Dynamic Kernel Fusion for Bulk Non-contiguous Data Transfer on GPU Clusters

Ching-Hsiang Chu, Kawthar Shafie Khorassani, Qinghua Zhou, Hari Subramoni, and Dhabaleswar K. Panda
Department of Computer Science and Engineering,
The Ohio State University, Columbus, OH, USA
Email: {chu.368, shafiekhorassani.1, zhou.2595}@osu.edu
Email: {subramon, panda}@cse.ohio-state.edu

***Abstract*—In the last decade, many scientific applications have been significantly accelerated by large-scale GPU systems. However, the movement of non-contiguous GPU-resident data is one of the most challenging components of scaling these applications using communication middleware like MPI. Although plenty of research has discussed improving non-contiguous data movement within communication middleware, the packing/unpacking operations on GPUs are still expensive. They cannot be hidden due to the limitation of MPI standard and the not-well-optimized designs in existing MPI implementations for GPU-resident data. Consequently, application developers tend to implement customized packing/unpacking kernels to improve GPU utilization by avoiding unnecessary synchronizations in MPI routines. However, this reduces productivity as well as performance as it cannot overlap the packing/unpacking operations with communication. In this paper, we propose a novel approach to achieve low-latency and high-bandwidth by dynamically fusing the packing/unpacking GPU kernels to reduce the expensive kernel launch overhead. The evaluation of the proposed designs shows up to 8X and 5X performance improvement for sparse and dense non-contiguous layout, respectively, compared to the state-of-the-art approaches on the Lassen system. Similarly, we observe up to 19X improvement over existing approaches on the ABCI system. Furthermore, the proposed design also outperforms the production libraries, such as SpectrumMPI, OpenMPI, and MVAPICH2, by many orders of magnitude.**

***Index Terms*—Datatype, GPU, MPI**

I. Introduction

The recent rapid developments of accelerators, domain-specific architectures, and the high-speed interconnects have led the large-scale heterogeneous clusters to become ubiquitous in the High-Performance Computing (HPC) community. In particular, the general-purpose graphic processing unit (GPU) has been widely deployed in current and next-generation supercomputers [1], [2]. As evident in the latest Top500 list, five out of Top10 supercomputers are powered by NVIDIA Volta GPU architecture with NVLink interconnect [3]. These large-scale HPC systems help numerous scientific applications in pushing their boundary [4] and also open new opportunities for artificial intelligence (AI) domains [5].

For utilizing the computing power and accelerating time-to-solution, it is inevitable to involve communication among nodes. In this context, communication models such as Message Passing Interface (MPI) are commonly used in HPC applications. To enable efficient GPU communication, GPU-aware MPI is proposed and extensively studied in the literature for various communication patterns including point-to-point [6]–[10] and collective operations [11]–[13].

Many scientific applications are heavily relying on non-contiguous data transfer to perform *halo-exchange* operations involved in the multi-dimensional domain decomposition [14], [15]. To improve the productivity for application developers, the MPI standard defines a rich set of Derived DataType (DDT) to flexibly represent the non-contiguous memory layout. Furthermore, the MPI runtime can implicitly or explicitly "pack" the non-contiguous data into a contiguous memory region and perform the communication. However, such "packing" and "unpacking" operations are often considered expensive due to the extra data movement and stride access. Moreover, the applications often require multiple non-contiguous data transfers to multiple neighbors. Such "bulk" non-contiguous data transfers severely escalate the overhead of GPU-driven packing/unpacking operations. Prior research, as summarized in TABLE I, has attempted to optimize the packing/unpacking routines by taking advantage of GPU parallelism at the application-level [14], [16], [17], MPI-level [18]–[23], or even exploring packing-free scheme over NVLink/PCIe [24].

A. Motivation

Although existing GPU-driven designs have demonstrated improvement over the traditional CPU-driven pack-

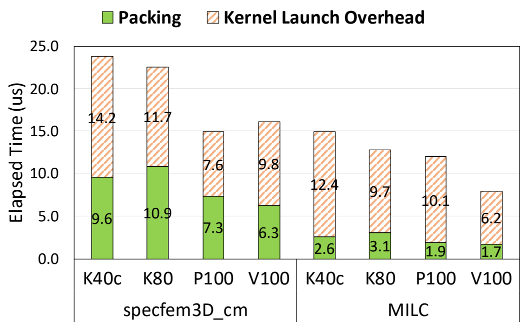


Fig. 1. Motivated example: Time breakdown of GPU-optimized packing kernels [21] on modern NVIDIA GPU architectures. Kernel launch is expensive and outweighs the fast packing kernels.

TABLE I
SUMMARY OF STATE-OF-THE-ART APPROACHES FOR HANDLING GPU-RESIDENT NON-CONTIGUOUS DATA

Bibliography Reference	**Primary Optimization Schemes**	**Layout Cache**	**GPU Driver Overhead**	**Overall Latency**	**Throughput**	**Overlap with Communication**
[14], [16], [17]	Offload packing/unpacking into the optimized GPU kernels for specific memory layout at application-level	N	High	Medium	High	Low
[18]–[22]	Offload packing/unpacking into the optimized GPU kernels for various MPI DDT	N	High	High	High	Medium
[23]	Overlapping packing/unpacking GPU kernels through multi-stream designs	N	High	Medium	High	High
[24], [25]	Adaptive CPU-GPU-Hybrid packing/unpacking and zero-copy methods	Y	Medium	Low	High	High
Proposed	Adaptive Hybrid Approach with Dynamic Kernel Fusion	Y	Low	Low	High	High

ing/unpacking process, they still have significant overhead due to the legacy kernel launch overhead and synchronizations. Fig. 1 presents the average execution time of GPU-optimized packing kernels [21] and its associated overhead to launch the kernels for two common workloads, namely Specfem3D and MILC, that use MPI derived datatype on modern NVIDIA GPU architectures. Though GPU architectures have significantly evolved over the years, the legacy kernel launch overhead remains relatively high and critical, where the packing/unpacking operations are relatively simple and small [26]. Clearly, it violates the best practices of GPU programming to achieve high concurrency [27]. In this situation, the state-of-the-art GPU-driven schemes fail to hide the kernel launch overhead since it spends more cycles on launching the kernels than the useful packing operations on the GPU. Moreover, such overhead delays the real data transfer and overall performance at the application level. This situation implies the low throughput and under-utilization of GPU resources, i.e., high bandwidth memory and massive parallelism.

Fig. 2 illustrates the existing designs for efficient GPU-driven packing/unpacking routines for GPU-resident data. Past research mostly focuses on optimizing the packing and unpacking kernels that require synchronization for each kernel, i.e., refer to *SYNCHRONOUS* in Fig. 2. In such a case, CPU remains busy on synchronization, and GPU may be idle during communication. To further improve the concurrency and overlap between packing kernels and communications, one can perform the packing kernel asynchronously as proposed in [23] (*ASYNCHRONOUS* in Fig. 2). However, the overlap opportunity is limited due to the above-mentioned high kernel launch overhead.

To mitigate the kernel launch overhead and achieve efficient DDT processing for GPU-resident data, this paper is addressing the following broad challenge: **How to minimize the overhead of launching packing/unpacking GPU kernels while maintaining high overlap between communication and packing/unpacking operations?**

B. Contribution

This paper tackles the challenges mentioned previously and proposes a dynamic kernel fusion scheme to alleviate the kernel launch overhead for "bulk" non-contiguous data transfer.

