# Optimizing Data Movement for GPU-Based In-Situ Workflow Using GPUDirect RDMA

Bo Zhang[1], Philip E Davis[1], Nicolas Morales[2], Zhao Zhang[3], Keita Teranishi[4], and Manish Parashar[1]

[1] Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, Utah 84112, USA
{bozhang, philip.davis, parashar}@sci.utah.edu
[2] Sandia National Laboratories, Livermore, California 94551, USA
nmmoral@sandia.gov
[3] Texas Advanced Computing Center, Austin, Texas 78758, USA
zzhang@tacc.utexas.edu
[4] Oak Ridge National Laboratory, Oak Ridge, Tennessee 37830, USA
teranishik@ornl.gov

**Abstract.** The extreme-scale computing landscape is increasingly dominated by GPU-accelerated systems. At the same time, in-situ workflows that employ memory-to-memory inter-application data exchanges have emerged as an effective approach for leveraging these extreme-scale systems. In the case of GPUs, GPUDirect RDMA enables third-party devices, such as network interface cards, to access GPU memory directly and has been adopted for intra-application communications across GPUs. In this paper, we present an interoperable framework for GPU-based insitu workflows that optimizes data movement using GPUDirect RDMA. Specifically, we analyze the characteristics of the possible data movement pathways between GPUs from an in-situ workflow perspective, and design a strategy that maximizes throughput. Furthermore, we implement this approach as an extension of the DataSpaces data staging service, and experimentally evaluate its performance and scalability on a current leadership GPU cluster. The performance results show that the proposed design reduces data-movement time by up to 53% and 40% for the sender and receiver, respectively, and maintains excellent scalability for up to 256 GPUs.

**Keywords:** In-Situ · Workflow · GPU · GPUDirect RDMA · Extreme-Scale Data Management.

## 1 Introduction

Emerging HPC systems have widely adopted Graphic Processing Units (GPUs) for their massive computing capability and high power efficiency. As of November 2022, seven of the top ten systems on the TOP500 list [23] have GPUs. Scientific simulations and analyses benefit from both the parallelism and energy efficiency of GPUs' architecture [12]. A variety of applications and tools, such as LAMMPS [9] and ZFP [16], have released GPU-optimized versions.

However, a vast I/O gap still remains for the loosely-coupled in-situ workflow [15], which typically consists of several scientific applications as its components. Within in-situ workflows, the component applications run on GPUs/CPUs and exchange data through a high-speed network. Although individual applications can leverage GPUs, the inter-GPU and GPU-CPU I/O cross codes are implemented in an ad hoc manner, which is prone to suboptimal performance. On the other hand, the latest hardware-specific technique, i.e., GPUDirect RDMA (GDR) [3], is available on many modern HPC systems and offers a performance improvement opportunity, while requiring deep hardware knowledge and low-level programming skills from domain scientists.

Existing solutions to the I/O across components in the workflows view the devices (GPU) and hosts (CPU) as individual entities and employ a sequential $device \rightleftharpoons host \rightleftharpoons network$ pathway. As can be seen, the involvement of the hosts is nonessential, and it slows down the I/O performance due to unnecessary data movement to/from the hosts. This slowdown will be exacerbated at larger scales. In addition, involving hosts during I/O across components requires the developers transfer data between hosts and devices with low-level GPU programming APIs, such as CUDA, HIP, etc. It is nontrivial for domain scientists to program with these low-level APIs, and such a programming approach often results in ad hoc solutions that are limited in both interoperability and portability, especially in cases of massive variables or complex I/O patterns.

Porting existing in-situ workflow to GPUs is an ongoing effort in many scientific computing communities. Figure 1 illustrates the challenges of this workflow porting problem: Some of the components have already been ready to run on GPUs, whereas others are in the porting process or still left as legacies. This heterogeneity complicates the I/O management between components in different porting stages and thus makes the plug-n-play almost impossible. Complex data communication patterns and a great number of variables make the situation even worse. For example, the I/O engine of MURaM workflow [19] contains seven separate procedures with 50 1-D variables, 63 2-D variables, and 34 3-D variables in total. We realized that although moving data between GPU and the host plus conventional communications between host buffers and the network remains a solution, it requires greatly repetitive code refactoring efforts during the porting process but still gains no flexibility of being ported to other

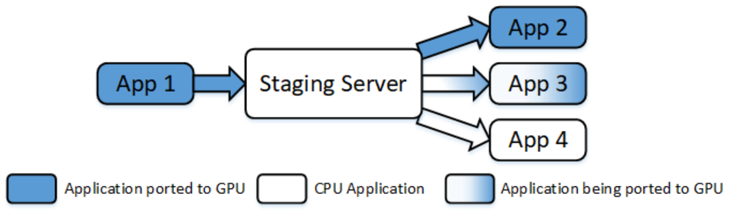


**Fig. 1.** A typical ongoing GPU-based in-situ workflow porting process. The main simulation has already been ported to GPU. Other components are ported, being ported, or still remain the CPU version. The I/O between components also has to change according to the data source/destination.

GPU ecosystems. In addition, we observed that the I/O performance degraded severely due to consecutive staging at the host buffer, which can be bypassed to reduce the I/O overhead for most cases.

