IMPLEMENTATION OF MPI IO AND DATATYPE ON CUDA GPU

Xindi Zuo, Ran Liao, Junchi Chen, Yuening Yang, Yijun Ma

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Message Passing Interface (MPI) is a standard message passing interface designed for parallel computing. This paper describes CUDA-MPI, a special implementation for MPI with the help of CUDA, a parallel computing platform and programming model that takes advantage of the power of the GPU. It focuses specifically on MPI IO and Datatypes. Several implementation trials are documented and comparably high efficiency is achieved.

1. INTRODUCTION

Message Passing Interface (MPI)[1] is a message-passing standard specification defined by MPI Forum for parallel computing architectures. Graphics Processing Unit (GPU) is a powerful piece of hardware for parallel computing, and is now a focus for both researching and industrial use, largely due to the current tide of machine learning technology. CUDA MPI project arises with the expectation of enabling users to write code conforming only to normal MPI standards while still being able to leverage GPU hardware.

We contribute to the current project of CUDA MPI in two modules: IO and user-defined datatype.

Motivation. We choose the IO module because it digs deep in both hardware and software system architecture and thus is possible for various improvements. Also, IO part can often be the bottleneck of an application, and thus dealing with IO efficiently would be essential to performance. These points make more sense for CUDA implementation of MPI because there's no direct passage between disk and GPU memory on off-the-shelf machines. On the other hand, we choose the user-defined datatype because it is an important feature provided by MPI standard due to the heterogeneity and non-contiguity of user data, and is the fundamental basis of many advanced features. Moreover, under a CUDA scenario, custom datatype has more possible optimizations in terms of concurrency.

Contribution. (1) **IO** There are two main categories of IO procedures: blocking IO and non-blocking IO. For blocking IO, we implemented a basic version and a buffered version, and the latter is significantly better in performance; we also carried out optimizations in terms of concurrency. For non-blocking IO, we implemented using Linux support for asynchronous IO. (2) **Datatype** After defining a class and relevant APIs of custom datatypes, we experimented with algorithmic improvements on the baseline implementation and gained better performance.

2. CUDA-GPU IMPLEMENTATION OF MPI

This section talks about the environment that we build our modules on. MPI processes are mapped to CUDA threads, and more specifically, threads are mapped one per GPU block. Thus they share no registers, no control flows, and no shared memory. This prevents different MPI processes from being limited by the single control flow of a warp execution.

Memory management. The private state of a particular MPI process is either on the stack or the heap of the CUDA thread that represents the process. This is done through normal CUDA memory. On the other hand, unified memory[2, 3] is used to manage memory that shares between device and host. This is a new memory model available since CUDA 6.0, which gives the programmer a unified memory address space accessible to both processors (GPU and CPU).

Device-host communication. First, unified memory is initialized by *cudaMallocManaged()* on the host side, allocating all possible free space and managing it internally. During the communication, the CUDA MPI internal memory allocation is called on the device side; an allocated buffer is then loaded with the control messages or data; a shared state is changed by the device, and the host checks that state in a while loop during device code execution, thus receiving the operations to perform and its data. After the host finishes executing, the device return to execution.

3. BASIC IO IMPLEMENTATION

The IO schema is shown in figure 1. The control message of IO, as well as the data, relies on the communication between device and host. For each IO operation, we rely on Linux's

support on the host side and meanwhile maintain a handler on the device side.