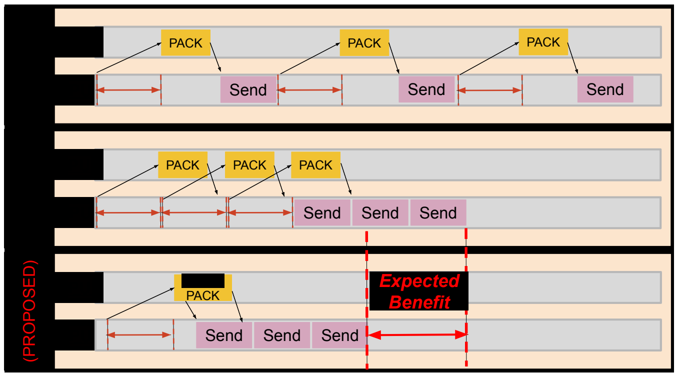


Fig. 2. Overview of Existing Designs and the proposed Dynamic Kernel Fusion for MPI datatype processing on GPU

As highlighted in the bottom of Fig. 2, the proposed scheme "fuses" multiple pack/unpacking kernels into a single one. Taking advantage of the powerful modern GPU architecture, e.g., higher clock speed and more streaming multiprocessors (SMs), we see that the fused kernel's execution time can be the same as the typical packing/unpacking kernel while only costing one launch overhead. Thus, more packing/unpacking operations can be done within the same duration. By intelligently fusing the kernels and not synchronizing at the kernel boundary, the proposed design can significantly reduce the launch overhead and packing/unpacking time. In the evaluation with representative application kernels, the proposed design can improve the performance of bulk non-contiguous data transfer in the order of magnitude on the GPU-enabled HPC systems, including top-ranked Lassen and ABCI systems. This paper makes the following key contributions.

- Provide an analysis of existing solutions for communicating bulk non-contiguous GPU-resident data (Section III).
- Propose a kernel-fusion framework to significantly reduce the overhead when performing datatype processing operations on GPU (Section IV).
- Propose an advanced dynamic kernel fusion scheme to leverage the concept of the cooperative group to maximize GPU utilization (Section IV).
- Demonstrate the performance improvement using the representative application kernels on GPU-enabled HPC systems (Section V).

II. Background

This section describes the background knowledge related to this paper.

A. GPU Communication

NVIDIA GPUDirect technology eliminates additional memory copies and reduces CPU overhead through enabling third-party PCIe devices such as the host channel adapter (HCA) to directly access GPU memory. This technology includes features such as peer-to-peer access and remote direct memory access (RDMA) that significantly improve bandwidth, reduce latency, and further enhance overall performance of applications [28]. Message Passing Interface (MPI) is a programming paradigm used in parallel applications to enable efficient communication between distributed processes. **CUDA-aware MPI** enables developing applications on the large-scale HPC systems equipped with NVIDIA GPUs. Through CUDA-aware MPI, GPU buffers can be directly passed to the MPI primitives, eliminating the need to explicitly stage device buffers through the host at the application-level. Many optimization schemes, built on top of GPUDirect technology, have been proposed in the literature to significantly improve the GPU communication in CUDA-aware MPI libraries [6]–[8], [10].

B. Non-contiguous Memory Transfers with CUDA-Aware MPI

In the multi-dimensional domain decomposition used in many scientific applications such as physics simulation and weather forecasting [14], [15]. As large multi-dimensional data is distributed into multiple GPUs, GPUs require communicating the boundary information (or so-called ghost region) with their neighbors. Such boundary information can often be non-contiguous in the GPU memory. Fig. 3 depicts an example of a 2D halo exchange among four GPUs where the 'columns' in the ghost regions are non-contiguous. To support non-contiguous data transfer over MPI primitives, MPI standard defines various derived datatypes including vector, indexed, subarray, struct, and more [29]. Although many optimization schemes have been proposed to improve the MPI derived datatype processing for GPU-resident data using carefully-designed CUDA kernels [18]–[22], application developers typically implement their own packing/unpacking routine due to the lack of performance or MPI primitives in GPU-aware MPI libraries. We will address these shortcomings of existing methods in Section III.

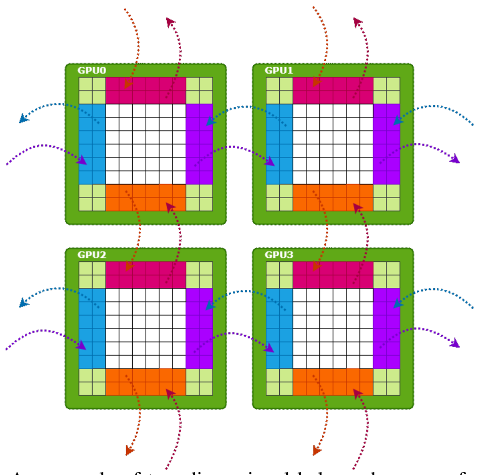


Fig. 3. An example of two-dimensional halo exchange on four GPUs

III. Analysis of Existing Solutions

In this section, we analyze the existing solutions with MPI primitives that can be used to accomplish bulk non-contiguous data transfer. Fig. 4 depicts three primary methods to perform non-contiguous data transfer in MPI applications.

A. MPI-level Explicit Packing/Unpacking

MPI standard defines blocking pack and unpack routines to allow implementations in the MPI library for its own packing and unpacking function. Algorithm 1 shows the usage of this approach for halo exchanges and Fig. 4(a) illustrates the communication flow. As can be seen, each MPI_Pack/MPI_Unpack call is a blocking routine and has to ensure the completion of packing/unpacking before returning to the application. Despite offloading the efficient packing/unpacking kernels to the GPU, the extra synchronization, in order to comply with the MPI semantic (i.e., red dotted line), prohibits overlapping the communication and packing/unpacking operations.

Algorithm 1 MPI-Level Explicit Pack/Unpack

```
1:  MPI_Type_create_*();
2:  MPI_Type_commit(mpiddt);
3:  for each neighbor $i$ do
4:      MPI_Irecv(packed_rbuf[i]);
5:      for each boundary buffer $j$ for $neighbor$ $i$ do
6:          MPI_Pack(buf[i][j], packed_sbuf[i][j], mpiddt);
7:      end for
8:      MPI_Isend(pack_sbuf[i]);
9:  end for
10: MPI_Waitall(...);
11: for each neighbor $i$ do
12:     for each boundary buffer $j$ for $neighbor$ $i$ do
13:         MPI_Unpack(packed_rbuf[i][j], rbuf[i][j], mpiddt);
14:     end for
15: end for
16: /* Computation on boundary data */
```

B. Application-level Packing/Unpacking

Since the previous approach requires explicit synchronization in each pack and unpack routine, and there are no non-blocking primitives defined, an application often implements their specialized GPU kernels to perform the packing and unpacking explicitly. In this way, applications may perform the synchronization only if needed. Algorithm 2 shows the implementation details of this approach. As can be seen, multiple asynchronous kernels are launched for packing and unpacking non-contiguous data (lines 2-5 and 12-16) and only a single synchronization point at the final kernel boundary. Fig. 4(b) shows the communication flow of this approach. In

Algorithm 2 Application Level Explicit Pack/Unpack

```
for each neighbor i do
    for each boundary buffer j for neighbor i do
        pack_gpu_kernel(sbuf[i][j] to packed_sbuf[i][j]);
    end for
end for
Synchronize_TO_GPU();
for each neighbor i do
    MPI_Irecv(packed_rbuf[i]);
    MPI_Isend(packed_sbuf[i]);
end for
MPI_Waitall(...);
for each neighbor i do
    for each boundary buffer j for neighbor i do
        unpack_gpu_kernel(packed_rbuf[i][j] to rbuf[i][j]);
    end for
end for
Synchronize_TO_GPU();
/* Computation on boundary data */
```

many cases, the application's optimized GPU may perform pack/unpack more efficiently than MPI runtimes provide. However, it is hard to generalize and reuse the kernel for other applications, unlike MPI. This approach requires more effort at the application level, as can be seen in Algorithm 2 (i.e., more lines of code compared to other approaches) and lacks overlap between packing/unpacking kernels and communication.