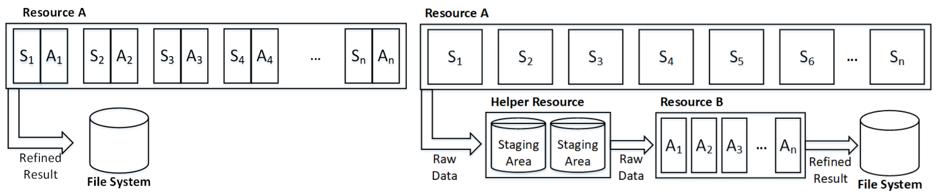
Based on these insights, we investigate several designs for inter-application bulk communications from/to GPUs and introduce a GDR design that circumvent unnecessary data movement path through the host. We therefore propose the first interoperable I/O abstraction for GPU-based in-situ workflow and implement it as an extension of the DataSpaces [10] data staging service. We make the following key contributions in this paper:

- We investigate several designs for bulk data exchanges between GPU applications with respect to the features of in-situ workflow, and then propose a GDR design that reduces I/O overhead by circumventing the host.
- We propose the first interoperable I/O abstraction for GPU-based in-situ workflow, implemented as the extension of the DataSpaces data staging service, which reduces the software refactoring cost and enables plug-n-play in the workflow porting process.
- We evaluate the proposed designs on current leadership GPU clusters using both synthetic and real workflows running on up to 256 GPUs and demonstrate that they can reduce up to 53% and 40% of the I/O time for sender and receiver, respectively, in comparison to the baseline.

## 2 Background

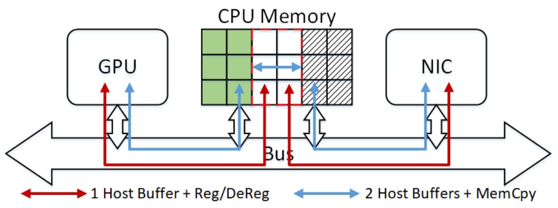
### 2.1 In-Situ Workflow

The traditional scientific workflow model first writes the simulation data to persistent storage, and then reads it back into memory for the analysis or visualization later, which is defined as a *post-hoc* method since it reflects that the visualization or analysis is performed "after the fact" [8]. We have witnessed a significant performance slowdown for this method as the computational throughput scaled up [4,5,7,17]. An alternative approach, which is named by the umbrella term *in-situ*, saves the huge I/O cost by removing the nonessential involvement of persistent storage. Two paradigms have emerged from the in-situ model: *tightly-coupled* method and *loosely-coupled* method [15]. The tightly-coupled paradigm is illustrated in Figure 2a. Simulation and analysis run in the same process using the same set of computing resources. They alternate in each iteration, sharing the data stored in the memory, and finally output the refined result to the file system. As for the loosely-coupled paradigm, simulation and analysis run asynchronously in the separate process groups on their dedicated resources, as shown

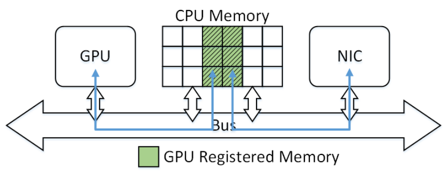

Resource A
$S_1$ $A_1$ $S_2$ $A_2$ $S_3$ $A_3$ $S_4$ $A_4$ ... $S_n$ $A_n$
Refined Result
File System
Resource A
$S_1$ $S_2$ $S_3$ $S_4$ $S_5$ $S_6$ ... $S_n$
Helper Resource
Staging Area
Staging Area
Raw Data
Raw Data
Resource B
$A_1$ $A_2$ $A_3$ ... $A_n$
Refined Result
File System

(a) Tightly-coupled in-situ workflow. (b) Loosely-coupled in-situ workflow.

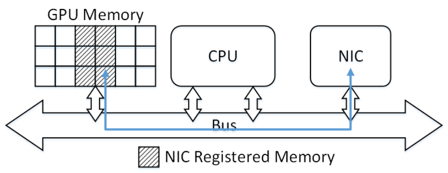
**Fig. 2.** A schematic illustration of in-situ workflow paradigms.



(a) Vanilla data flow between GPU and NIC. The red path requires the data to be staged in only one host buffer but needs a sequential DMA deregistration and registration by GPU/NIC or NIC/GPU, respectively. The blue path allows GPU and NIC to register at two host buffers, but an extra memory copy is needed.



(b) GPUDirect data flow between GPU and NIC. GPU and NIC and register at the same host buffer at the same time.



(c) GPUDirect RDMA (GDR) data flow between GPU and NIC. CPU data staging is avoided.

**Fig. 3.** Data flow paths between GPU and network interface card (NIC).

in Figure 2b. They exchange the shared data over the high-speed network with the help of a staging server, which takes extra resources to manage the data forwarding.

The loosely-coupled in-situ workflow maintains its flexibility and modularity by isolating the computational tasks at an appropriate granularity. We define each isolated computational task that runs separately as an individual *component* in the context of loosely-coupled in-situ workflow. Then, the flexibility means that the running scale of each component can be configured individually according to its characteristics, avoiding the inefficiency under the holistic resources allocation. Besides, the modularity supports easy plugin-and-play for new components to join the workflow, which saves the significant development cost and enables more complicated extensions. Both features offer great improvement opportunities for in-situ workflows by leveraging the GPUs equipped in the modern HPC systems. Assigning each component to its best-fit hardware will finally improve the overall performance of the workflow.