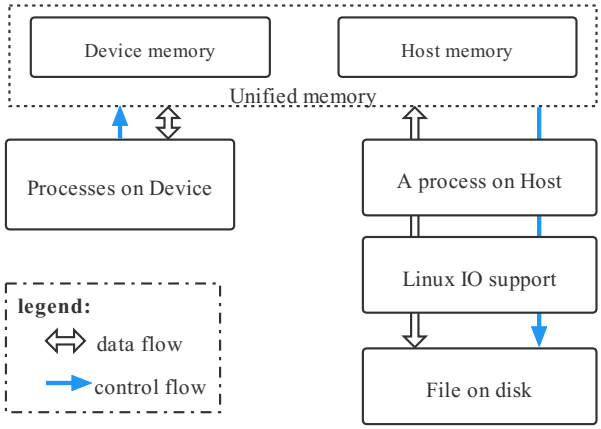

Fig. 1. IO schema

Communication with CPU. The IO operations, need to interact with the underlying file system to handle files. This needs to be ultimately done on the host side due to the lack of a direct path between GPU and disks. GPU sends IO request messages to the CPU. CPU then performs IO on the device's behalf and writes the results back. Device resume execution when the host function finishes.

Linux support. For blocking IO: Normal POSIX IO functions are applied. For Non-blocking IO: There are several possible choices of non-blocking IO support.[4] We pick glibc version of asynchronous IO due to portability.

Single thread to perform file manipulation. All file manipulation operations (to open, close, delete a file) are executed by all processes in a group according to MPI standards. Since they are only needed to be performed once per file, and they are relatively costly, only thread one in the communicator group will actually do the operation while all other threads will wait. The MPI barrier is used to block the idle threads, and thread one broadcasts the result to others to achieve the synchronized return.

File handler in global memory. The MPI_File instances, which is the internal handler of a file, are stored in the CUDA global memory shared by all threads in different blocks. If a block contains multiple threads, they may use the shared memory as a cache, which is only accessible within the block but has a higher accessing speed, to store related data.

4. IO OPTIMIZATION

Basic implementation of MPI_File_read and MPI_File_write is straightforward. It requires frequent communications between GPU, CPU and disk, however. One way to reduce the number of communications is to use the file buffer. In this

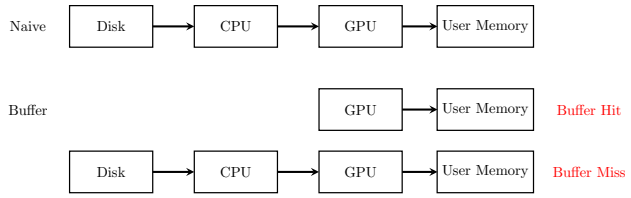

Fig. 2. Dataflow of Basic and Buffer Implementation

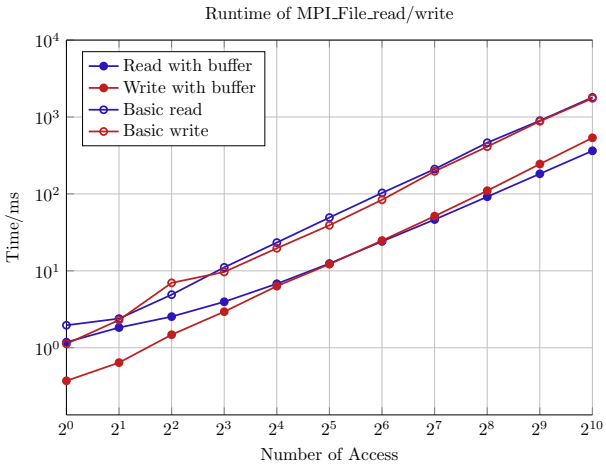

Fig. 3. Runtime of basic version and buffer version of MPI_File_read and MPI_File_write, under different number of accesses. Block size = 2048 bytes, #Process = 1, #Block = 1.

section, we come up with a version of read and write, which "caches" the file as buffer blocks, and further improve it by using finer lock and private buffer, finetuning block size and optimizing with memory allocate mechanism.

Buffer blocks. By treating each file as a sequence of blocks, a pointer array is initially allocated, which will point to the buffer of each file block. When the user calls MPI_File_read or MPI_File_write, the program firstly checks whether this causes a buffer hit. If it is, directly read/write the buffer; otherwise, do the same as the basic implementation. The data flows of these two implementations are illustrated in figure 2. This brings better efficiency due to the decrease of communication between GPU and CPU.