C. MPI-level Implicit Packing/Unpacking

To avoid explicit synchronization, we can also pass the non-contiguous buffer to the MPI primitives. In such a case, the packing or unpacking operations may be performed by the MPI library implicitly. Algorithm 3 explains the usage of this approach, and we can see a productive way to use only 10 lines to implement a halo exchange routine, which can be significantly cleaner than previous two approaches. One key difference here is that we issue send and receive operations for each boundary buffer instead of only issue one send/recv operation with the packed buffer to a neighbor. This approach gives high flexibility to the MPI runtime to schedule the communication and to perform packing/unpacking operations. Fig. 4(c) depicts one possible flow where asynchronous approach is used for offloading packing/unpacking to GPUs [23]. Ideally, the overlap between communication and packing/unpacking is possible. However, as discussed earlier, the high GPU driver overhead introduces significant penalties.

In this paper, we aim to propose a kernel fusion framework to mitigate the driver overhead and transparently maximize the productivity and performance of the MPI-level implicit packing/unpacking approach.

IV. Proposed Designs

To mitigate the overhead of launching GPU kernels and improve the utilization of GPU computational resources during non-contiguous data communication, we propose a framework

Algorithm 3 MPI-Level Implicit Pack/Unpack

```
MPI_Type_create_*();
MPI_Type_commit(mpiddt);
for each neighbor i do
    for each boundary buffer j for neighbor i do
        MPI_Irecv(rbuf[i][j], mpiddt,...);
        MPI_Isend(sbuf[i][j], mpiddt,...);
    end for
end for
MPI_Waitall(...);
/* Computation on boundary data */
```

to exploit the kernel fusion approach. This section elaborates on the proposed framework to support kernel fusion for bulk non-contiguous data transfer, and how a communication middleware such as MPI can take advantage of it to achieve overlap between DDT processing and communication. It also elaborates on the design considerations to optimize the kernel fusion framework in practice.

A. Kernel Fusion Framework

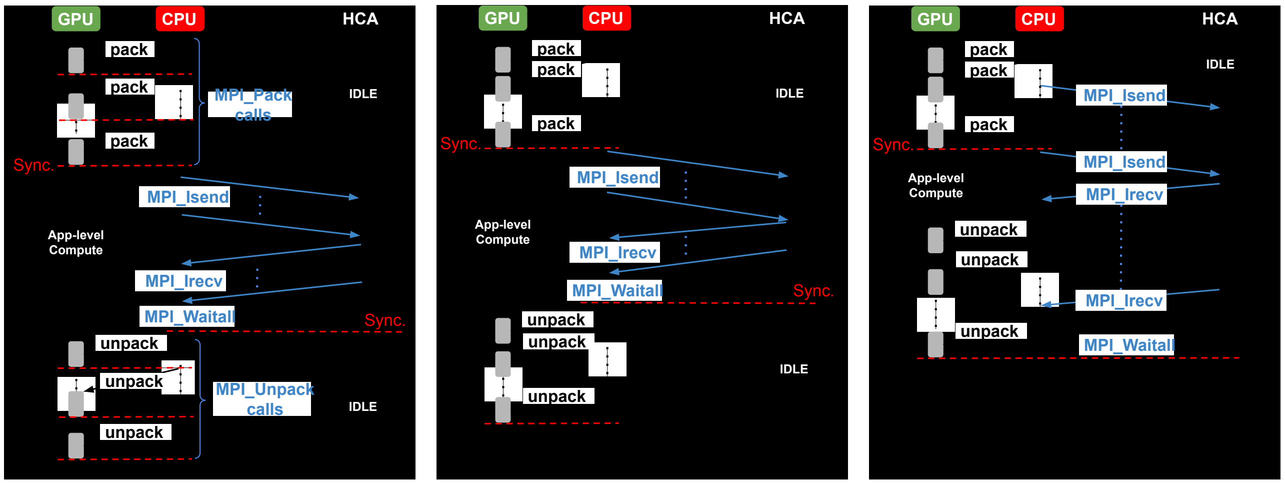
To design a kernel fusion framework, there are three vital requirements: 1) a request queue to record the operations/kernels to be fused, 2) a scheduler to maintain the request queue and trigger the proper fused kernel when needed, 3) an efficient fused kernel to concurrently execute various operations for non-contiguous buffers.

1) Request list: As shown at the top of Fig. 5, we use a circular buffer to contain multiple requests that require supported operations on the GPU for non-contiguous buffers. Each request should contain the following necessary information:

- *UID*: a unique number, e.g., unsigned integer number, used to identify the request object.
- *requested operation*: this could be one of *Packing, Unpacking or DirectIPC* (i.e., a direct non-contiguous data transfer over NVLink/PCIe proposed in [24]) kernels.
- *origin buffer*: the pointer of a non-contiguous buffer to be packed or contiguous buffer to be unpacked.
- *target buffer*: the pointer of a non-contiguous buffer to store unpacked data or contiguous buffer to store packed data.
- *data layout*: the cached data layout entry (follow the scheme proposed in [24]).
- *request status*: the status of *IDLE*, *PENDING*, *BUSY* or *COMPLETED* for progress engine to query.
- *response status*: similar to request status, but only allow the GPU to update for reporting the status of a given request.

In addition, the *Scheduler* maintains two indexes, namely Head and Tail, to know which requests are pending to be fused.

2) Scheduler: Next, we need a *scheduler* to maintain the request list for interacting with the communication progress engine and GPU. As illustrated in Fig. 5, a scheduler has four primary functions: ① enqueue requests from the progress



(a) MPI-level Explicit Packing/Unpacking (b) Application-level Packing/Unpacking (c) MPI-level Implicit Packing/Unpacking

Fig. 4. Existing GPU-enabled approaches to handle non-contiguous data transfer for GPU-resident data

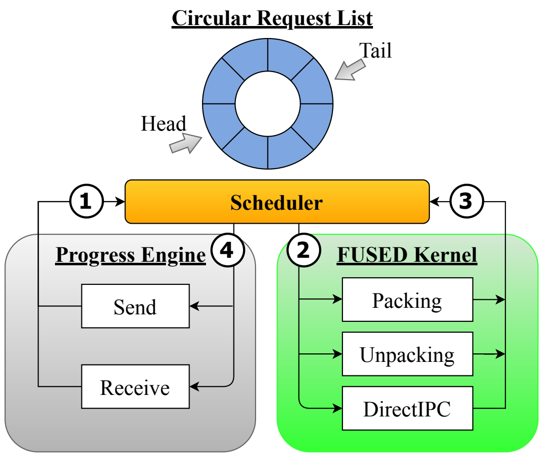


Fig. 5. Proposed Framework of Dynamic Kernel Fusion

engine, ② launch fused kernel on the GPU with proper configuration, ③ complete the requests, and ④ provide status to the progress engine.

In ①, the progress engine may have packing requests for a non-contiguous buffer. The scheduler takes the information from the progress engine and creates a corresponding request object, i.e., fill up the information described in Section IV-A1, and insert it to the tail of the request list, then the Tail will be moved to the next IDLE entry. Here, the scheduler returns a UID to the progress engine to be used for checking the status. Also, the UID can be a negative number to notify the progress engine that the operation cannot be fused for various reasons such as running out of the request list, then the progress engine can take a fallback mechanism and proceed with other alternatives.

If there are multiple operations in the request list and meet the conditions (will be discussed in Section IV-C), the scheduler may launch a single fused kernel with an array of requests as the input (② in Fig. 5). As soon as GPU kernel completes any request, a GPU thread immediately signals the completion by updating the *response status* for a given request object (③ in Fig. 5). As a result, the scheduler does not have to perform an explicit synchronization at the kernel boundary.

Finally, whenever the progress engine needs to act on a request, it can use the request UID to make a query through the scheduler (④ in Fig. 5). The scheduler can simply compare the *request status* to *response status* to realize the status. If the progress engine has no more non-contiguous buffers to be processed and fused, it can request the scheduler to immediately launch the fused kernel to avoid wasting cycles.