## 2.2 GPUDirect Technologies

Direct Memory Access (DMA) requires memory registration before data access. The DMA engine of GPU has to register a CPU memory region to enable asynchronous data movement, and the network interface card (NIC) also requires this registration to transfer data over the network. Therefore, as shown in Figure 3a, either GPU and NIC registering the same host buffer sequentially or registering two separate host buffers at the same time but introducing an extra data copy is required for GPU data communication. Both the de-register/register process and the extra memory copy can be summarized as the DMA overhead that increases

I/O latency. NVIDIA GPUDirect technologies eliminate unnecessary memory copies between GPUs and other devices, decrease CPU overheads, and reduce latency [3], thereby significantly enhancing data movement and access for GPUs. Released with CUDA 4.0, the initial GPUDirect enabled both GPU and NIC to register the same memory region on the CPU, avoiding the DMA overhead at the host as shown in Figure 3b. From CUDA 5.0, GPUDirect RDMA (GDR) is released as the extension of GPUDirect, which supports GPU memory registration by any third-party standard PCIe device. Figure 3c illustrates the direct data exchange path between GPU and NIC.

AMD GPUs also support this peer-direct technique in their ROCm ecosystem, namely ROCmRDMA [2]. In this work, however, we use the umbrella term *GPUDirect RDMA (GDR)* to refer to all direct data exchange solutions between GPU and NIC. We focus on NVIDIA GPUs with the CUDA programming ecosystem and the RDMA-enabled network in the rest of the paper.

## 3 Related Work

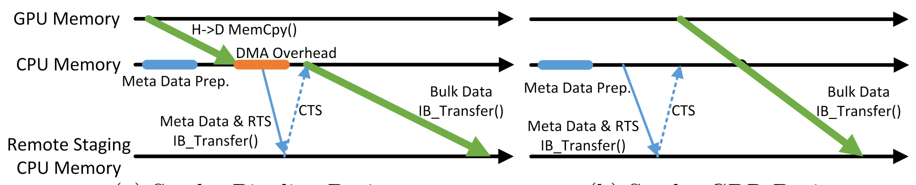
Over last ten years, a fair amount of contributions from HPC community have been made to accelerate GPU-related I/O in widely used programming models and network substrates by GPUDirect technologies. Wang et al. proposed MVAPICH2-GPU [26], which is the first GPU-aware MPI implementation with the GPUDirect optimization for CUDA-based GPUs. Potluri et al. upgraded the GPU-aware MPI libraries using GPUDirect RDMA (GDR) and proposed a hybrid solution that benefits from the best of both GDR and host-assisted GPU communication [18]. Shi et al. designed GDRCopy [22], a low-latency copy mechanism between GPU and host memory based on GDR, which improved the efficiency of small message transfer. NVIDIA NCCL [13], as a popular backend for leading deep learning frameworks, also supported GDR in its communication routine set. In addition, programming frameworks that simplify the GPU application porting process have been explored as well. Kokkos [24], RAJA [6] and SYCL [20] support compile-time platform specification for applications written in their abstractions. However, research work in either data movement optimization or I/O abstraction from a workflow perspective is extremely limited. ADIOS2 [11], a high-performance I/O framework that often plays as a data coupler between components in a workflow, is extended to support GPU-aware I/O [1]. However, its GPU I/O support works only for binary-pack version 4 (BP4) and BP5 file engines, which are still solutions based on persistent storage. Zhang et al. explored the data layout mismatch in the CPU-GPU hybrid loosely-coupled in-situ workflow and proposed a solution to minimize the data reorganization overhead [27]. However, they did not optimize the data movement pathway for GPU components. Wang et al. presented a conceptual overview of the GPU-aware data exchange in an in-situ workflow [25], but they proposed only a preliminary idea with neither implementation details nor quantitative evaluation at scale. Our work is distinguished from related efforts in being the first interoperable GPU-aware I/O abstraction for the inter-component data exchange in loosely-coupled in-situ workflows. We provide a systematically comprehensive evaluation of the GDR design at the largest scales we were able to reach on a state-of-the-art production HPC system equipped with GPUs.

## 4 Design

In this section, we discuss the baseline and optimized designs for the inter-application bulk data movement from/to GPUs. As shown in Figure 1, components in the workflow can be generally classified in three categories as staging server, sender, and receiver. The staging server typically works as a memory-bounded component that is responsible for storing the intermediate shared data and processing the asynchronous I/O requests made by all other applications. Therefore, even if the staging server may also run on nodes equipped with GPUs, CPU main memory is still chosen as the primary storage media for its consideration of capacity and cost. The sender is typically a simulation that produces multidimensional data on GPUs and sends it out to the staging server. The receiver is usually an analysis or a visualization that fetches the data from the staging server and consumes it. A loosely-coupled in-situ workflow has only one staging server component and at least one sender and one receiver. In the following subsections, we discuss these two parts, respectively. We also present the implementation overview of the proposed I/O framework and demonstrate its interoperability through a code snippet.

