The Memory Allocation Kinds Side Document

Version 1.0

MPI Forum Hybrid Working Group
(September 2023)

Acknowledgments

Authors
Rohit Zambre, James Dinan, Maria Garzaran, Daniel Holmes, Edgar Gabriel, Michael Klemm, and Pedram Alizadeh

Contents

Chapter 1

Overview

Modern computing systems contain a variety of memory types, each closely associated with a distinct type of computing hardware. For example, compute accelerators such as GPUs typically feature their own memory that is distinct from the memory attached to the host processor. Additionally, GPUs from different vendors also differ in their memory types. The differences in memory types influence feature availability and performance behavior of an application running on such modern systems. Hence, MPI libraries need to be aware of and support additional memory types. For a given type of memory, MPI libraries need to know the associated memory allocator and the limitations on memory access. The different memory kinds capture the differentiating information needed by MPI libraries for different memory types.

This MPI side document defines the memory allocation kinds and their associated restrictors that users can use to query the support for different memory kinds provided by the MPI library. These definitions supplement those found in section 11.4.3 of the MPI standard, which also explains their usage model.

Chapter 2

Definitions

This section contains definitions of memory allocation kinds and their restrictors.

2.1 Kind: cuda

The cuda memory kind refers to the memory allocated by the CUDA runtime system [1].

Restrictors

- host: Support for memory allocations on the host system that are page-locked for direct access from the CUDA device (e.g., memory allocations from the `cudaHostAlloc()` function). These memory allocations are attributed with `cudaMemoryTypeHost`.
- device: Support for memory allocated on a CUDA device (e.g., memory allocations from the `cudaMalloc()` function). These memory allocations are attributed with `cudaMemoryTypeDevice`.
- managed: Support for memory that is managed by CUDA's Unified Memory system (e.g., memory allocations from the `cudaMallocManaged()` function). These memory allocations are attributed with `cudaMemoryTypeManaged`.

Example 2.1 This example demonstrates the usage of the different kinds to perform communication in a manner that is supported by the underlying MPI library.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
    int cuda_device_aware = 0;
    int cuda_managed_aware = 0;
    int len = 0, flag = 0;
    int *managed_buf = NULL;
    int *device_buf = NULL, *system_buf = NULL;
    MPI_Info info;
    MPI_Session session;
    MPI_Group wgroup;
```

```
    MPI_Comm system_comm;
    MPI_Comm cuda_managed_comm = MPI_COMM_NULL;
    MPI_Comm cuda_device_comm = MPI_COMM_NULL;

    MPI_Info_create(&info);
    MPI_Info_set(info, "mpi_memory_alloc_kinds",
                       "system,cuda:device,cuda:managed");
    MPI_Session_init(info, MPI_ERRORS_ARE_FATAL, &session);
    MPI_Info_free(&info);

    MPI_Session_get_info(session, &info);
    MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
                        &len, NULL, &flag);

    if (flag) {
        char *val, *valptr, *kind;

        val = valptr = (char *) malloc(len);
        if (NULL == val) return 1;

        MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
                            &len, val, &flag);

        while ((kind = strsep(&val, ",")) != NULL) {
            if (strcasecmp(kind, "cuda:managed") == 0) {
                cuda_managed_aware = 1;
            }
            else {
                if (strcasecmp(kind, "cuda:device") == 0) {
                    cuda_device_aware = 1;
                }
        }
        free(valptr);
    }

    MPI_Info_free(&info);

    MPI_Group_from_session_pset(session, "mpi://WORLD" , &wgroup);

    // Create a communicator for operations on system memory
    MPI_Info_create(&info);
    MPI_Info_set(info, "mpi_assert_memory_alloc_kinds", "system");
    MPI_Comm_create_from_group(wgroup,
            "org.mpi-side-doc.mem-kind.example.system",
            info, MPI_ERRORS_ABORT, &system_comm);

    MPI_Info_free(&info);

    // Check if all processes have CUDA managed support
    MPI_Allreduce(MPI_IN_PLACE, &cuda_managed_aware, 1, MPI_INT,
                  MPI_LAND, system_comm);

    if (cuda_managed_aware) {
        // Create a communicator for operations that use
```