It is worth noting that one single CPU thread is sufficient to act as scheduler and progress engine to avoid any context switching overheads if an application has a dedicated thread for communication. If an application is utilizing one CPU thread for MPI communication and heavy computation, it would benefit from using a separate thread to work as the scheduler. In this work, we implemented and evaluated the scheduler on the same thread as communication progress engine, which is the common scenarios for most GPU-enabled applications.

3) Implementation of Fusion Kernel: The primary functionality of the fused kernel is to partition the thread blocks, distribute the workload, then perform the required operations. Based on the input size in each request, we could decide how many threads are required to perform the packing, unpacking, or DirectIPC operation. For example, for a packing operation on a small buffer, which only has 128 elements, we could schedule a thread block with less than 128 threads to efficiently complete the task and free to the rest of the threads for other requests. To achieve this high concurrency kernel with fine-grained thread blocks, we leverage a feature called 'Cooperative Group' in the CUDA programming platform for NVIDIA GPUs [30]. As demonstrated in Fig. 6, after the first *Partition* phase, each thread block will work on different requests with corresponding operations (i.e., device functions). Note that each thread block will perform synchronization separately and signal the completion to the request list.

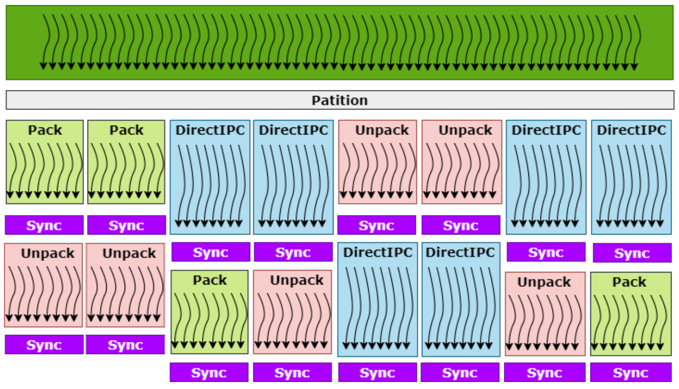


Fig. 6. An Example of a fused kernel that uses 8 thread blocks to process 16 requests, and each thread block can work on different operations concurrently

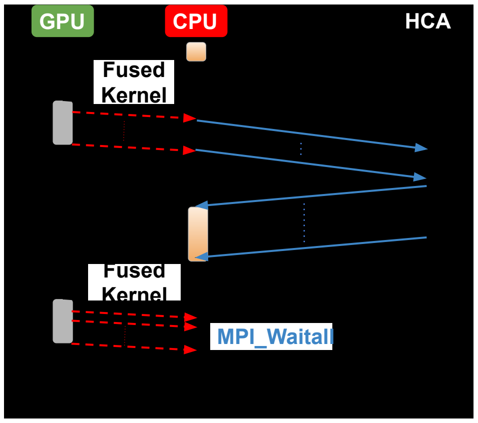


Fig. 7. Communication flow with Dynamic Kernel Fusion

B. Communicating Non-contiguous Buffers with Kernel Fusion Framework

Once we have the kernel fusion framework mentioned above, we now discuss how the progress engine in a communication middleware can take advantage of it. Fig. 7 illustrates a high-level overview of communication flow when kernel fusion is used. Here, we elaborate on the designs from the sender and receiver perspectives.

1) Sender side: On the sender side, the only possible operation is packing the non-contiguous buffers. When an application is calling multiple non-blocking send operations, i.e., MPI_Isend, with non-contiguous data, the sender process first retrieves the cached data layout (please refer to [24] for designs of datatype layout cache) and signals the scheduler with the required information mentioned in Section IV-A. Once the packing request is enqueued, sender associates the UID returned by the scheduler with the corresponding send request (e.g., MPI_Request), then the sender returns immediately to the application, i.e., the communication gets delayed. This process may repeat multiple times if there are multiple non-contiguous buffers to be transferred.

To progress the delayed communication, the runtime may enter the progress engine periodically and check the status with the scheduler. Note that the progress engine and scheduler may be running on a separate and desiccate CPU thread to increase concurrency. As soon as any packing request is completed by the GPU (red dot line in Fig. 7), the progress engine issues the send operation immediately. If a send operation uses an eager protocol for the small message, the progress engine sends out the packed data. If it is an operation with rendezvous protocol, there are two sub-protocols that could be used, namely RGET and RPUT. In the RGET protocol, the progress engine sends out Request-To-Send (RTS) packets after the non-contiguous data is packed. Then the receiver may perform an RDMA-READ operation through InfiniBand. For the RPUT protocol, the sender can send the RTS packet before the packing operation has taken place and only check the completion of the packing request when a Clear-To-Send (CTS) packet is received. In this way, the hand-shaking procedure can be overlapped with the packing operations.

2) Receiver side: For a receiver side, it may fuse Unpacking or DirectIPC operations into the kernel. Similar to the sender side, the receiver prepares the required information when a non-contiguous data is expected to be received (e.g., calling MPI_Irecv) and insert a callback function to that receive request. Upon receiving data, the callback function gets called and signals the scheduler to enqueue the unpacking or DirectIPC requests. For unexpected messages, the signal method occurs until the application reaches the corresponding MPI_Irecv, and no callback function is required. The progress engine checks the completion of fused requests only when the application requires the data, e.g., when calling MPI_Wait or MPI_Waitall.

Fig. 7 shows a simplified scenario where send and receive operations happen serially. Note that the send and receive operations can occur simultaneously, and the proposed framework may fuse the packing, unpacking, and DirectIPC operations into a single kernel.

C. Design Considerations for Optimal Kernel Fusion

In this section, we discuss the design considerations for the optimal kernel fusion scheme to maximize the GPU utilization while balancing the delayed communication.

To have an efficient kernel fusion framework, it is critical for the scheduler to realize when to launch the fused kernels to balance the reduced kernel launch overhead with fused kernel and delayed communication. Naively fusing all kernels may result in performance degradation as the communication gets delayed and also can reduce the overlap opportunities between the the communication and the fused kernels. Broadly, there are two primary scenarios the scheduler should launch the kernel: 1) the communication progress engine has no more operations to request and reaches the synchronization point, e.g., MPI_Waitall, 2) the fused kernel has enough work to do, e.g., the execution time can be longer than the kernel launch overhead.

In the first scenario, the scheduler must launch the kernel to allow the progress engine to proceed. However, the second scenario is difficult to realize. In this paper, we use the heuristic method to activate the fused kernel based on the experimental results. Fig. 8 shows the performance

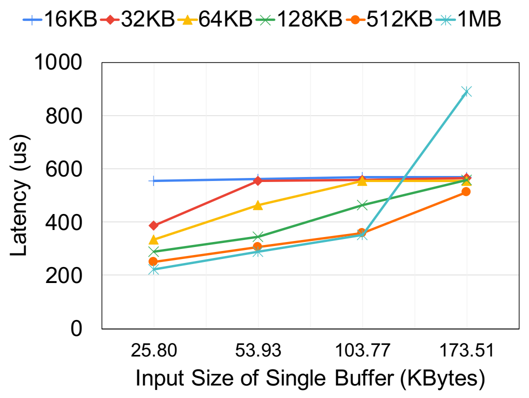


Fig. 8. Performance Effects of fused kernel threshold: using specfem3D_cm workload (i.e., MPI indexed type)

effects when using a different threshold to launch the packing/unpacking kernel for the specfem3D_cm workload, which has a sparse layout (e.g., thousands of blocks), with 32 continuous MPI_Isend/MPI_Irecv operations. As can be seen, when the threshold is too low, e.g., 16KB, the execution time remains high because it activates the kernel launch frequently. We refer this phenomena as *under-fused*. As we increase the threshold, we start observing reduced latency due to the reduced kernels. If the threshold is too high (i.e, *over-fused.*), e.g., above 1MB, we observe performance degradation for large input size as the delayed communication is not able to overlap with the fused kernel on GPU.