### 4.1 Sender Side

**Baseline Design** To send the GPU data out to the staging server, the baseline design simply uses CUDA memory copy from device to host and then sets up a conventional CPU-CPU bulk data transfer between the sender and staging server. This straightforward approach takes the bulk I/O as an ensemble by concatenating the GPU to CPU and CPU to the staging server data transfer together, which runs sequentially after the meta-data preparation phase on the CPU. The DMA control sequence introduced in Section 2.2 must be gone through between the two concatenated I/O procedures, which increases the overall latency. Although this design is intuitive and simple to implement, its weakness becomes apparent when frequent and consecutive `put` requests are made because the fixed overhead is incurred for every request.



(a) Sender Pipeline Design. (b) Sender GDR Design.

**Fig. 4.** A schematic illustration of sender side designs.

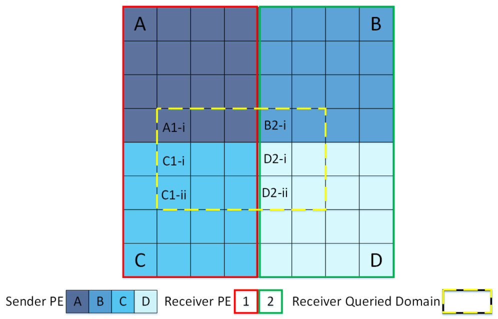


**Fig. 5.** Data Object Reassembly on the receiver side. 4 PEs of the sender put local 4x4 2D arrays into a global 8x8 domain. 2 PEs of the receiver expect to get a subset of the 8x8 shared domain, a 3x3 array and a 3x2 array, respectively. Receiver PE1 has to find the data object A, copy the memory line once, and find data object C, copy the memory line twice. Receiver PE2 has to find the data objects B, D and copy the memory line accordingly.

**Pipeline Design** We are able to optimize the baseline design by overlapping independent tasks after splitting the entire send procedure into several stages and analyzing their dependencies carefully. Figure 4a illustrates the pipeline design that requires no additional prerequisites. The meta-data are prepared on the host when the bulk data is copied from GPU to the host. Also, the DMA operations partially overlap with the connection setup between the sender and the staging server. This design exploits the potential overlaps between different stages of the send procedure by leveraging the asynchrony on both the GPU and CPU.

**GDR Design** As long as the GPU data are transferred to the staging server, the less intermediate stages result in better performance. We fully circumvent the host involvement by employing GDR in the bulk data transfer. Figure 4b presents the neat GDR design. After the essential meta-data preparation and connection setup phase, data are directly sent from GPU memory to the staging server. No memory allocations and DMA overhead are incurred in this design.

### 4.2 Receiver Side

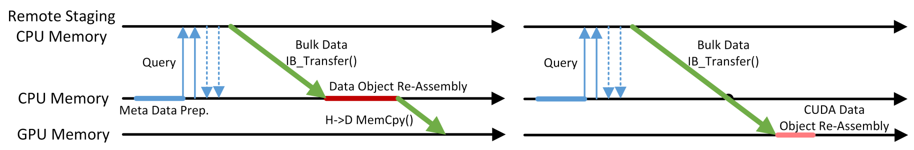
Due to the simplicity of the sender design, on the receiver side, a data object reassembly stage is introduced in addition to the bulk data I/O for the scale flexibility mentioned in Section 2.1 and the random access to the multidimensional data specified by a geometric descriptor. We discuss these two stages separately in this subsection.

**Data Object Reassembly** Every `get` request on the receiver side has to go through the data reassembly stage before delivering the queried data to users. Figure 5 demonstrates the necessity of data reassembly by an example. A 2-D data domain is shared by two applications served as a sender with four processing elements(PEs) and a receiver with two PEs, respectively. The sender puts four data objects into the staging server, while the receiver expects to get a subset of data in each of two PEs. Therefore, each PE in the receiver has to figure out the original data objects, extract each subset, and finally reassemble the subset to a contiguous data object accordingly. Even if the receiver query the entire domain, reassembling the original data objects to the queried contiguous data object is still essential as long as the two applications are running at different scales.

**Algorithm 1** CUDA Data Object Reassembly Kernel

```
Input: $src\_obj$, $dst\_obj.bbox${bounding box descriptor}
Output: $dst\_obj.data$
  $its\_bbox \leftarrow$ Intersection($src\_obj.bbox$, $dst\_obj.bbox$)
  $src\_nx$, $src\_ny$, $src\_nz \leftarrow$ Distance($src\_obj.bbox$)
  $dst\_nx$, $dst\_ny$, $dst\_nz \leftarrow$ Distance($dst\_obj.bbox$)
  $sub\_nx$, $sub\_ny$, $sub\_nz \leftarrow$ Distance($its\_bbox$)
  $i \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
  $j \leftarrow blockIdx.y * blockDim.y + threadIdx.y$
  $k \leftarrow blockIdx.z * blockDim.z + threadIdx.z$
  if $i < sub\_nx$ and $j < sub\_ny$ and $k < sub\_nz$ then
      $dst\_idx \leftarrow i + j * dst\_nx + k * dst\_nx * dst\_ny$
      $src\_idx \leftarrow i + j * src\_nx + k * src\_nx * src\_ny$
      $dst\_obj.data[dst\_idx] \leftarrow src\_obj.data[src\_idx]$
  end if
```

The existing solution for data object reassembly is based purely on CPU. It iteratively calls `Memcpy()`, which moves a data line along the lowest dimension at once, for the multidimensional data. Since the data destination is ported to GPU memory in the GPU applications, we design a CUDA kernel to accelerate this data object reassembly task by utilizing the intrinsic parallelism of GPU architecture. Algorithm 1 describes the details of the kernel. Although an individual kernel supports only up to 3-D data object reassembly, it can be iteratively launched several times for data in more dimensions. Asynchronous kernel launch is utilized for concurrency depending on the capability of the target CUDA device.



