

Memory Allocation Kinds
An MPI Side Document
Version 1.0

MPI Forum Hybrid Working Group
(XXXX 2023)

This document defines memory allocation kinds that are compatible with the MPI-4.1 standard.

Version 1.0: XXXX 2023 This document defines the first set of memory allocation kinds. This and future versions of this side document to the MPI standard are ratified by the MPI Forum, but not an official part of the standard itself.

Acknowledgments

This document represents the work of many people who have served on the **MPI Forum Hybrid Working Group**. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has led to the **Memory Allocation Kinds** document. The technical development was carried out by subgroups, whose work was reviewed by the full committee.

Those who served as primary coordinators in Version 1.0 are:

- James Dinan, Rohit Zambre, Nvidia CUDA
- Pedram Alizadeh, Edgar Gabriel, Michael Klemm, AMD ROCm
- Maria Garzaran, Dan Holmes, Intel Level Zero

The following list includes some of the active participants in the **MPI Forum Hybrid Working Group**.

| | | | |
|-----------------|-------------------|----------------------|--------------------|
| Pedram Alizadeh | James Dinan | Dmitry Durnov | Edgar Gabriel |
| Maria Garzaran | Yanfei Guo | Khaled Hamidouche | Dan Holmes |
| Michael Klemm | Guillaume Mercier | Christoph Niethammer | Kenneth Raffenetti |
| Joseph Schuart | Rohit Zambre | Hui Zhou | |

Contents

| | |
|-------------------------------|----|
| Acknowledgments | iv |
| 1 Overview | 1 |
| 2 Definitions | 2 |
| 2.1 Kind: cuda | 2 |
| Restrictors | 2 |
| 2.2 Kind: rocm | 2 |
| Restrictors | 2 |
| 2.3 Kind: levelzero | 3 |
| Restrictors | 3 |
| 3 Examples | 4 |
| Bibliography | 8 |

1 Chapter 1

2 Overview

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

13 This MPI side document defines the memory allocation kinds and their associated
14 restrictors that users can use to query the support for different memory kinds provided
15 by the MPI library. These definitions supplement those found in section 11.4.3 of the MPI
16 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].

Example 3.1 showcases usage of some of the memory kinds defined in this section.

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`.

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`.

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

4 2.3 Kind: `levelzero`

5 The `levelzero` memory kind refers to the memory allocated by the Level Zero runtime sys-
6 tem [3].

7 Restrictors

- 8 • `host`: Support for memory allocated on the host that is accessible by Level Zero de-
9 vices (e.g., memory allocations from the `zeMemAllocHost()` function). These memory
10 allocations are attributed with `ZE_MEMORY_TYPE_HOST`.
- 11 • `device`: Support for memory allocated on a Level Zero device (e.g., memory allocations
12 from the `zeMemAllocDevice()` function). These memory allocations are attributed
13 with `ZE_MEMORY_TYPE_DEVICE`.
- 14 • `shared`: Support for memory allocated that will be shared between the host and one
15 or more Level Zero devices (e.g., memory allocations from the `zeMemAllocShared()`
16 function). These memory allocations are attributed with `ZE_MEMORY_TYPE_SHARED`.

Chapter 3

Examples

This section includes examples demonstrating the usage of memory kinds defined in Chapter 2.

Example 3.1 This CUDA 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 <assert.h>
#include <mpi.h>
#include <cuda_runtime.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;
    int nranks = 0;
    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) {
```

```

1      char *val, *valptr, *kind;
2
3      val = valptr = (char *) malloc(len);
4      if (NULL == val) return 1;
5
6      MPI_Info_get_string(info, "mpi_memory_alloc_kinds",
7                          &len, val, &flag);
8
9      while ((kind = strsep(&val, ",")) != NULL) {
10         if (strcasecmp(kind, "cuda:managed") == 0) {
11             cuda_managed_aware = 1;
12         }
13         else if (strcasecmp(kind, "cuda:device") == 0) {
14             cuda_device_aware = 1;
15         }
16     }
17     free(valptr);
18 }
19
20 MPI_Info_free(&info);
21
22 MPI_Group_from_session_pset(session, "mpi://WORLD" , &wgroup);
23
24 // Create a communicator for operations on system memory
25 MPI_Info_create(&info);
26 MPI_Info_set(info, "mpi_assert_memory_alloc_kinds", "system");
27 MPI_Comm_create_from_group(wgroup,
28                             "org.mpi-side-doc.mem-kind.example.system",
29                             info, MPI_ERRORS_ABORT, &system_comm);
30 MPI_Info_free(&info);
31
32 MPI_Comm_size(system_comm, &nrank);
33
34 /** Check for CUDA awareness */
35 // Check if all processes have CUDA managed support
36 MPI_Allreduce(MPI_IN_PLACE, &cuda_managed_aware, 1, MPI_INT,
37               MPI_LAND, system_comm);
38
39 if (cuda_managed_aware) {
40     // Create a communicator for operations that use
41     // CUDA managed buffers.
42     MPI_Info_create(&info);
43     MPI_Info_set(info, "mpi_assert_memory_alloc_kinds",
44                   "cuda:managed");
45     MPI_Comm_create_from_group(wgroup,
46                               "org.mpi-side-doc.mem-kind.example.cuda.managed",
47                               info, MPI_ERRORS_ABORT, &cuda_managed_comm);
48     MPI_Info_free(&info);
49 }
50 else {
51     // Check if all processes have CUDA device support
52     MPI_Allreduce(MPI_IN_PLACE, &cuda_device_aware, 1, MPI_INT,
53                   MPI_LAND, system_comm);
54     if (cuda_device_aware) {

```