We note that such threshold varies from system to system based on the hardware (e.g., GPU, network models), firmware, and software (e.g., driver versions, OS distributions). To figure out the optimal threshold, the principle is to make sure the running time of the fused kernel is longer than the kernel launch overhead either through fusing more kernels or fusing more data in each kernel. In this paper, we use the above-mentioned heuristic method to find the optimal threshold for a given workload on a given system. In many workloads including sparse and dense layout, we observe that fusing around 512KB data can achieve the best performance on the systems we tested in this paper. In future work, we plan to develop a model-based prediction to dynamically figure out the optimal threshold for kernel fusion that can maximize the overlap between the fused kernel and communication.

V. Performance Evaluation

This section includes the performance evaluation of the proposed design compared to state-of-the-art MPI datatype processing schemes for GPU-resident data.

A. Experimental Platforms and Setup

The experiments were carried out on two cutting-edge GPU platforms: 1) Lassen system, operated by Lawrence Livermore National Laboratory, U.S.A., is powered by IBM POWER9 CPUs and NVIDIA V100 GPUs, and 2) AI Bridging Cloud Infrastructure (ABCI) system is operated by the National Institute of Advanced Industrial Science and Technology, Japan. ABCI is powered by Intel Xeon Gold CPU and NVIDIA V100 GPU architectures. More hardware and software details are summarised in Table II. The key difference is the interconnect where Lassen's POWER9 architecture enables NVLink between CPU and GPU, and ABCI system is using traditional PCIe Gen3, which is slower than NVLink. We evaluated the proposed and existing schemes as follows: 1) the proposed dynamic kernel fusion scheme (*Proposed* and *Proposed-Tuned*), 2) existing GPU-driven synchronous scheme (*GPU-Sync*) [8], [22], 3) existing GPU-driven asynchronous scheme (*GPU-Aync*) [23], and 4) existing CPU-GPU-Hybrid scheme (*CPU-GPU-Hybrid*) [24]. The schemes as mentioned above were implemented on top of the MVAPICH2-GDR library, which is a high-performance GPU-Aware MPI implementation [31]. We also compared the proposed design with production libraries such as SpectrumMPI v10.3.0 that is only available on POWER systems (*SpectrumMPI*), and OpenMPI v4.0.3 with UCX v1.8.0 (*OpenMPI*).

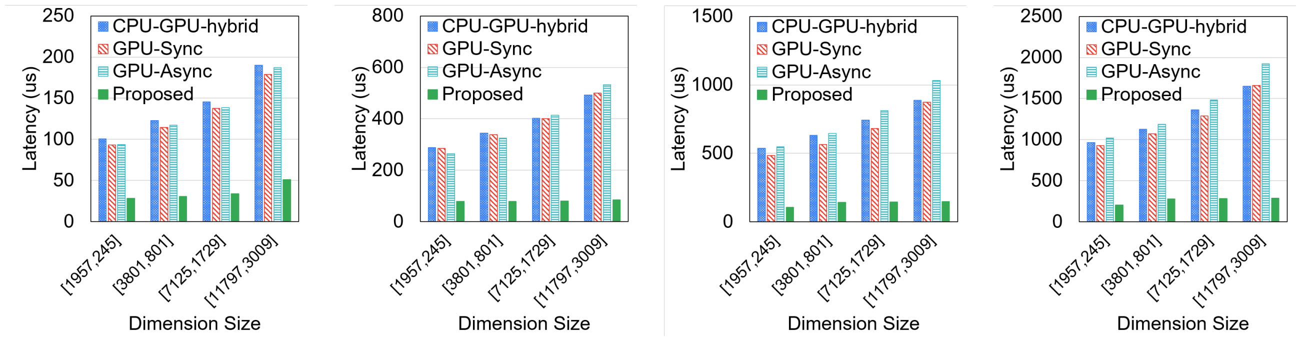
In this paper, we evaluated the application kernels with representative datatype layouts. Specifically, we use the sparse layout, i.e., more than thousands of small blocks, created by indexed (*specfem3D_oc*) and struct-on-indexed (*specfem3D_cm*) which commonly used in Geophysical Science applications. Also, we also studied the dense layout, i.e., less than thousand of blocks created by vectors (*NAS_MG*) and nested vectors (*MILC*) that are used for Fluid Dynamics and Quantum Chromodynamics applications. We implemented the GPU-enabled application kernels based on the popular benchmarks including ddtbench [32] and a kernel of 3D domain decomposition [33]. All numbers reported in the paper are the average of 500 iterations, excluding the 50 warm-up iterations.

B. Performance Effects on Bulk Communication

In multi-dimensional domain decomposition, as explained in Section II-B, each GPU performs multiple halo exchange operations with its neighbors. In this section, we present the evaluation and study how the bulk communication may affect the performance of existing packing schemes and the proposed schemes. Figures 9 and 10 presents the performance of bulk communication with sparse and dense layout using specfem3d_cm and MILC workloads, respectively, on the Lassen system. We increased the number of buffers to be exchanged (i.e., number of boundary data to neighbors) from 1 to 16 to observe the impact to various GPU-driven DDT processing schemes. In the sparse layout exhibited in Fig. 9, the proposed kernel fusion design outperforms all existing

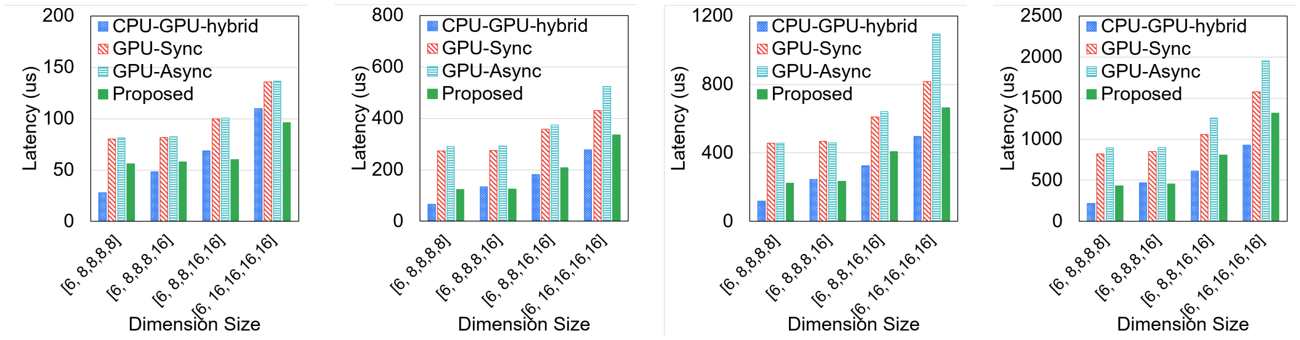
TABLE II
Experimental environment

Cluster Specs	**LLNL Lassen**	**ABCI**
CPU Processor	Dual-socket IBM POWER9 AC922 3.1GHz, 44 Cores/socket	Dual-socket Intel Xeon Gold 6148 2.4 GHz, 20 Cores/socket
System Memory	256 GB	384 GB
GPU Processor	NVIDIA Tesla V100×4	NVIDIA Tesla V100×4
GPU Memory	16 GB	16 GB
Interconnects between CPU and GPU	NVLink-2 (one-way 75 GB/s)	PCIe Gen3×16 and ×64 switches (one-way 32 GB/s)
Interconnects between GPUs	NVLink-2 (one-way 75 GB/s)	NVLink-2 (one-way 50 GB/s)
Interconnects between nodes	Dual-rail Mellanox InfiniBand EDR (one-way 25 GB/s)	Mellanox InfiniBand EDR×2 (one-way 25 GB/s)
Operating System	RHEL 7.3 (4.14.0-115.10.1.1)	CentOS (3.10.0-862)
NVIDIA Driver Version	418.87.00	440.33.01
CUDA Toolkit Version	10.1.243	10.2.89



