HAND: A Hybrid Approach to Accelerate Non-contiguous Data Movement using MPI Datatypes on GPU Clusters

Rong Shi, Xiaoyi Lu, Sreeram Potluri, Khaled Hamidouche, Jie Zhang and Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University
Email: {shir, luxi, potluri, hamidouc, zhanjie, panda}@cse.ohio-state.edu

***Abstract*—Increasing number of MPI applications are being ported to take advantage of the compute power offered by GPUs. Data movement continues to be the major bottleneck on GPU clusters, more so when data is non-contiguous, which is a common case in scientific applications. Existing techniques to optimize MPI datatype processing to improve performance of non-contiguous data movement handle only certain data patterns efficiently while incurring overheads for the others. In this paper, we first propose a set of optimized techniques to handle different MPI datatypes. Next, we propose a novel framework (HAND) that enables hybrid and adaptive selection among different techniques and tuning to achieve better performance with all datatypes. Our experimental results using modified DDTBench suite demonstrate up to 98% reduction in datatype latency. We also apply datatype aware design on an N-Body particle simulation application. Performance evaluation of this application on a 64 GPU cluster shows that our proposed approach can achieve up to 80% and 54% increase in performance by using struct and indexed datatypes compared to the existing best design. To the best of our knowledge, this is the first attempt to propose a hybrid and adaptive solution to integrate all existing schemes to optimize arbitrary non-contiguous data movement using MPI datatypes on GPU clusters.**

***Keywords*-MPI, GPU, CUDA, Datatype**

I. Introduction

It is becoming increasingly common for High Performance Computing clusters to use accelerators, such as NVIDIA GPUs, to push their peak compute capabilities. This trend is evident in the TOP500 list released in November 2013, where 53 systems make use of accelerator technology [1]. An increasing number of scientific applications are also being ported to take advantage of such clusters [2, 3]. Many of these applications work on multi-dimensional data which gives rise to inter-process communication involving non-contiguous data (e.g. multi-grid (MG) on a sequence of meshes in fluid dynamics). Table I summarizes a set of representative applications which use non-contiguous datatype. For these applications, data movement continues to be a primary bottleneck on GPU clusters and the movement of non-contiguous data poses a bigger challenge. GPU programming platforms, like CUDA, offer memory copy APIs to move multi-dimensional data between GPUs and CPUs for convenience. However these yield poor performance, especially for sparse data distribution, such as SPECFEM3D_oc kernel in geophysical science. Most state-of-the-art applications use hand-coded CUDA kernels to pack data on the device before communicating it between processes. This approach gives better performance but has several drawbacks. It increases the effort required by the application developers and it adds the perpetual overhead of tuning and maintaining the code for different existing architectures and for newer architectures as they emerge. Such effort is substantially higher when data packing and movement steps have to be overlapped for better performance.

Application	Testname	Datatype	Access Pattern
Fluid Dynamics	NAS_MG_y	vectors	3D face exchange in y direction
Geophysical Science	SPECFEM3D_oc SPECFEM3D_cm	indexed struct on indexed	unstructured exchange of data for diverse earth layers
Atmospheric Science	WRF_y_sa	struct on subarray	struct of 2D/3D/4D face exchanges in y direction
Quantum Chromodynamics	MILC_su3_zd	nested vectors	4D face exchange in z direction
Metereological Science	COSMO	3D subarray	halo-update
Stencil Code	Jacobi	2D subarray	5 point 2D Stencil

Table I: A representative set of applications using various datatypes

User-defined datatypes in MPI enable developers to represent non-contiguous data and to use any of the MPI communication routines to move this data between and among processes. MPI runtimes can take care of handling the movement of non-contiguous data efficiently, while hiding all the complexity from the user. CUDA-aware MPI libraries have enabled applications to use standard MPI interfaces to move data between NVIDIA GPUs like they would move data between hosts [4, 5]. They make it easier for developers to express communication in CUDA+MPI applications. This approach also enables the MPI runtimes to optimize data movement between GPUs using advanced features (such as GPUDirect RDMA and CUDA IPC) and techniques (such as pipelining) which are hard for application developers to use and maintain directly in applications [6, 7].

*1) **Motivation**:* Table II summarizes the existing efforts to optimize datatype-based communication on GPU clusters.

*This research is supported in part by National Science Foundation grants #OCI-0926691, #OCI-1148371 and #CCF-1213084.