(a) Receiver Baseline Design. (b) Receiver GDR Design.

**Fig. 6.** A schematic illustration of receiver side designs.

**Bulk Data Transfer** The bulk data I/O path on the receiver side keeps the same options as the sender side: CPU-CPU transfer plus CUDA memory copy from host to device or GDR. Therefore, we propose three receiver designs as the combination of bulk data transfer and data object reassembly options. Figure 6a illustrates the baseline design that reassembles the received data objects on the host to a new CPU buffer, and then transfers it to the GPU destination. Because

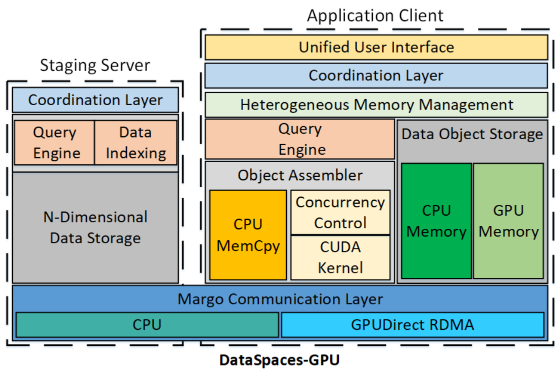


**Fig. 7.** Architecture of DataSpace-GPU. Existing modules are extended to support both GPU computation and storage under the heterogeneous memory management layer.

two buffers are used on the host and the data object reassembly intrinsically finishes the memory copy between them, no DMA overhead is incurred in this design. The hybrid design takes the conventional I/O path but uses the CUDA kernel for data object reassembly. It holds only one buffer for both receiving data from the staging server and transferring to the GPU, but its DMA overhead partially overlaps with the CUDA kernel computation since multiple data objects are received typically and the following work is done asynchronously. The GDR design keeps clear as shown in Figure 6b. There is no CPU involvement, and the data object reassembly is done by CUDA kernels.

### 4.3 Implementation and Interoperability

Our designs are implemented in the existing DataSpaces staging framework as an extension to support the data exchanges from/to GPU components inside an in-situ workflow. Figure 7 presents a schematic overview of the DataSpaces-GPU. It leverages the existing components by reusing its data transport, indexing, and querying capabilities. The GDR capability of the Margo [21] communication layer is employed by the GPU memory extension of the data object storage module at the application client. The object assembler module adds support to launch the concurrent CUDA kernel when the target GPU is capable. All the GPU extensions are organized by the heterogeneous memory management layer, which determines whether the user data are located on the CPU or GPU. For the purpose of minimizing the software porting cost, we design a set of unified APIs for both CPU and GPU I/O to address the interoperability issue with the legacy CPU workflows. Figure 8 presents a code example that compares the lines of code (LOC) changes for a single variable I/O procedure with or without our framework. After setting up the proper meta-data, only one LOC is needed to send or receive the CPU data. When the data are located on the GPU, we need to calculate the data size, manage the CPU memory buffer, and handle the CUDA memory copy at the sender and receiver side, respectively. Approximately 10 LOC are added on each side for a single variable without any performance opti-

```
/* Meta Data Defination */
  struct meta_data {
    char var_name[128];
    unsigned int version;
    int element_size;
    int ndim;
    uint64_t* lb; // upper bound
    uint64_t* ub; // lower bound
  };

  struct meta_data meta;
  void* host_data, device_data;

/* Meta Data Preparation */
  prepare(meta);
```

```
/* I/O procedure call for GPU data
      without DataSpaces-GPU */
  size_t data_size = meta.element_size;
  for(int i=0; i<meta.ndim; i++) {
    data_size *= meta.ub[i] - meta.lb[i] +1;
  }
  void* host_buffer = (void*) malloc(data_size);
// Sender Side
  cudaMemcpy(host_buffer, device_data, data_size,
          cudaMemcpyDeviceToHost);
  dspaces_put(meta, host_buffer);
// Receiver Side
  dspaces_get(meta, host_buffer);
  cudaMemcpy(device_data, host_buffer, data_size,
          cudaMemcpyHostToDevice);
  free(host_buffer);
```

```
/* I/O procedure call for
    CPU data */
// Sender Side
  dspaces_put(meta, host_data);
// Receiver Side
  dspaces_get(meta, host_data);
```

```
/* I/O procedure call for GPU data
      with DataSpaces-GPU */
// Sender Side
  dspaces_put(meta, device_data);
// Receiver Side
  dspaces_get(meta, device_data);
```

**Fig. 8.** Code example of porting a single variable I/O procedure to GPU with or without DataSpaces-GPU.

mization. However, with DataSpaces-GPU, the only effort that needs to be made is changing the CPU pointer to the GPU pointer and no extra code is required, which saves great software porting costs, especially when many variables are communicated or the communication pattern is complex. DataSpaces-GPU thus enables procedure-wise I/O plug-n-play in the entire workflow porting process.

