}
mcclGroupEnd();

mcclGroupStart();
for (int i = 0; i < nranks; ++i) {
    mcclAllGather((const void*)sendbuff[i], (void*)recvbuff[i],
                        size, mcclFloat, comms[i], s[i]);
}
mcclGroupEnd();

mcclGroupStart();
for (int i = 0; i < nranks; ++i) {
    mcclAllReduceScatter((const void*)sendbuff[i], (void*)recvbuff[i],
                        size, mcclFloat, mcclSum, comms[i], s[i]);
}
mcclGroupEnd();
```

3.3 组调用

组调用可以把一系列 MCCL 操作合并在一起放入后台执行，不同 stream 的操作可互不影响地执行。这样可以实现 MCCL 操作在不同 stream 并发执行。

mcclGroupStart

```
mcclResult_t mcclGroupStart();
```

开始一个组编程。在此之后的所有 MCCL 函数都不会立即执行，而是放入一个执行队列中等待执行。

mcclGroupEnd

```
mcclResult_t mcclGroupEnd();
```

结束组编程。立即启动队列中的所有操作，并根据不同 stream 放入不同队列执行。在所有操作完成之前，阻塞当前线程。

示例

```
mcclGroupStart();
mcclSend(sendbuff[0], size, mcclFloat, 1, comms[0], s[0]);
mcclRecv(recvbuff[1], size, mcclFloat, 0, comms[1], s[1]);
```

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

```
mcclGroupEnd();
```

3.4 点对点通讯

点对点的通讯，即把数据从一个 rank 复制给另一个 rank，也就是将显存中的内容，由一个显卡复制到另一个显卡中。

mcclSend

```
mcclResult_t mcclSend(const void* sendbuff, size_t count, mcclDataType_t datatype, int peer, mcclComm_t comm, mcStream_t stream);
```

将 sendbuff 中的内存发送给接收端 peer。count 为数据的个数，datatype 为数据类型。

comm 为通信器对象，stream 为 MXMACA 的 stream。

mcclRecv

```
mcclResult_t mcclRecv(void* recvbuff, size_t count, mcclDataType_t datatype, int peer, mcclComm_t comm, mcStream_t stream);
```

从 peer 端接收数据到 recvbuff 中。count 为数据的个数，datatype 为数据类型。

comm 为通信器对象，stream 为 MXMACA 的 stream。

示例

```
mcclSend(sendbuff[0], size, mcclFloat, 1, comms[0], s[0]);
mcclRecv(recvbuff[1], size, mcclFloat, 0, comms[1], s[1]);
```

3.5 数据类型

MCCL 的数据类型用于指定各操作原语所使用的数据类型，对应 MXMACA 的数据类型。对应关系如下：

- mcclInt8：char
- mcclChar：char
- mcclUint8：unsigned char
- mcclInt32：int
- mcclInt：int
- mcclUint32：unsigned int
- mcclInt64：long
- mcclUint64：unsigned long

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- mcclFloat16：half
- mcclHalf：half
- mcclFloat32：float
- mcclFloat：float
- mcclFloat64：double
- mcclDouble：double
- mcclBfloat16：bfloat16

3.6 返回值类型

- `mcclSuccess`：成功
- `mcclUnhandledMacaError`：调用 MXMACA API 时返回错误
- `mcclSystemError`：调用系统函数错误
- `mcclInternalError`：MCCL 内部错误，可能是内存错误
- `mcclInvalidArgument`：参数错误
- `mcclInvalidUsage`：使用错误，可能是用户编程导致
- `mcclRemoteError`：多节点场景时，网络连接错误

3.7 归约操作类型

- mcclSum：求和
- mcclProd：求乘积
- mcclMax：求最大值
- mcclMin：求最小值
- mcclAvg：求平均值

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

4. 附录

4.1 调试信息

4.1.1 环境变量

MCCL_DEBUG

通过环境变量 MCCL_DEBUG 设置日志输出等级，可选等级如下：

- 0：关闭日志输出
- 1：仅打印 error 级别日志
- 2：输出 warning 及 error 级别日志
- 3：输出 information、warning 及 error 级别日志
- 4：输出全部日志

MCCL_MAX_NCHANNELS

环境变量 MCCL_MAX_NCHANNELS 用于限制 MCCL 使用的通道数。减少通道数会减少 MCCL 用于通信 kernel 函数的 blocks 数量和 buffer 内存大小，因而影响 GPU 计算资源。可设置的值应大于等于 1。

MCCL_MIN_NCHANNELS

环境变量 MCCL_MIN_NCHANNELS 用于控制 MCCL 使用的最小通道数。增加通道数会增加 MCCL 用于通信 kernel 函数的 blocks 数量和 buffer 内存大小，这可能会带来性能的提升，但会使用更多 GPU 计算资源。默认值因不同平台而异，可设置的值应大于等于 1。

MCCL_BUFFSIZE

环境变量 MCCL_BUFFSIZE 控制用于两 GPU 间 MCCL 数据通信的 buffer 大小。如果在使用 MCCL 时遇到内存不足问题（例如 GPU 开启 VF 时，设备内存有所限制），或者认为不同的 buffer 大小可能带来性能提升时，可以考虑设置此环境变量。设置的值须为整数，单位为 bytes，默认值为 8388608 (8M)，推荐设置的值为 2 的幂。

4.1.2 共享内存

为了进程或线程间通信，MCCL 会在 /dev/shm 中创建共享内存。因而操作系统对共享内存的限制需要相应地增加。如果共享内存不够，MCCL 会在初始化时失败。增加共享内存大小可参考操作系统相关文档。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

Docker 容器的默认配置会限制共享内存和固页内存的大小。当在 Docker 容器中运行 MCCL 应用程序时，需调整共享内存大小以确保程序可以成功运行。例如，在 Docker 运行时可以添加如下命令：

```
--shm-size=1g --ulimit memlock=-1
```

4.2 术语/缩略语

术语/缩略语	全称	说明
Docker		一个开源的应用容器引擎
MCCL	Metax Collective Communications Library	沐曦提供 GPU 间通信原语的库
MetaXLink		沐曦 GPU D2D 接口总线
MPI	Message Passing Interface	消息传递接口
MXMACA	MetaX Advanced Compute Architecture	沐曦推出的 GPU 软件栈，包含了沐曦 GPU 的底层驱动、编译器、数学库及整套软件工具套件
socket		网络编程标准接口，套接字

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

声明

版权所有 ©2023-2024 沐曦集成电路（上海）有限公司。保留所有权利。

本文档中呈现的信息属于沐曦集成电路（上海）有限公司和/或其附属公司（以下统称为“沐曦”），非经沐曦事先书面许可，任何实体或个人均不得获得本文档的副本，且无权以任何方式处理本文档，包括但不限于使用、复制、修改、合并、出版、发行、销售或传播本文档的部分或全部。

本文档内容仅供参考，不提供任何形式的、明示或暗示的保证，包括但不限于对适销性、适用于任何目的和/或不侵权的保证。在任何情况下，沐曦均不对因本文档引起的、由本文档造成的、或与之相关的任何索赔、损害或其他责任负责。

沐曦保留自行决定随时更改、修改、添加或删除本文档的部分或全部的权利。沐曦保留最终解释权。

沐曦、MetaX 和其他沐曦图标是沐曦的商标。本文档中提及的所有其他商标和商品名称均为其各自所有者的财产。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00