Design	Datatypes	Enhancements	[8]	[9]	[10]	This Paper
Direct Transfer	all	active scheduling	N	N	N	Y
Targeted Kernels	vector	2D thread block	N	Y	N	Y
	subarray	1/2/3/4D targeted	N	N	N	Y
	indexed_block	new targeted	N	N	N	Y
	the above three	adaptive tuning	N	N	N	Y
Transformation	others		N	Y	N	Y
		automatic switch	N	N	N	Y
		adaptive tuning	N	N	N	Y
Host Bypass	others		N	N	Y	Y
		automatic kernel selection	N	N	N	Y
		adaptive tuning	N	N	N	Y
HAND	all	automatic scheme selection	N	N	N	Y

Table II: Comparison with Existing Work

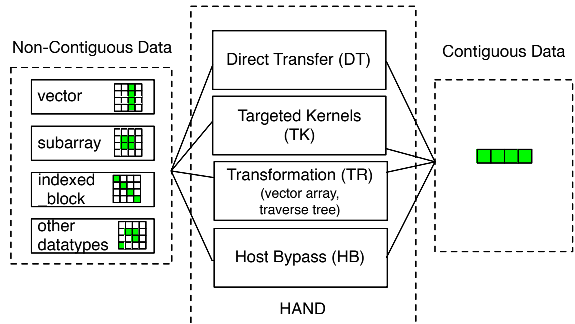


Figure 1: Overview of the HAND Framework

Some of them have used CUDA memory copy APIs to direct transfer multi-dimensional data on the device while pipelining data movement between GPU device and host memory with transfer over the network. There have also been efforts to use CUDA targeted kernels to pack/unpack vector and 3D subarray in the MPI library [8]. This approach ignores the specific cases of 1D/2D/4D subarray and indexed_block. Also, without 2D thread block support, vector kernels cannot fully explore the parallelism on GPUs.

Furthermore, Wang et al. [9] proposes a transformation scheme to convert non-contiguous data into vectors, before parallelizing the packing of each vector, using a CUDA kernel. Without adaptive tuning, this scheme exhibits bad flexibility for different shapes of datatype. Also, this scheme discards useful information provided by MPI datatypes, applying a uniform transformation for all shapes of data. This introduces inefficiencies (e.g. subarray). And when datatype has an irregular distribution (e.g. indexed with non-uniform displacement), the process of transforming them into vectors can result in a large number of small vectors and kernel launches, thus hurting performance. Jenkins et al. [10] represents datatype information in a pattern conducive to parallel access on GPUs and carry out both pre-processing and data packing directly on the device. When data density is high, packing using GPU kernels can incur unnecessary overheads compared to direct transfer between CPU and GPU. Similarly, when datatype has a regular distribution, the pre-processing on the GPU can incur an overhead. A better choice is to actively schedule the direct transfer to avoid the overhead of kernel invocation. Further, rather than adaptive tuning, assigning one thread per element, as is done in their effort, may not be optimal for all data shapes (e.g. vector). It is important that a hybrid MPI datatype processing framework should consider the information available about the structure of data (e.g. indexed_block) and automatically choose a scheme to avoid above limitations appropriately. Such a framework can optimize the datatype packing/unpacking transparently, thus provide good programming efficiency for application users. This paper focuses on such a design and its evaluation.

*2) **Contribution**:* In this paper, we present a comprehensive framework (shown in Figure 1), Hybrid Approach to Accelerate Non-contiguous Data Movement (HAND), for efficient communication using MPI datatypes on GPU clusters. We propose designs to improve the performance of different datatype processing and packing techniques spanning **Direct Transfer**, **Targeted Kernels**, **Datatype transformation**, and **Host Bypass**. We also propose a three-layer framework that chooses among these alternatives and tunes the chosen technique for improved performance. We make the following key contributions through this paper:

1) Propose a HAND framework, fully integrated with MVAPICH2 library [4], that facilitates the dynamic selection and tuning of these designs to support non-contiguous data movement on GPU clusters using arbitrary user-defined datatypes.
2) Propose optimizations for processing and packing user-defined MPI datatypes on NVIDIA GPUs using targeted kernels and datatype transformation.
3) Propose a novel GPU kernel-based host bypass design to efficiently pack/unpack arbitrary non-contiguous datatypes.
4) Present a quantitative evaluation of the proposed framework across clusters with diverse GPU configurations using micro benchmarks, micro application kernels and applications.