(a) 1 Non-contiguous Buffer (b) 4 Non-contiguous Buffers (c) 8 Non-contiguous Buffers (d) 16 Non-contiguous Buffers

Fig. 9. Performance Evaluation of bulk non-contiguous inter-node data transfer: Sparse layout with specfem3D_cm workload (lower is better)



(a) 1 Non-contiguous Buffer (b) 4 Non-contiguous Buffers (c) 8 Non-contiguous Buffers (d) 16 Non-contiguous Buffers

Fig. 10. Performance Evaluation bulk non-contiguous inter-node data transfer: Dense layout with MILC workload (lower is better)

GPU-driven and CPU-GPU-Hybrid schemes for all dimension sizes up to 5.9X. On the other hand, the dense layout presented in Fig. 10 shows that *CPU-GPU-Hybrid* can actually perform better although the proposed design outperforms *GPU-Sync* and *GPU-Async* schemes. This is because the *CPU-GPU-Hybrid* use a low-overhead CPU-driven method for packing/unpacking dense and small data that can totally remove the GPU driver overhead [24]. Note that it requires a special kernel module called GDRCopy [34], which may not be available in all HPC systems. We can also observe that *GPU-Async* performs worse than *GPU-Sync* even if there are multiple packing/unpacking operations that can potentially to be overlapped with each other. However, the extra synchronizations, e.g., *cudaEventRecord* and *cudaEvenQuery* calls, are adding more penalties that cause the performance to degrade when again the packing/unpacking operations are not long enough on modern GPUs to overlap and hide the overhead.

To provide the insights into these GPU-driven designs, Fig. 11 shows the time breakdown of them using the MILC workloads with two GPU nodes on the ABCI cluster. Here, we compare five primary costs when performing back-to-back 16 non-contiguous data transfers over 100 iterations: 1) *(Un)Pack*: Packing and Unpacking kernels, 2) *Launching*: Overhead of launching GPU kernels, 3) *Scheduling*: *GPU-Async* schedules kernels without explicit synchronization using CUDA APIs such as *cudaEventRecord* [23]. In the proposed design, scheduler enqueues and dequeues the pack/unpack tasks. This cost is meaningless for *GPU-Sync*. 4) *Sync.*: synchronization

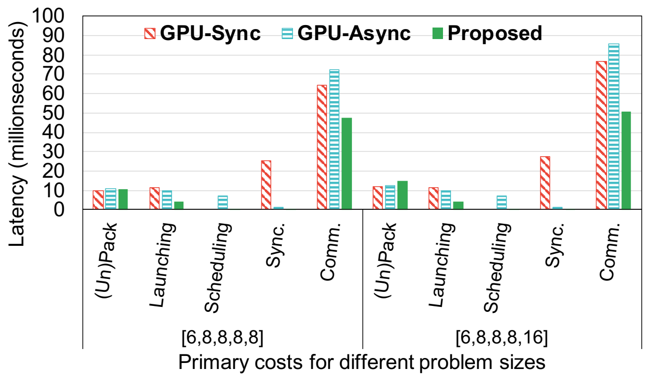
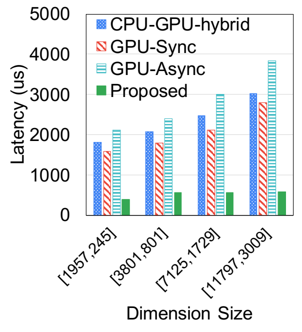
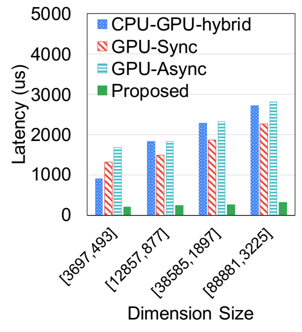


Fig. 11. Time breakdown of different GPU-driven designs for MILC workloads: 1) (Un)Pack: Pack/Unpack kernels, 2) Launching: Kernel launch overhead, 3) Scheduling: Scheduling GPU kernels, 4) Sync.: CPU-GPU Synchronization, 5) Comm.: Communication between GPUs.

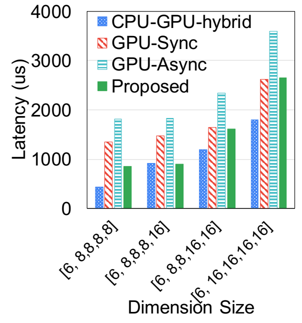
between CPU and GPU to complete the packing and unpacking kernels. In *GPU-Sync*, explicit synchronization such as *cudaStreamSynchronize* is used. *GPU-Async* may use APIs like *cudaEventQuery* to ensure the completion of GPU kernels. The proposed scheduler uses a polling scheme mentioned in Section IV-A2. 5) *Comm.*: ‘observed‘ communication time for packed data, i.e., some communication may be overlapped with pack/unpack kernels. As can be seen in Fig. 11, *GPU-Sync* and *GPU-Async* has higher launching overhead than the proposed designs while they spend some amount of time on packing/unpacking kernels. *GPU-Sync* always has the highest overhead on explicit synchronization. The proposed kernel fusion design has the lowest launching and synchronization



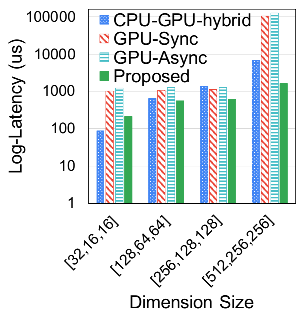
(a) specfem3D_cm



(b) specfem3D_oc

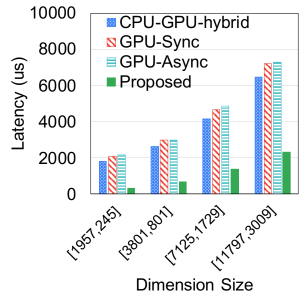


(c) MILC

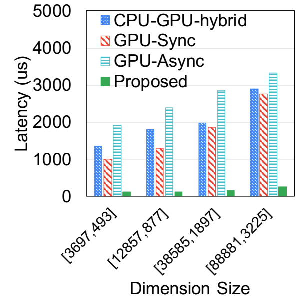


(d) NAS_MG

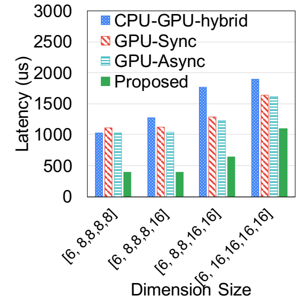
Fig. 12. Performance Evaluation of bulk non-contiguous data transfer between two GPU nodes on Lassen system (lower is better)



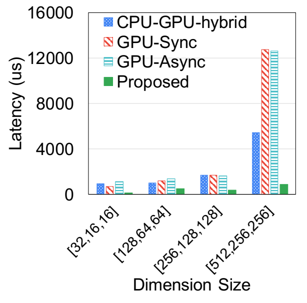
(a) specfem3D_cm



(b) specfem3D_oc



(c) MILC



(d) NAS_MG

Fig. 13. Performance Evaluation of bulk non-contiguous data transfer between two GPU nodes on ABCI system (lower is better)

overhead while maintaining the highest overlap with communication. The scheduling overhead of the proposed scheduler has insignificant overhead as low as $2\mu s$ per message. Note that we omit the results of other workloads to avoid repetition as they show a similar trend.