## 5 Evaluation

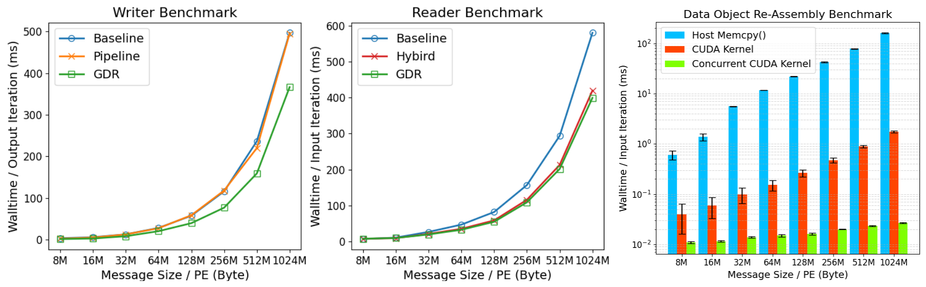
In this section, we present an evaluation of the proposed GDR design compared to the conventional host-based designs in terms of both time-to-solution and scalability. The end-to-end benchmark is tested using a synthetic workflow emulator, and the weak scaling experiment is performed on a real scientific workflow that consists of LULESH-CUDA [14] and ZFP-CUDA.

Our synthetic workflow emulator uses two application codes, namely writers and readers, to simulate the inter-application data movement behaviors in real loosely-coupled in-situ workflows. Writers produce simulation data and send it to the staging servers, whereas readers fetch the data from staging servers and then perform some analysis. In our real workflow experiment, LULESH is the simulation that writes the data out and ZFP is the reader. The data are organized in a 3-D Cartesian grid format with $X \times Y \times Z$ scale in both workflows.

All the experiments were performed on the Phase 2 GPU nodes of the Perlmutter supercomputer at National Energy Research Scientific Computing (NERSC). Phase 2 GPU nodes have a single socket of an AMD EPYC 7763 (Milan) 64-core processor with 160GB of DDR4 RAM. Each node equips four NVIDIA Ampere A100 GPUs with four Slingshot-11 Cassini NICs. All the nodes run libfabric-1.15.0 with Cray Slingshot-11 `cxi` support. All four NICs are leveraged and evenly mapped to the PEs on each node. Concurrent CUDA kernel

**Table 1.** Experimental setup configurations for end-to-end benchmark

| | |
|---|---|
| No. of Parallel Writer Cores / GPUs / Nodes | 128 / 64 / 16 |
| No. of Parallel Reader Cores / GPUs / Nodes | 128 / 64 / 16 |
| No. of Staging Cores / Nodes | 32 / 8 |
| Total I/O Iterations | 10 |
| I/O Iteration Frequency | Every 2 seconds |



(a) I/O time (b) Data object reassembly time

**Fig. 9.** Performance comparison per I/O iteration among proposed designs with increasing message size.

launch is enabled, and the maximum number of concurrent kernel launch is set to 32 as the default. In subsequent sections, all measured times refer to the wall time of the blocking I/O routine that guarantees the message is sent to the destination. All the test runs have been executed three times, and the average result is reported.

## 5.1 End-to-end benchmark

This experiment compares the I/O performance between applications and staging servers for various message sizes using different designs introduced in Section 4. Table 1 details the setup for all test cases in this experiment. In order to alleviate the iterative interference, we set the I/O frequency to 2 seconds. The message size we choose to evaluate starts from 8MB, which is the smallest data size for a single variable in each parallel PEs in a typical fine-grained domain decomposition.

Figure 9a presents the benchmark result for writers and readers, respectively. In general, although GDR is designed to optimize small and frequent communication to/from GPUs, it achieves better performance in bulk data transfer than other host-involved designs as well. On the writer side, the baseline method and pipeline method show almost the same trend, which means the overhead of sending metadata is negligible in the bulk data movement. Compared to the baseline and pipeline methods, the GDR method reduces 53% of the put time for the 8MB bulk transfer while still maintaining a 34% of reduction when sending 1024MB messages.

On the reader side, the hybrid and GDR methods achieve up to 28% and 33% reduction of the get time compared to the baseline. The GDR method always performs slightly better than the hybrid method since it avoids the DMA overhead introduced in Section 2.2. Both methods use the CUDA kernel for the

**Table 2.** experimental setup configurations of data domain, core-allocations and size of the staged data for shock hydrodynamics workflow

| Data Domain | 512 × 512 × 512 | 768 × 768 × 768 | 1024 × 1024 × 1024 | 1280 × 1280 × 1280 | 1536 × 1536 × 1536 |
|---|---|---|---|---|---|
| No. of LULESH-CUDA Cores / GPUs / Nodes | 8 / 8 / 2 | 27 / 27 / 7 | 64 / 64 / 16 | 128 / 128 / 32 | 256 / 256 / 64 |
| No. of ZFP-CUDA Cores / GPUs / Nodes | 8 / 8 / 2 | 27 / 27 / 7 | 64 / 64 / 16 | 128 / 128 / 32 | 256 / 256 / 64 |
| No. of Staging Cores / Nodes | 4 / 1 | 16 / 4 | 32 / 8 | 64 / 16 | 128 /32 |
| Total Staged Data Size (3 variables, 10 I/O Iterations) | 30 GB | 60 GB | 120GB | 240GB | 480GB |
| I/O Iteration Frequency | Every 100 computing iteration | | | | |