Figure 3 shows the performance of MPI_File_read and MPI_File_write. We assigned one process to repeatedly access the same block for a various number of times, and measured the total execution time. As can be seen, the buffer implementation becomes stable while the number of access increases. It achieves up to 5x speedup for both read and write.

Finer granularity of lock. Previously, the correctness of concurrent writings is ensured by treating the whole function body of MPI_File_write as the critical region. This

means all write requests are processed serially. In fact, this lock is too coarse and may result in more conflicts than necessary. Writings on different blocks can be solved concurrently. Thus, lock per buffer block is used to replace the original coarse lock. Note that this strategy will not bring deadlock problems, as each write request only acquires one lock at one time, and applies for the lock of another block only after it releases the old lock.

The effectiveness of this strategy is proved by our benchmark. In the benchmark, each process accesses different blocks. As shown in figure 4, the fine lock version works with up to 100x improvement as the number of processes increases.

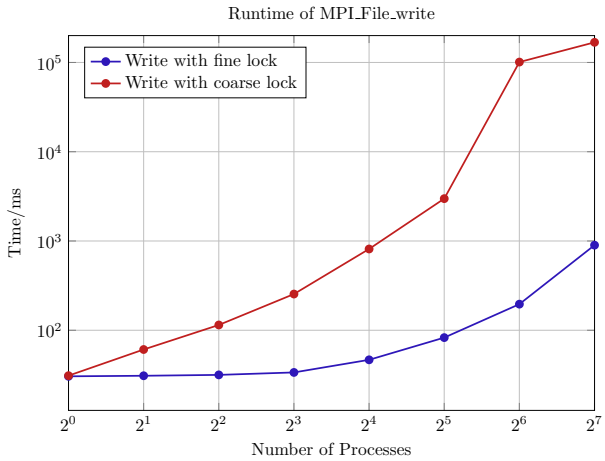

Fig. 4. Runtime of MPI_File_write with fine lock and with coarse lock, under different number of processes. Block size = 2048 bytes, #Writing = 100, each process writes different blocks.

Private buffer. For now, the buffer of a particular file is shared between all processes that open the file collectively. The performance still suffers when multiple processes write data to the same block at the same time, since each of them needs to acquire a lock.

One possible optimization is using a private buffer per process. When a process wants to write a whole block of data, it allocates a private buffer and writes data into it. Then it modifies the corresponding pointer in the pointer array, making it point to the private buffer, such that this data is available to other processes. This optimization reduces the length of the critical region that must be executed sequentially significantly. Note that this optimization only works when writing a whole block of data. Otherwise, it's very hard to maintain sequential consistency while keeping high throughput.

We measured the runtime when multiple processes keep writing data to the same block. As shown in figure 5, using a private buffer in this situation can give up to a 5x speedup.

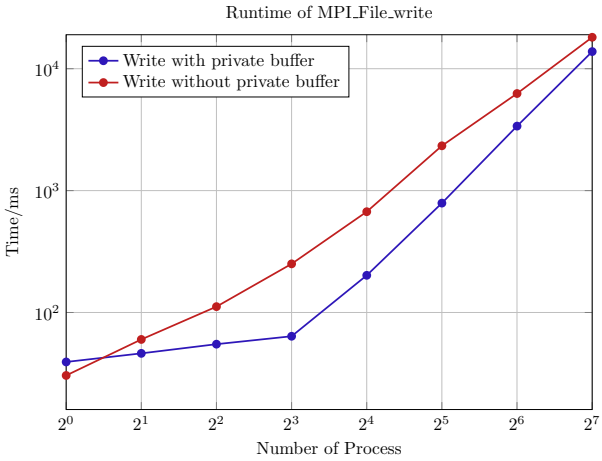

Fig. 5. Runtime of MPI_File_write with private buffer and without private buffer, under different number of processes. Block size = 2048 bytes, #Block = 1, #Writing = 100.