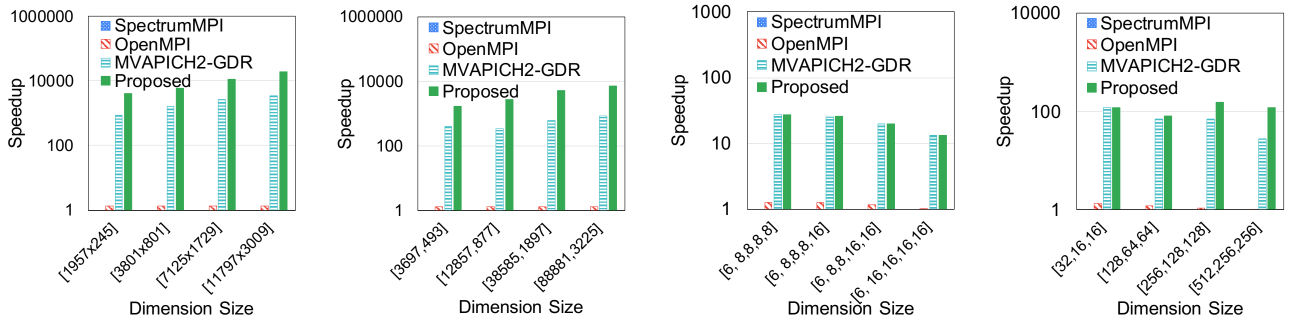
C. Evaluation of different non-contiguous data layouts

Next, we present the evaluation of various application kernels with the 3D halo exchange operations that involves 32 non-blocking communication that uses non-contiguous data layout. Note that a typical 3D domain decomposition would involve 27 boundary data to be exchanged. We explicitly stressed the communication in these experiments. Fig. 12 reports the performance on Lassen system. For the sparse layouts shown in figures 13(a) and 13(b), the proposed design significantly outperform all existing designs. Specifically, the proposed design yields up to 8.5X, 7.1X and 8.9X improvement compared to *CPU-GPU-Hybrid*, *GPU-Sync*, and *GPU-Async* schemes, respectively. Fig. 12(c) shows an only exception where *CPU-GPU-Hybrid* performs the best for small and dense layout, as explained previously. Nevertheless, for large and dense layout like NAS shown in Fig. 12(d), the proposed design again significantly outperforms *CPU-GPU-Hybrid*, *GPU-Sync*, and *GPU-Async* schemes around 1.4X to 5.8X. For NAS workload with large dimension size, i.e., large buffer size, up to 80X performance improvement can be observed from the proposed design over *GPU-Async*. This could be caused by the poor to zero overlaps between packing/unpacking kernels and communication. On the other side, the proposed design significantly reduces the latency for all workloads on ABCI systems, as depicted in Fig. 13. We can also see that *GPU-Async* can slightly outperform *GPU-Sync* as shown in figures 13(c) and 13(d). Such an outcome may be caused by the slower PCIe interconnect on the ABCI system to allow more overlap. Nonetheless, the proposed design with kernel fusion can achieve up to 19X and 14.7X improvements over the existing schemes for sparse and dense layouts.

Finally, we compare the proposed design with the production GPU-Aware MPI libraries including *SpectrumMPI*, *OpenMPI* with UCX, and *MVAPICH2-GDR* on Lassen system. We note that the *SpectrumMPI* and *OpenMPI* do not have optimized support for non-contiguous data movement and use a naive approach, which uses multiple memory copies such as *cudaMemcpyAsync*, to pack and unpack non-contiguous GPU-resident data. As a result, the optimized proposed design can be thousand times faster than *SpectrumMPI* and *OpenMPI*. Compared to the optimized scheme in *MVAPICH2-GDR*, which adaptive use *CPU-GPU-Hybrid* and *GPU-Sync* schemes, the proposed design can achieve up to 8.8X and 4.3X lower latency for sparse and dense layouts.

VI. Related Work

To efficiently mitigate performance penalties caused by transferring non-contiguous data, extensive research has been explored with MPI derived datatypes processing. Traff et al. propose "flattening on the fly" scheme to optimize the parse of MPI DDT layout [35]. Gropp et al. provide a guideline for using various aspects of datatype [36] based on the performance evaluation. Byna et al. propose packing algorithms that take advantage of memory-optimization techniques, which



(a) specfem3D_cm (b) specfem3D_oc (c) MILC (d) NAS_MG

Fig. 14. Performance comparison with production communication libraries on Lassen system (Normalized to SpectrumMPI; higher is better)

improves the performance of derived datatypes [37]. There are other approaches for MPI datatype communication over the InfiniBand network such as pack/unpack-based, and copy-reduced approaches [38]. To support processing MPI datatype routines efficiently outside of the MPI implementations, Ross et al. propose an open-source library, MPITypes [39]. Nevertheless, none of these approaches have taken the opportunity to eliminate the expensive packing/unpacking operations. Santhanaraman et al. design a new scheme called Send Gather Receive Scatter, to achieve zero-copy MPI derived datatype communication over InfiniBand networks across nodes [40]. In [25], the MPI DDT layout extraction and caching are analyzed thoroughly. A new zero-copy scheme for MPI DDT is proposed by leveraging inter-process load-store operations on CPU and GPU memory within the node.

In respect to the GPU-resident data, the first study provides a significant speedup over CPU-based design for datatype processing was done by Wang et al. [18]. They process non-contiguous datatypes by leveraging a multi-stage pipeline of data transfer and offloading packing/unpacking processing from the host to the device. Jenkins et al. concluded that non-contiguous data transfer could improve performance by kernelizing the packing operations into the GPU [19], [20]. Rong et al. propose a novel packing framework, called HAND, to efficiently pack and unpack non-contiguous data on GPU directly [21]. To obtain a higher overlap between CPU and GPU executions and eliminate unnecessary synchronizations, [23] propose an asynchronous design by taking advantage of several CUDA features. Wu et al. implement a different way to offload the packing and unpacking operations onto the GPU and seamlessly integrate with RDMA networks [22]. However, significant packing/unpacking overhead still exists in these works because of the expensive operations such as memory allocations, copies, and synchronizations.

Girolamo et al. [41] implement full non-contiguous memory transfer processing and work with sPIN, a packet streaming processor, to develop scheduling strategies that enhance datatype processing. Chu et al. [24] propose a zero-copy scheme to exploit load-store semantics over NVLink/PCIe and achieve pack-free mechanism. Moreover, they implement an adaptive scheme on selecting the optimal CPU- or GPU-driven packing/unpacking scheme if NVLink/PCIe is not available between GPUs. The proposed framework can be combined with its adaptive protocol as an additional option.

VII. Conclusion

In many scientific applications that require operations such as multi-dimensional domain decomposition among GPUs, the 'bulk' non-contiguous data transfer between GPUs is involved when running at scale and dominates the overall communication time. With the lack of MPI standard for non-blocking pack/unpack routine and proper runtime optimizations in the communication library, applications often rely on its specialized packing/unpacking scheme, which significantly reduces productivity and performance. In this paper, we analyze the existing packing/unpacking approaches at the application level and MPI level. Based on the observation, we propose a kernel fusion approach to transparently reduce the expensive kernel launch overhead that is the main culprit of performance degradation in existing designs. The proposed kernel fusion framework combines the proper design considerations with existing communication protocols to significantly reduce the driver overhead while increasing GPU utilization and maintaining high communication overlap. The performance evaluation shows improvement up to 19X and 5X for sparse and dense non-contiguous layout, respectively, compared to the state-of-the-art GPU-driven approaches on the top supercomputers, including Lassen and ABCI systems. The proposed schemes will be made publicly available under the MVAPICH2 project [31].

In the future, we plan to evaluate the proposed designs with more application workloads that involve bulk non-contiguous data transfer for GPU-resident data.We also plan to develop a model-based prediction method to automatically optimize the parameters for kernel fusion framework to always maximize the overlap on any GPU-enabled systems.

Acknowledgment

This research is supported in part by NSF grants #1931537, #1450440, #1664137, #1818253, and XRAC grant #NCR-130002. The authors would like to thank Dr. Olga Pearce and Dr. Kathryn Mohror for providing access to Lassen system. We also thank Dr. Shinichiro Takizawa for providing access to ABCI testbed. Finally, we thank the anonymous reviewers for the valuable comments to improve this paper.