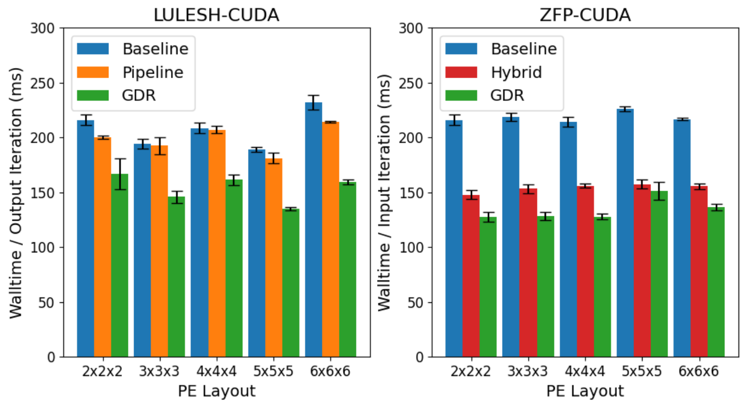


**Fig. 10.** Weak scaling comparison of I/O time per I/O iteration among proposed designs in the LULESH workflow.

data object reassembly instead of the host `Memcpy()` function, which contributes mainly to the performance improvement. Figure 9b extracts and compares the data object reassembly performance from the overall I/O time. The CUDA kernel accelerates the the data object reassembly task by up to 90x as the message size increases. By utilizing the asynchronous kernel execution feature of CUDA devices, launching the kernels concurrently with a barrier that waits all kernels to complete can even achieve an acceleration up to 6000x.

## 5.2 Real Scientific Workflow

In addition to evaluations based on the synthetic workflow emulator, we also apply our proposed designs to a CUDA-based shock hydrodynamics simulation workflow. We use the LULESH-CUDA component for the simulation purpose, which generates 3-D data and sends them to the staging servers. For the analysis, the ZFP-CUDA component gets the data from the staging servers, compresses and writes it to the persistent storage. We select three scalar data fields (energy, pressure, mass) from 13 variables to perform the inter-application data exchange. The experimental configurations of our hydrodynamics workflow tests are listed in Table 2. Since LULESH supports only cubic PEs increment, our evaluation was performed with 8, 27, 64 cores, with a $1:1$ mapping to GPUs and a $\sim 4:1$ mapping to nodes. The grid domain sizes were chosen such that each core was assigned a spatial local domain of size $256 \times 256 \times 256$. We keep this same data volume per LULESH/ZFP core to perform a weak scaling test in this evaluation.

Figure 5.1 compares the proposed designs in the weak scaling workflow with a fixed ratio of LULESH/ZFP resources to the staging server. The GDR design still takes $\sim 24\%$ less time to consecutively send the data fields out compared to others, while performing ∼40% and ∼17% better in fetching the data fields than the baseline and hybrid design, respectively. The I/O time remains relatively constant for all designs as the overall problem size and total resources increase. Little overhead is introduced as the amount of resources increases, which indicates that all proposed designs maintain great scalability to solve the problem in a larger scale.

From our synthetic and real scientific workflow evaluations, we can infer that the straightforward baseline design of data movement between GPU applications performs poorly at any scale due to the sequential $device \rightleftharpoons host \rightleftharpoons network$ pathway. Pipelining I/O design on the sender side and applying CUDA kernels for data object reassembly on the receiver side improves the performance, but nonessential host involvement remains. In contrast, our GDR design enables direct data movement between GPU memory and the RDMA-enabled network, which reduces up to 53% of the I/O time compared to the baseline. In addition, our I/O abstraction for the GPU-based in-situ workflow shares the same API with the conventional CPU-based workflow, which minimizes the software porting effort. In summary, our GDR I/O optimization can effectively reduce the overhead of data exchanges between GPU components in the scientific workflow, while maintaining the interoperability with legacy CPU-based applications.

## 6 Conclusion and Future Work

GPUDirect RDMA has emerged as an effective optimization for inter-node communications from/to GPUs, but it has been adopted only by I/O substrate designed for individual applications. In this paper, we present a novel design that applies GPUDirect RDMA to the bulk data movement between GPU applications within a workflow. Also, we propose the first interoperable I/O abstraction for GPU-based in-situ workflows, which simplifies the GPU workflow porting process and enables procedure-wise plug-n-play through the unified interface. We implemented the proposed solution based on the DataSpaces framework and evaluated it on the NERSC Perlmutter system. Our experimental results, using both synthetic and real GPU workflows, demonstrate that the proposed solution yields an I/O improvement of up to 53% and 40% for sender and receiver, respectively, while maintaining great scalability for up to 256 processing elements on 256 GPUs. As future work, we plan to investigate the performance portability of our design on other network hardware, such as Mellanox EDR and HDR interconnect. We also plan to provide comprehensive support to AMD GPUs in our workflow I/O abstraction.