Best block size. Block size is one of the hyperparameters facing trade-offs. If the block size is small, we have to allocate more blocks to support the same amount of data. This puts lots of pressure on our memory management mechanism, especially when the memory allocation mechanism is not efficient enough. At the same time, it wastes lots of memory as each memory allocation needs to store some sort of control structure.

On the other hand, if the block size is large, it's more likely that the data accessed by multiple processes turns out to be in the same block, thus, harms the performance.

As shown in figure 6, we tried different block size and measured the runtime. We found that block size should be at least 2048 bytes to be efficient. As we want to make it as small as possible, we fix the block size to be exact 2048 bytes.

Memory management optimizaiton. We manage the unified memory through a linked-list structure. Initially, as shown in the upper part of figure 7, all available memory spaces are considered as one continuous block. When an internal allocation is invoked, it finds the first *free* block, then split and chain them. One strategy is to allocate the **first** one of the two new blocks, as shown in the middle part of figure 7. This is inefficient as each allocation requires walking through the linked list from the beginning. The time complexity is $O(n)$.

The optimization would be allocating the **second** one of the two new blocks, as shown in the bottom part of figure 7. This time, there is no walking through the link-list until we run out of memory. The time complexity is amortized $O(1)$.

Our benchmark confirms the theory. We ran the program using a different number of processes, each process

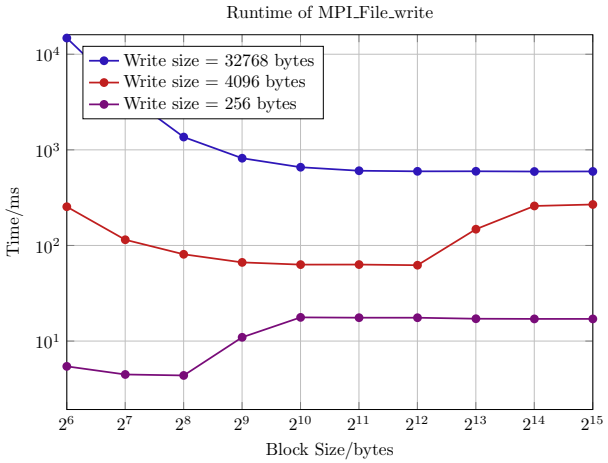

Fig. 6. Runtime of MPI_File_write with different write size, under different block sizes. #Process = 4, #Writing = 100, each process writes a specific region.

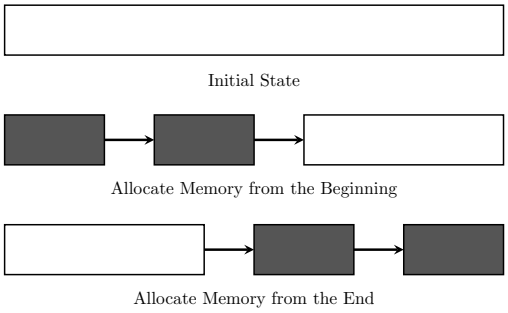

Fig. 7. Memory Allocation Strategy

writes 4096 bytes to different blocks. As shown in figure 8, the basic strategy can be more than 10 times slower than the optimized one if memory allocation is frequently invoked.

5. IO WITH VIEW

Multiple MPI processes may try to access the same file at different places. Instead of asking processes to calculate the target position and seek there explicitly, it is more user-friendly to provide standard APIs that enable users to set templates that define the way a single process views a file and handle the seek-and-access internally. This section starts with the idea of MPI-standard view and introduces the basic implementation of view based on typemap. An improvement will be gone through to show that the overhead is mainly introduced by repeatedly acquiring the write locks and can be reduced by pre-analysis of the typemap. We'll also present a novel algorithmic improvement on the concrete implementation of typemap: to define a tree-shape storage instead of a plain linear one.