References

[1] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "TOP 500 Supercomputer Sites," http://www.top500.org, 1993, Accessed: August 16, 2020.

[2] Oak Ridge National Laboratory, "Frontier," https://www.olcf.ornl.gov/frontier/, 2019, Accessed: August 16, 2020.

[3] NVIDIA, "Whitepaper: NVIDIA Tesla P100, section 'NVLink High Speed Interconnect'," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2019, Accessed: August 16, 2020.

[4] O. Fuhrer, T. Chadha, T. Hoefler, G. Kwasniewski, X. Lapillonne, D. Leutwyler, D. Lüthi, C. Osuna, C. Schär, T. C. Schulthess, and H. Vogt, "Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0," *Geoscientific Model Development*, vol. 11, no. 4, pp. 1665–1681, 2018.

[5] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, "Exascale Deep Learning for Climate Analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 51:1–51:12.

[6] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, Oct 2014.

[7] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters With NVIDIA GPUs," in *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 2013, pp. 80–89.

[8] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, and D. K. Panda, "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)*, Dec 2014, pp. 1–10.

[9] K. S. Khorassani, C.-H. Chu, H. Subramoni, and D. K. Panda, "Performance Evaluation of MPI Libraries on GPU-enabled OpenPOWER Architectures: Early Experiences," in *International Workshop on OpenPOWER for HPC (IWOPH 19) at the 2019 ISC High Performance Conference*, 2018.

[10] S. S. Sharkawi and G. A. Chochia, "Communication protocol optimization for enhanced GPU performance," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 9:1–9:9, 2020.

[11] X. Luo, W. Wu, G. Bosilca, T. Patinyasakdikul, L. Wang, and J. Dongarra, "ADAPT: An Event-based Adaptive Collective Communication Framework," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: ACM, 2018, pp. 118–130.

[12] C.-H. Chu, X. Lu, A. A. Awan, H. Subramoni, B. Elton, and D. K. Panda, "Exploiting Hardware Multicast and GPUDirect RDMA for Efficient Broadcast," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 575–588, March 2019.

[13] Y. Ueno and R. Yokota, "Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2019, pp. 430–439.

[14] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "STELLA: A Domain-Specific Tool for Structured Grid Methods in Weather and Climate Models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807627

[15] O. Pearce, "Experiences Using CPUs and GPUs for Cooperative Computation in a Multi-Physics Simulation," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, ser. ICPP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3229710.3229711

[16] T. Okamoto, H. Takenaka, T. Nakamura, and T. Aoki, *Accelerating Large-Scale Simulation of Seismic Wave Propagation by Multi-GPUs and Three-Dimensional Domain Decomposition*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 375–389.

[17] G. Shi, S. Gottlieb, A. Torok, and V. Kindratenko, "Design of MILC Lattice QCD Application for GPU Clusters," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 363–371.

[18] H. Wang, S. Potluri, M. Luo, A. Singh, X. Ouyang, S. Sur, and D. Panda, "Optimized Non-contiguous MPI Datatype Communication for GPU Clusters: Design, Implementation and Evaluation with MVAPICH2," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, Sept 2011, pp. 308–316.

[19] J. Jenkins, J. Dinan, P. Balaji, N. F. Samatova, and R. Thakur, "Enabling Fast, Noncontiguous GPU Data Movement in Hybrid MPI+GPU Environments," in *2012 IEEE International Conference on Cluster Computing*, Sept 2012, pp. 468–476.

[20] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N. F. Samatova, and R. Thakur, "Processing MPI Derived Datatypes on Noncontiguous GPU-Resident Data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2627–2637, Oct 2014.

[21] R. Shi, X. Lu, S. Potluri, K. Hamidouche, J. Zhang, and D. Panda, "HAND: A Hybrid Approach to Accelerate Non-contiguous Data Movement Using MPI Datatypes on GPU Clusters," in *43rd International Conference on Parallel Processing (ICPP)*, Sept 2014, pp. 221–230.

[22] W. Wu, G. Bosilca, R. vandeVaart, S. Jeaugey, and J. Dongarra, "GPU-Aware Non-contiguous Data Movement In Open MPI," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 231–242.

[23] C.-H. Chu, K. Hamidouche, A. Venkatesh, D. S. Banerjee, H. Subramoni, and D. K. Panda, "Exploiting Maximal Overlap for Non-Contiguous Data Movement Processing on Modern GPU-Enabled Systems," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 983–992.

[24] C.-H. Chu, J. M. Hashmi, K. S. Khorassani, H. Subramoni, and D. K. Panda, "High-Performance Adaptive MPI Derived Datatype Communication for Modern Multi-GPU Systems," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 267–276.

[25] J. M. Hashmi, C.-H. Chu, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "FALCON-X: Zero-copy MPI Derived Datatype Processing on Modern CPU and GPU Architectures," *Journal of Parallel and Distributed Computing (JPDC)*, 2020.

[26] Zhang, Lingqi and Wahib, Mohamed and Matsuoka, Satoshi, "Understanding the overheads of launching cuda kernels," in *ICPP'19 Poster*, 2019.

[27] Justin Luitjens, "CUDA Streams: Best Practices and Common Pitfalls," NVIDIA GPU TECHNOLOGY CONFERENCE, 2014, Accessed: August 16, 2020. [Online]. Available: http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf

[28] NVIDIA, "NVIDIA GPUDirect," https://developer.nvidia.com/gpudirect, 2011, Accessed: August 16, 2020.

[29] Message Passing Interface Forum, http://www.mpi-forum.org/, Accessed: August 16, 2020.

[30] Mark Harris and Kyrylo Perelygin, "Cooperative Groups: Flexible CUDA Thread Programming," NVIDIA Developer Blog, 2017, Accessed: August 16, 2020. [Online]. Available: https://devblogs.nvidia.com/cooperative-groups/

[31] Network-Based Computing Laboratory, "MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE," http://mvapich.cse.ohio-state.edu/, 2001, Accessed: August 16, 2020.

[32] T. Schneider, R. Gerstenberger, and T. Hoefler, "Micro-Applications for Communication Data Access Patterns and MPI Datatypes," in *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, vol. 7490. Springer, Sep. 2012, pp. 121–131.

[33] J. Burmark, https://github.com/LLNL/Comb, 2019.

[34] Davide Rossetti, "NA fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology," https://github.com/NVIDIA/gdrcopy, Accessed: August 16, 2020.

[35] J. L. Träff, R. Hempel, H. Ritzdorf, and F. Zimmermann, "Flattening on the Fly: efficient handling of MPI derived datatypes," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, E. Luque, and T. Margalef, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 109–116.

[36] W. Gropp, T. Hoefler, R. Thakur, and J. L. Träff, "Performance Expectations and Guidelines for MPI Derived Datatypes," in *Recent Advances in the Message Passing Interface*, Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 150–159.

[37] Byna, Gropp, X.-H. Sun, and Thakur, "Improving the performance of MPI derived datatypes by optimizing memory-access cost," in *2003 Proceedings IEEE International Conference on Cluster Computing*, Dec 2003, pp. 412–419.

[38] J. Wu, P. Wyckoff, and D. Panda, "High performance implementation of MPI derived datatype communication over InfiniBand," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, pp. 14–.

[39] R. Ross, R. Latham, W. Gropp, E. Lusk, and R. Thakur, "Processing MPI Datatypes Outside MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 42–53.

[40] G. Santhanaraman, J. Wu, and D. K. Panda, "Zero-Copy MPI Derived Datatype Communication over InfiniBand," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 47–56.

[41] S. D. Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, and T. Hoefler, "Network-Accelerated Non-Contiguous Memory Transfers," 2019.