```
        // CUDA managed buffers.
        MPI_Info_create(&info);
        MPI_Info_set(info, "mpi_assert_memory_alloc_kinds",
                     "cuda:managed");
        MPI_Comm_create_from_group(wgroup,
            "org.mpi-side-doc.mem-kind.example.cuda.managed",
            info, MPI_ERRORS_ABORT, &cuda_managed_comm);
        MPI_Info_free(&info);
    }
    else {
        // Check if all processes have CUDA device support
        MPI_Allreduce(MPI_IN_PLACE, &cuda_device_aware, 1, MPI_INT,
                      MPI_LAND, system_comm);
        if (cuda_device_aware) {
            // Create a communicator for operations that use
            // CUDA device buffers.
            MPI_Info_create(&info);
            MPI_Info_set(info, "mpi_assert_memory_alloc_kinds",
                         "cuda:device");
            MPI_Comm_create_from_group(wgroup,
                "org.mpi-side-doc.mem-kind.example.cuda.device",
                info, MPI_ERRORS_ABORT, &cuda_device_comm);
            MPI_Info_free(&info);
        }
        else {
            printf("Warning: cuda alloc kind not supported\n");
        }
    }

    MPI_Group_free(&wgroup);

    if (cuda_managed_aware) {
        // Allocate managed buffer and initialize it
        cudaMallocManaged(&managed_buf, sizeof(int));
        *managed_buf = 1;

        // Perform communication using cuda_managed_comm
        // if it's available.
        MPI_Allreduce(MPI_IN_PLACE, managed_buf, 1, MPI_INT,
                      MPI_SUM, cuda_managed_comm);

        cudaFree(managed_buf);
    }
    else {
        // Allocate system buffer and initialize it
        system_buf = malloc(sizeof(int));
        *system_buf = 1;

        // Allocate CUDA device buffer and initialize it
        cudaMalloc(&device_buf, sizeof(int));
        cudaMemcpy(device_buf, system_buf, sizeof(int),
                   cudaMemcpyHostToDevice);

        if (cuda_device_aware) {
```

```
            // Perform communication using cuda_comm
            // if it's available.
            MPI_Allreduce(MPI_IN_PLACE, device_buf, 1, MPI_INT,
                          MPI_SUM, cuda_comm);
        }
        else {
            // Otherwise, copy data to a system buffer,
            // use system_comm, and copy data back to device buffer
            cudaMemcpy(system_buf, device_buf, sizeof(int),
                   cudaMemcpyDeviceToHost);
            MPI_Allreduce(MPI_IN_PLACE, system_buf, 1, MPI_INT,
                          MPI_SUM, system_comm);
            cudaMemcpy(device_buf, system_buf, sizeof(int),
                   cudaMemcpyHostToDevice);

            cudaFree(device_buf);
            free(system_buf);
        }
    }

    if (cuda_managed_comm != MPI_COMM_NULL)
        MPI_Comm_disconnect(&cuda_managed_comm);
    if (cuda_device_comm != MPI_COMM_NULL)
        MPI_Comm_disconnect(&cuda_device_comm);
    MPI_Comm_disconnect(&system_comm);

    MPI_Session_finalize(&session);

    return 0;
}
```

2.2 Kind: rocm

The rocm memory kind refers to the memory allocated by the ROCm runtime system [2].

Restrictors

- host: Support for memory allocated on the host system that is page-locked for direct access from the ROCm device (e.g., memory allocations from the `hipHostMalloc()` function). These memory allocations are attributed with `hipMemoryTypeHost`.

- device: Support for memory allocated on the ROCm device (e.g., memory allocations from the `hipMalloc()` function). These memory allocations are attributed with `hipMemoryTypeDevice`.

- managed: Support for memory that is managed automatically by the ROCm runtime (e.g., memory allocations from the `hipMallocManaged()` function). These memory allocations are attributed with `hipMemoryTypeManaged`.

2.3 Kind: levelzero

The levelzero memory kind refers to the memory allocated by the Level Zero runtime system [3].

Restrictors

- host: Support for memory allocated on the host that is accessible by Level Zero devices (e.g., memory allocations from the `zeMemAllocHost()` function). These memory allocations are attributed with `ZE_MEMORY_TYPE_HOST`.

- device: Support for memory allocated on a Level Zero device (e.g., memory allocations from the `zeMemAllocDevice()` function). These memory allocations are attributed with `ZE_MEMORY_TYPE_DEVICE`.

- shared: Support for memory allocated that will be shared between the host and one or more Level Zero devices (e.g., memory allocations from the `zeMemAllocShared()` function). These memory allocations are attributed with `ZE_MEMORY_TYPE_SHARED`.

Annex A

Changelog

This annex summarizes changes from the previous versions of the *Memory Allocation Kinds* side document to the version presented by this document.

A.1 Version 1.0

1. The first version of this document.

Bibliography

[1] CUDA Runtime API. https://docs.nvidia.com/cuda/cuda-runtime-api/.

[2] HIP Programming Manual. https://rocm.docs.amd.com/en/latest/reference/hip.html.

[3] Level Zero Programming Guide. https://spec.oneapi.io/level-zero/latest/core/PROG.html.