The rest of the paper is organized as follows. In Section II, we present the overview of proposed HAND design. We then introduce and analyze the redesign of targeted kernels and optimizations of transformation, and the design of host bypass in Section III and IV, respectively. In Section V, we discuss a case study of redesigning N-Body application using HAND. We present experimental results in SectionVI. Finally, we summarize the related work in Section VII and conclude the paper in Section VIII.

II. Proposed Framework of HAND

A. *Design Overview of HAND*

Figure 2 provides an overview of the proposed HAND framework within MVAPICH2 library. HAND integrates all

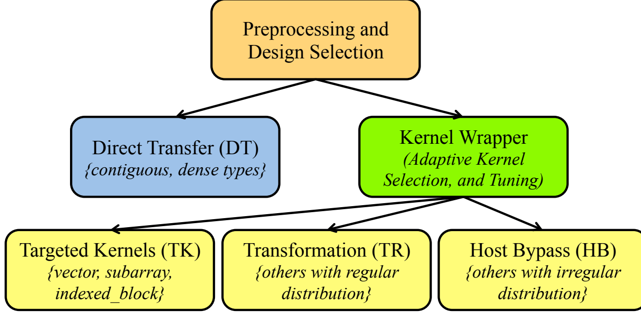


Figure 2: Detailed Overview of the HAND Framework

four schemes and dynamically selects the most suitable one based on the characteristic of the datatype. We present the three-layer HAND design in a top-down approach.

All modules of HAND are within the MPI communication routines. First, preprocessing and design selection module receives the datatype information from application. Inherited from the dataloops design in MPICH [11], in MVAPICH2 library, preprocessing module converts datatype information to iov structure representing tree structure for arbitrary datatype, which consists of a list of $\langle start_address, block_length \rangle$ pairs for each contiguous subblock defined by datatype primitives. Then, HAND chooses the proper scheme based on the shape of datatypes represented in iov structure. HAND integrates four schemes and represents them as four modules:

Direct Transfer module (DT) applies CUDA API to copy each subblock of the datatype. Compared to other kernel-based scheme, DT avoids the overhead of kernel invocation which is the key consideration for contiguous and dense datatypes packing/unpacking. Thereby, rather than treating DT as a last choice, HAND treats DT as a good candidate for contiguous and dense datatypes.

Targeted Kernel module (TK) handles specific shapes of datatypes. By fully exploring the iov information of these specific datatypes, targeted kernels are designed to manipulate these datatypes in the most efficient manner.

Transformation module (TR) represents datatype information in an intermediate structure (e.g. vector array) and then packs/unpacks each intermediate structure using targeted kernels (e.g. vector).

Host Bypass module (HB) bypasses the transformation of iov structure and offloads the management of iov structure and memory copy to GPU kernels. Compared to TR, HB avoids the large number of kernel invocations for datatype with irregular distribution.

HAND provides targeted kernels for (h)vector, subarray and (h)indexed_block; and for other non-contiguous datatypes, HAND chooses the more suitable scheme between transformation and host bypass based on the shapes of datatype. In the HAND framework, other datatypes include irregular one-layer datatypes (e.g. indexed and struct), and hierarchical datatypes (e.g. nested vectors).

Besides the basic type and count defined in MPI datatypes, targeted kernels only need a subset of the whole iov structure because of the regularity of corresponding datatypes.

Except for the direct transfer, each scheme has a respective kernel wrapper which optimizes thread block and configures kernel parameters (e.g. splitting memory between L1 cache and shared memory). Both preprocessing routines and kernel wrappers are executed on CPU hosts, and all corresponding kernels are launched on GPU devices.

III. Redesign of Targeted Kernels and Optimizations of Transformation Scheme

In this section, we discuss redesign of targeted kernels and optimizations of transformation scheme. Figure 3 shows the design for respective optimizations. In our experiment, we test the targeted kernels (TK) with classic ping-pang latency on hvector, 4D subarray and indexed_block datatypes.