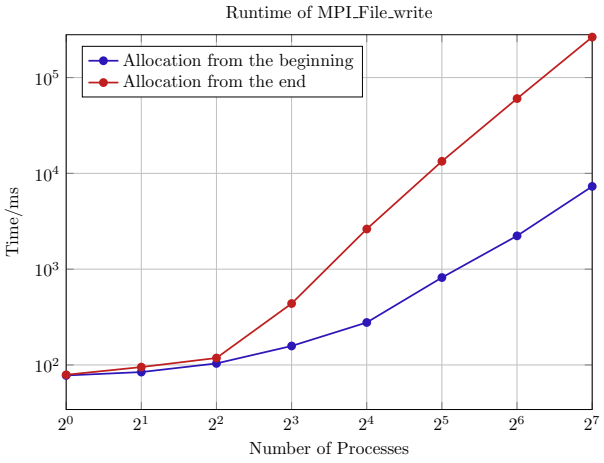

Fig. 8. Runtime of MPI_File_write with optimized allocate and without optimized allocate, under different number of processes. Write size = 4096 bytes, #Writing = 100, each process writes different blocks.

View principle. MPI standard defines a so-called "filetype" to describe the partition of a file among processes. It is a MPI predefined or derived datatype that has all its basic element types the same. Displacements of each element are defined in the filetype's typemap, which stores the elements' types and positions as a list of tuples [1]. Since the displacement of each element is stored explicitly, the partition may not be contiguous and holes may be included.

Fig. 9. View structure and file partition [1].

A MPI standard view includes a displacement that indicates the very first hole, an etype that is the smallest element in which the process views the file, and a filetype. The filetype will be repeatedly referred to as the file accessing template, with a so-called "repetition gap" indicating the final holes in a filetype repetition. Figure 9 shows that, with complementary views set to different processes in a group, the global data distribution can be achieved [1].

Two common access patterns are shown here in Figure 10 and Figure 11, which most following tests and benchmarks are based on. In Figure 10, the filetype is created by extending the typemap of a single basic element contigu-

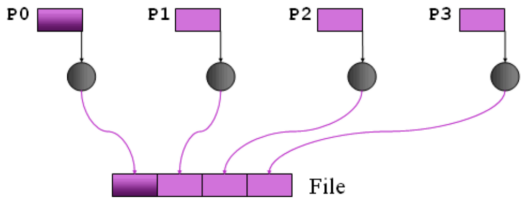

Fig. 10. Access the file according to contiguous filetype [5].

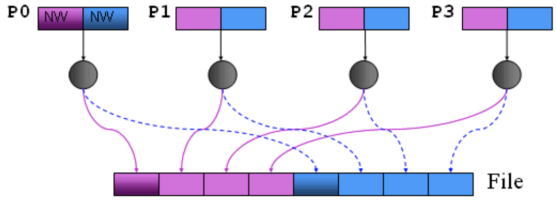

Fig. 11. Access the file according to vector filetype [5].

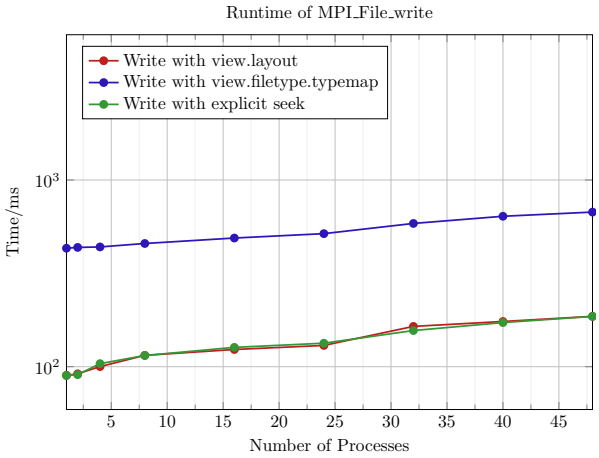

Fig. 12. Runtime of MPI_File_write with contiguous views, under different number of processes. Write size = 4096 bytes, displacement = 4096, contiguous size = 32 MPI_CHAR, #Writing = 100, each process writes different blocks

