

CPU + GPU architectures

- Memory might consist of multiple, distinct memory spaces
 - CPU might not be able to operate on all memory addresses with standard OS access mechanisms
 - GPUs might not be able to access memory addresses that are located on the CPU in all instances
- Increasing interest in
 - Initiate Operations from GPU kernels (e.g. partitioned communication)
 - Closer integration of GPU programming models with MPI (e.g. incorporating MPI communications in Streams/Graphs)
 - Executing some MPI functionality on GPUs: (e.g. Pack/Unpack)
- What are the changes required to the MPI specification to enable these scenarios from a high-level perspective

Current Status: GPUs + MPI (I)

- One can use device buffers in communication operations
 - Requirements on device buffer synchronizations are not clearly stated and/or enforced
- Some additional aspect might still require clarifications, e.g.
 - is a derived data type spanning multiple memory spaces (e.g host + device memory, or memory from two different devices) allowed
 - Portable vs. non-portable datatypes when using GPUs
- Limitations of current implementation, e.g.
 - e.g. supporting only 1 GPU per MPI process


Current status: GPUs + MPI (II)

- One can not invoke MPI functions from a GPU kernel
 - Not explicitly disallowed in the MPI specification, just the reality of
 - Having multiple memory spaces
 - Different ISAs between CPU and GPU
 - GPUs not being able to execute certain OS functionalities
 - GPUs not being very good at executing certain type of code sequences
 - GPUs are mentioned in MPI 4.0 in two examples (interestingly both being Fortran examples) Example 19.8 and 19.10

Header file

- Do we need one (or multiple separate) header file(s) for supporting GPU functionality in MPI?

```
__device__ MPI_Send_special()
```



- Regular C compiler will fail if this is part of mpi.h
- MPI library might support multiple GPU vendors and/or programming models simultaneously
- Same (non-standard C) keyword might be used in multiple programming models but might not mean exactly the same thing
- Implications on ABI

Header file

- Do we need one (or multiple) for GPU functionality in MPI

```
__device__ MPI_Send_special()
```



- Regular C compiler will fail if this is part of mpi.h
- MPI library might support multiple GPU vendors and/or programming models simultaneously
- Same (non-standard C) keyword might be used in multiple programming models but might not mean exactly the same thing
- Implications on ABI

Takeaways:

- Separate header file not required
- It is acceptable to have some of the MPI function prototypes in mpi.h being protected by compiler/vendor macros
- Does this apply to Fortran as well?
- ABI implications
 - ABI will most likely not extend to optional features in side documents

Header file

```
#ifdef CUDA
__device__ MPI_Send_special(...,int streamType, void *stream);
#endif
#ifdef ROCM
__device__ MPI_Send_special(...,int streamType, void *stream)
#endif
...
```

or

```
#ifdef CUDA
__device__ MPI_Send_special(..., cudaStream* stream);
#endif
#ifdef ROCM
__device__ MPI_Send_special(..., hipStream* stream)
#endif
...
```

- Redefinition of prototype if both CUDA and ROCM are defined
- Single implementation possible, with the separation for various GPU types happening in the function

- Introduces constructs of another programming model in MPI
- Different prototypes for different vendors

Header file

```
#ifdef CUDA
__device__ MPI_Send_s
#endif
#ifdef ROCM
__device__ MPI_Send_s
#endif
...
```

or

```
#ifdef CUDA
__device__ MPI_Send_special(..., cudaStream* stream);
#endif
#ifdef ROCM
__device__ MPI_Send_special(..., hipStream* stream)
#endif
...
```

Takeaways:

- Introducing abstractions for constructs of other programming language for the language independent interface (e.g. STREAM)
- Mapping of keywords to actual implementation not yet clear
 - Manual: e.g. define STREAM to cudaStream, hipStream etc.
-> potential explosion in number of interfaces
(GPU programming models x languages supported by MPI)
 - Automatic: e.g. C++ templates
- Suggestion: the function prototype in the MPI spec could avoid listing GPU programming model specific constructs (e.g. __device__) and just indicate that it needs to be able run on device. This is the approach that was taken by OpenMP.

- Introduces constructs of another programming model in MPI
- Different prototypes for different vendors

MPI handles

- Is a GPU kernel allowed to access an MPI handle
- The handle itself might not be the issue (it's a single int or pointer), but the data structure behind it
 - Data structure referenced by the handle might contain one or more dynamic arrays/structures
 - Implementation options
 - Communication with main process to retrieve required elements through proxy thread
 - Allocations could be handled in a manner that makes it accessible on GPUs (e.g. `xxxHostMalloc()` or `xxxMallocManaged()`)
- Except for `MPI_Status`, accessing the data structures is only allowed through MPI functions

MPI Functions (I)

- Option1 :
 - a) Formalize that MPI functions can by default only be executed from CPU/host memory
 - b) Option 1a:
 - Define subset of functions that can be invoked from GPU
 - Main MPI specification vs. (vendor specific) side document
 - The same function being used on GPUs and CPUs mandates using the same MPI handles (e.g. `MPI_Request`, `MPI_Status`, `MPI_Datatype`) on both CPU and GPU
 - Option 1b:
 - Define new functions that can be executed on GPUs
 - Would allow (but not automatically necessitate) to define new MPI handle types that are valid on GPUs or CPU+GPU (e.g. `MPI_Request` vs. `MPI_Hrequest`)
- Both versions might require additional sync functionality is required (e.g. `MPI_Request_sync()`) when accessing a handle in a different memory space

MPI Functions (I)

- Option1 :

- a) Formalize that MPI functions access memory

- b) Option 1a:

- Define subset of functions that can be executed on GPUs
 - Main MPI specification vs. (vs. Option 1b)
 - The same function being used on GPUs and CPUs mandates using the same MPI handles (e.g. `MPI_Request`, `MPI_Status`, `MPI_Datatype`) on both CPU and GPU

- Option 1b:

- Define new functions that can be executed on GPUs
 - Would allow (but not automatically necessitate) to define new MPI handle types that are valid on GPUs or CPU+GPU (e.g. `MPI_Request` vs. `MPI_Hrequest`)

- Both versions might require additional sync functionality is required (e.g. `MPI_Request_sync()`) when accessing a handle in a different memory space

Takeaways:

- Option 1a and 1b could coexist
- Depending on the functionality executed from the GPU kernel, only a subset of the data structure(s) might need to be accessed from the kernel
- Whether synchronizing of Requests etc. is required might be use-case/functionality dependent. Better to wait for scenario that actually needs that instead of making a blanket statement.
- Suggestion: a function that allows to indicate whether a Request/Communicator etc. will be used in different memory spaces, somewhat similar to the memtype info keys

MPI Functions (II)

- Option 2:
 - Formalize that all MPI functions can also be executed from GPU
 - > probably not realistic
- Option 3:
 - Continue the current status quo, i.e. the MPI specification does not make a statement on this topic
 - Default option if we don't address the topic
 - Could lead to fragmentation among MPI libraries due to undefined behavior

MPI Functions (III)

- Device function syntax clarification
 - single thread invocation vs. collective across all threads in a thread-block
 - Thread-collective execution of an MPI function is also a new term/concept
- Does it make sense to allow different vendors/architecture to allow different behavior of the same function