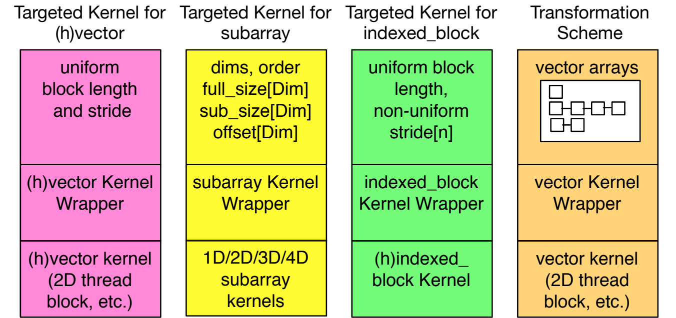


Figure 3: Design Overview of Targeted Kernels and Transformation Scheme

For regular datatypes, we extend previous targeted kernels [8, 9] in terms of redesign of kernels and general optimization by adaptive tuning.

A. Redesign of (h)vector Kernels

Vector datatype specifies strided blocks of data of oldtype which is useful for Cartesian arrays. Hvector datatype creates non-unit strided vectors and is useful for composition (e.g. nested vector). Hvector is identical to vector, except that the stride is given in bytes, rather than in elements. As shown in Figure 3, (h)vector has uniform block length and stride which provides opportunity for targeted kernels.

Previous work [8, 9] provides targeted kernel for (h)vector with static 1D/2D thread block. 1D thread block incurs performance loss for vector with large block length (e.g. dense vector). Without adaptive tuning, static 2D thread block losses flexibility for different shapes of datatypes. To overcome these limitations, HAND utilizes adaptive 2D thread block for both vector and hvector kernels. 2D thread block enhances the thread level parallelism in terms of count and block length. And the adaptive tuning dynamically

selects the optimal thread block based on the shapes of datatypes, thereby further improves the performance for various datatypes.

Figure 4 evaluates the targeted vector kernel among 1D thread block (TK-vector-1D), 2D thread block (TK-vector-2D) and new adaptive thread block (TK-vector-Opt). This experiment uses fixed count of 32 and increasing block lengths. We just select the hvector because vector shares the same routine. Compared to TK-vector-2D, TK-vector-Opt exchanges the mapping order of thread block to provide better thread access locality. In Figure 4, TK-vector-Opt achieves an average of 77% and 5.5%, and up to 82% and 13.6% decrease in latency compared to TK-vector-1D and TK-vector-2D, respectively.

B. Redesign of subarray Kernels

Subarray datatype specifies subarray of n-dimensional array by using displacements and subsizes. As shown in Figure 3, subarray routine needs array storage order, dimension, as well as full sizes, sub sizes and displacements on each dimension. Previous work [8] uses targeted kernels for 3D subarray and applies direct transfer on other dimenional subarrays. To make up this inefficiency, HAND provides targeted kernels to handle 1D, 2D and 4D subarrays separately. For other dimensional subarrays, either transformation or host bypass scheme is selected. Furthermore, since application programmers can specify either MPI_C_ORDER or MPI_FORTRAN_ORDER for subarrays, our design offers good programming efficiency by supporting both orders in all targeted subarray kernels.

Figures 5(a) and 5(b) show the cases of 4D subarray with various innermost and outermost dimension sizes. To prevent overlapping, we leave the evaluation of 2D/3D subarray to later micro-application-level evaluation. For 4D subarray with various dimension sizes, new design provides two targeted kernels: “LoopT” and “LoopX”. Suppose the dimension sizes of 4D subarray are represented as X, Y, Z and T in order, LoopT design maps 3D thread block of the kernel onto X, Y, Z and each thread loop on innermost T dimension. LoopX design maps the 3D thread block onto Y, Z, T and loop on outermost X dimension. Both kernels achieve an average of above 85% improvement compared to direct transfer (DT) which uses CUDA memory copy API, and TK-subarray-LoopX exhibits additional up to 77% improvement compared to TK-subarray-LoopT for innermost case. For outermost case, TK-subarray-LoopT shows an average of 13% performance improvement with dimension size of X larger than 32; while for small sizes of X, TK-subarray-LoopX still achieves an average of 15% improvement. Based on these observations, HAND introduces experienced switch threshold for 4D subarray with large outermost dimension.