ously. As a result, the process will see the file contiguously with only the very first hole indicated by the view displacement. The access pattern is the same as that the user explicitly seeks to the displacement position and access the file without view. Figure 11 shows a more complex filetype that is organized as a vector and instructs the process to write a contiguous block, skip a long hole where other processes are supposed to write their block, and then continue to write another block, repeatedly. The same result can be achieved if the user gets the numbers of the process in the group, calculate the gap, and repeatedly seek the new block start in order to write the block.

IO based on typemap. Since the filetype itself maintains a typemap for elements positions, a native way to access the file is element by element, i.e. seek to the position indicated by the typemap and access one element, then seek to the next position.

The seek positions of different processes in the group, kept in the array stored together with the file handler in MPI_File instance, remains the real file position, so that the translation overheads will not be introduced to the r/w procedures. A modification is applied to the seek and get_position operations to translate between the real file pointer and the view position. Overheads are introduced when those two operations are called externally. However, since the r/w operations keep their own seek pointer internally in the function and only do seek after access is done, there are few overheads while accessing the file.

Typemap pre-analysis. Though accessing the file based on the typemap provided by filetype seems pretty straightforward, benchmark tests show that it does cause a big overhead when writing the file. The reason is that every time a process wants to write to the buffer, which is divided into multiple small blocks, it needs to obtain the lock for the corresponding block to maintain the r/w consistency. If the process is writing element by element, it needs to obtain and release the lock of a block multiple times.

A solution is to pre-analyze the typemap, i.e. gathering all contiguous elements into sections and writing the buffer section by section. This design works because the typemap of filetype is defined for file access specifically and thus having the properties. It has all the elements being the same type, so that the contiguity is possible to be easily justified. It is also of high possibility that a large part of the typemap is contiguous so that the elements can be gathered by only a small number of sections. In this case, the number of times for a process to acquire a lock is largely reduced.

The benchmark results shown in Figure 12 and Figure 13 supports our expectations. In Figure 12, it can be easily pointed out that for simple writes such as writing contiguously, the speed for writing with typemap-pre-analyzed view keeps almost the same efficiency as writing without view, while writing directly based on typemap has a great overhead. This supports our assumption that reducing the times the process acquiring the lock can highly reduce the r/w overhead. In Figure 13, the processes are writing according to a complex view shown in Figure 11. The tests suggest that typemap-pre-analysis way of writing still keeps the efficiency high. However, with a template full of holes, writing without view and with explicit seeks becomes relatively slow.

Implementation of the typemap, and tree-shape compact storage. As is stated above, the seek and get position operations need to translate between the real file pointer and

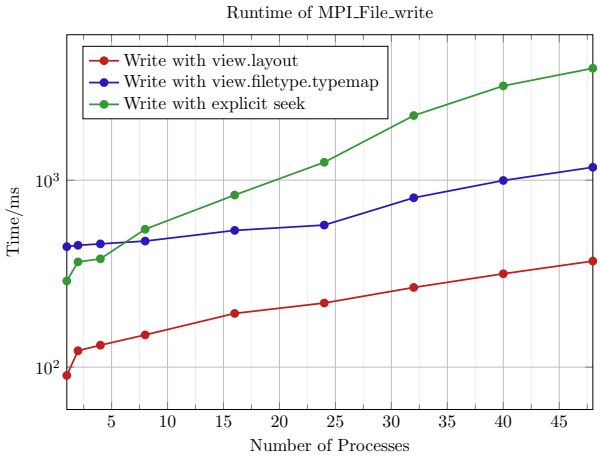

Fig. 13. Runtime of MPI_File_write with vector views, under different number of processes. Write size = 4096 bytes, displacement = 32*rank, block size = 32 MPI_CHAR, #Writing = 100, each process writes different blocks