**Acknowledgements** This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of

Energy's National Nuclear Security Administration (NNSA) under contract DE-NA0003525. This work was funded by NNSA's Advanced Simulation and Computing (ASC) Program. This manuscript has been authored by UT-Battelle LLC under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). This work is also based upon work by the RAPIDS2 Institute supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research through the Advanced Computing (SciDAC) program under Award Number DE-SC0023130. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

## References

1. ADIOS 2 Documentation (2022), `https://adios2.readthedocs.io/en/latest/advanced/gpu_aware.html`
2. AMD ROCm Information Portal - v4.5 (2022), `https://rocmdocs.amd.com/en/latest/Remote_Device_Programming/Remote-Device-Programming.html`
3. NVIDIA GPUDirect RDMA Documentation (2022), `https://docs.nvidia.com/cuda/gpudirect-rdma/index.html`
4. Ahrens, J., Rhyne, T.M.: Increasing scientific data insights about exascale class simulations under power and storage constraints. IEEE Computer Graphics and Applications **35**(2), 8–11 (2015)
5. Asch, M., Moore, T., Badia, R., Beck, M., Beckman, P., Bidot, T., Bodin, F., Cappello, F., Choudhary, A., de Supinski, B., et al.: Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry. The International Journal of High Performance Computing Applications **32**(4), 435–479 (2018)
6. Beckingsale, D.A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A.J., Pearce, O., Robinson, P., Ryujin, B.S., Scogland, T.R.: RAJA: Portable performance for large-scale scientific applications. In: 2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc). pp. 71–81. IEEE (2019)
7. Bethel, E.W., Childs, H., Hansen, C.: High performance visualization: Enabling extreme-scale scientific insight. CRC Press (2012)
8. Bethel, E., Bauer, A., Abbasi, H., Ahrens, J., Childs, H., Geveci, B., Klasky, S., Moreland, K., O'Leary, P., Vishwanath, V., et al.: In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report (2021)
9. Brown, W.M.: GPU acceleration in LAMMPS. In: LAMMPS User's Workshop and Symposium (2011)
10. Docan, C., Parashar, M., Klasky, S.: Dataspaces: an interaction and coordination framework for coupled simulation workflows. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. pp. 25–36 (2010)
11. Godoy, W.F., Podhorszki, N., Wang, R., Atkins, C., Eisenhauer, G., Gu, J., Davis, P., Choi, J., Germaschewski, K., Huck, K., et al.: Adios 2: The adaptable input output system. a framework for high-performance data management. SoftwareX **12**, 100561 (2020)

12. Goswami, A., Tian, Y., Schwan, K., Zheng, F., Young, J., Wolf, M., Eisenhauer, G., Klasky, S.: Landrush: Rethinking in-situ analysis for gpgpu workflows. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). pp. 32–41. IEEE (2016)
13. Jeaugey, S.: Nccl 2.0. In: GPU Technology Conference (GTC). vol. 2 (2017)
14. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Tech. Rep. LLNL-TR-641973 (August 2013)
15. Kress, J., Klasky, S., Podhorszki, N., Choi, J., Childs, H., Pugmire, D.: Loosely coupled in situ visualization: A perspective on why it's here to stay. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization. pp. 1–6 (2015)
16. Lindstrom, P.: Fixed-rate compressed floating-point arrays. IEEE transactions on visualization and computer graphics **20**(12), 2674–2683 (2014)
17. Moreland, K.: The tensions of in situ visualization. IEEE computer graphics and applications **36**(2), 5–9 (2016)
18. Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D., Panda, D.K.: Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. In: 2013 42nd International Conference on Parallel Processing. pp. 80–89. IEEE (2013)
19. Pulatov, D., Zhang, B., Suresh, S., Miller, C.: Porting IDL programs into Python for GPU-Accelerated In-situ Analysis (2021)
20. Reyes, R., Brown, G., Burns, R., Wong, M.: SYCL 2020: more than meets the eye. In: Proceedings of the International Workshop on OpenCL. pp. 1–1 (2020)
21. Ross, R.B., Amvrosiadis, G., Carns, P., Cranor, C.D., Dorier, M., Harms, K., Ganger, G., Gibson, G., Gutierrez, S.K., Latham, R., et al.: Mochi: Composing data services for high-performance computing environments. Journal of Computer Science and Technology **35**(1), 121–144 (2020)
22. Shi, R., Potluri, S., Hamidouche, K., Perkins, J., Li, M., Rossetti, D., Panda, D.K.D.: Designing efficient small message transfer mechanism for inter-node MPI communication on InfiniBand GPU clusters. In: 2014 21st International Conference on High Performance Computing (HiPC). pp. 1–10. IEEE (2014)
23. Strohmaier, E., Dongarra, J., Simon, H., Meuer, M.: TOP500 List, November 2022 (2022), `https://www.top500.org/lists/top500/2022/11/`
24. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., et al.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2021)
25. Wang, D., Foran, D.J., Qi, X., Parashar, M.: Enabling asynchronous coupled data intensive analysis workflows on gpu-accelerated platforms via data staging
26. Wang, H., Potluri, S., Luo, M., Singh, A.K., Sur, S., Panda, D.K.: MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. Computer Science-Research and Development **26**(3), 257–266 (2011)
27. Zhang, B., Subedi, P., Davis, P.E., Rizzi, F., Teranishi, K., Parashar, M.: Assembling Portable In-Situ Workflow from Heterogeneous Components using Data Reorganization. In: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). pp. 41–50. IEEE (2022)