```

1      // Create a communicator for operations that use
2      // CUDA device buffers.
3      MPI_Info_create(&info);
4      MPI_Info_set(info, "mpi_assert_memory_alloc_kinds",
5                      "cuda:device");
6      MPI_Comm_create_from_group(wgroup,
7                                "org.mpi-side-doc.mem-kind.example.cuda.device",
8                                info, MPI_ERRORS_ABORT, &cuda_device_comm);
9      MPI_Info_free(&info);
10     }
11     else {
12         printf("Warning: cuda alloc kind not supported\n");
13     }
14 }
15
16 MPI_Group_free(&wgroup);
17
18 /** Execute according to level of CUDA awareness */
19 if (cuda_managed_aware) {
20     // Allocate managed buffer and initialize it
21     cudaMallocManaged((void*)&managed_buf, sizeof(int));
22     *managed_buf = 1;
23
24     // Perform communication using cuda_managed_comm
25     // if it's available.
26     MPI_Allreduce(MPI_IN_PLACE, managed_buf, 1, MPI_INT,
27                  MPI_SUM, cuda_managed_comm);
28
29     assert((*managed_buf) == n ranks);
30
31     cudaFree(managed_buf);
32 }
33 else {
34     // Allocate system buffer and initialize it
35     system_buf = (int*)malloc(sizeof(int));
36     *system_buf = 1;
37
38     // Allocate CUDA device buffer and initialize it
39     cudaMalloc((void*)&device_buf, sizeof(int));
40     cudaMemcpyAsync(device_buf, system_buf, sizeof(int),
41                    cudaMemcpyHostToDevice);
42
43     cudaStreamSynchronize(0);
44     if (cuda_device_aware) {
45         // Perform communication using cuda_device_comm
46         // if it's available.
47         MPI_Allreduce(MPI_IN_PLACE, device_buf, 1, MPI_INT,
48                      MPI_SUM, cuda_device_comm);
49
50         assert((*device_buf) == n ranks);
51     }
52     else {
53         // Otherwise, copy data to a system buffer,
54         // use system_comm, and copy data back to device buffer

```

```

1         cudaMemcpyAsync(system_buf, device_buf, sizeof(int),
2                           cudaMemcpyDeviceToHost);
3
4         cudaStreamSynchronize(cudaStreamDefault);
5         MPI_Allreduce(MPI_IN_PLACE, system_buf, 1, MPI_INT,
6                       MPI_SUM, system_comm);
7         cudaMemcpyAsync(device_buf, system_buf, sizeof(int),
8                           cudaMemcpyHostToDevice);
9
10        cudaStreamSynchronize(cudaStreamDefault);
11        assert((*system_buf) == nranks);
12    }
13
14    cudaFree(device_buf);
15    free(system_buf);
16 }
17
18 if (cuda_managed_comm != MPI_COMM_NULL)
19     MPI_Comm_disconnect(&cuda_managed_comm);
20 if (cuda_device_comm != MPI_COMM_NULL)
21     MPI_Comm_disconnect(&cuda_device_comm);
22 MPI_Comm_disconnect(&system_comm);
23
24 MPI_Session_finalize(&session);
25
26 return 0;
27 }

```

¹ Bibliography

- ² [1] CUDA Runtime API. <https://docs.nvidia.com/cuda/cuda-runtime-api/>.
- ³ [2] HIP Programming Manual. <https://rocm.docs.amd.com/en/latest/reference/hip.html>.
- ⁴ [3] Level Zero Programming Guide. [https://spec.oneapi.io/level-](https://spec.oneapi.io/level-zero/latest/core/PROG.html)
- ⁵ [zero/latest/core/PROG.html](https://spec.oneapi.io/level-zero/latest/core/PROG.html).