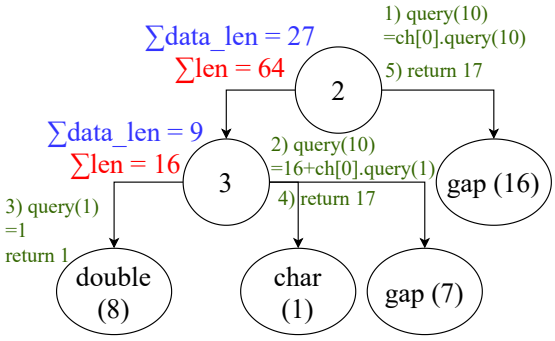

Fig. 14. An example tree-shape typemap: {(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40), (double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104)} (format: (type, position)). An example of doing translation is also shown: the user queries the 10th byte in the view, and the tree finally returns the 17th byte in the file.

the virtual position in view, based on the typemap. Straightforwardly, the typemap can be implemented in a linear way with linear space and time complexity. However, the two common access patterns (contiguous filetype and vector filetype, as discussed above in Figure 10 and 11) indicate that there must exist a lot of duplication in the typemap, so we can compact it by introducing some form of multiplicity. Also, note that the translation process is quite like querying the k^{th} largest element in a sorted data structure (say, a search tree). Consequently, we implemented a novel tree-form typemap storage, supporting the translation process in $O(tree\_height)$ time and a space complexity of $O(n/multiplicity)$.

To be more specific, we compact the typemap by introducing a multiplicity field in our typemap and thus merging the duplicating elements. And we treat the translation process like querying the k^{th} largest element in a sorted data structure (say, a search tree), where the view's virtual position corresponds to the k, and the real file pointer corresponds to the element under query.

Therefore, we can define a tree-form typemap storage: a non-leaf node contains the multiplicity of the typemaps represented by its subtrees; each subtree is also a tree-shape typemap; a leaf node contains either a basic type element or a gap. This is also a search tree, because the order in the real file corresponds to the order in the subtree (from left to right). An example is shown in Fig 14; the "3" (and its subtrees) means: a double followed by a char and then followed by 7 bytes of gap, and then 2 duplicates of that. We have also implemented the translation process analogously to a query on a search tree.

The space complexity is $O(n/root\_multiplicity)$. Compared to the basic linear storage with $O(n)$ space, this saves much when the user creates many instances of custom datatypes. And the time complexity is $O(tree\_height)$, no worse than the linear version with $O(n)$ time.

6. CONCLUSIONS

Our implementation for CUDA MPI in IO and datatypes fulfill many users' needs for GPU IO, including blocking and non-blocking IO, IO with different thread views, and user-defined datatypes. We implemented GPU-side IO buffer, along with several concurrency optimizations on it, to enable multi-process highly-efficient simultaneous access. We extended the basic file access to the view version to enable file partition with simple APIs, while still keeping the access efficiency. A novel tree-shape compact storage of typemap is proposed as well, which could be meaningful when MPI standard extends to support multiple etypes in the future. Relevant benchmarking results show the effectiveness of our implementations.

7. REFERENCES

[1] Message Passing Interface Forum, "Mpi: A message-passing interface standard version 3.1," online:https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[2] Mark Harris, "Unified memory for cuda beginners," online:https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[3] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek, "Cuda, release: 11.6," online:https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmlum-unified-memory-programming-hd.

[4] Shuveb Hussain, "Asynchronous programming under linux," online:https://unixism.net/loti/async_intro.html.

[5] Cornell Virtual Workshop, "Parallel i/o: File view examples," online:https://cvw.cac.cornell.edu/parallelio/fileviewex.