C. Redesign of (h)indexed_block Kernels

Indexed_block datatype is used to pull irregular subsets of data from a single array with the uniform block length. As shown in Figure 3, unlike vector which only needs uniform displacement, indexed_block needs the whole non-uniform displacement arrays. Hindexed_block is identical to indexed_block, except that the stride is given in bytes, rather than in elements. Previous design [9] treats the indexed_block datatype as the general datatype and applies the transformation scheme. As discussed in Section I, the transformation scheme delivers significant performance degradation if indexed_block has non-uniform displacement. To explore the property of same block length for indexed_block, new targeted kernel applies the simplified design of host bypass kernel for both indexed_block and hindexed_block, in which prefix-sum calculation is avoided because of uniform block length. Targeted kernel performs much better than transformation schemes in most scenarios by avoiding large number of kernel invocations. In Figure 6, compared to direct transfer (DT), targeted kernel (TK-idxblk) shows an average reduction in latency of 61% and 75% for cases of fixed block length and fixed count, respectively.

D. Optimizations of Transformation Scheme

As shown in Figure 3, transformation scheme converts non-contiguous datatypes into vector arrays and launches targeted vector kernels to pack/unpack each vector array. As discussed in Section II, HAND fully integrates four schemes. For other irregular datatypes, rather than fully relying on transformation scheme, the selector dynamically chooses the optimal scheme. Also, selector offers the automatic switch from transformation scheme to host bypass scheme if the former scheme creates too many small vector arrays. To achieve this goal, HAND maps other datatypes with regular distribution onto transformation scheme, while maps the remaining arbitrary datatypes onto host bypass scheme. Therefore, the HAND design can take advantage of the benefit of transformation scheme on irregular datatypes while avoiding performance loss in other cases.

In addition, existing transformation scheme in MVAPICH2 library utilizes vector kernels to pack/unpack arbitrary datatypes after transforming the iov structure to a list of vector arrays. The performance of targeted vector kernel directly decides the efficiency of transformation scheme. Therefore, our redesign of targeted (h)vector kernels, discussed in Section III-A, directly boosts the performance of transformation scheme.

IV. Proposed Design of Host Bypass Scheme

This section covers in depth the design strategies and characterizes the benefit of different strategies with micro benchmarks in a top-down approach: host bypass preprocessing routines, kernel wrappers and GPU packing kernels.

A. Preprocessing Routine

In preprocessing routine, our design passes down the iov information directly onto GPUs. We propose two strategies of copying the iov structure from CPU to GPU memory:

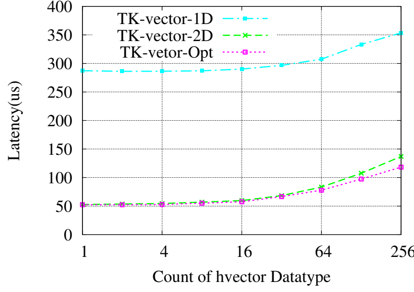
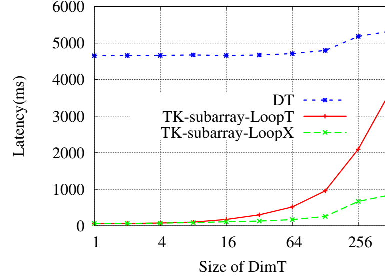
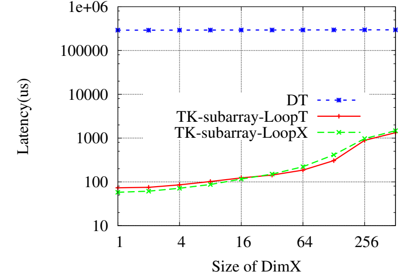


Figure 4: Evaluation of hvetor

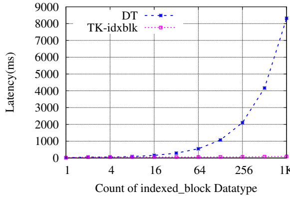


(a) $8 \times 8 \times 8 \times T/2$

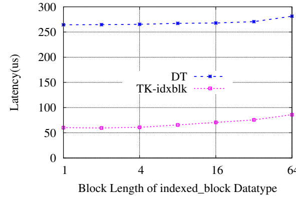


(b) $X/2 \times 8 \times 8 \times 8$

Figure 5: Evaluation of 4D subarray

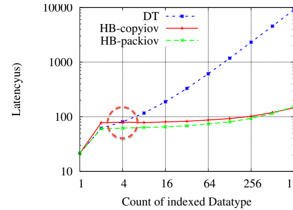


(a) block_length=1, various counts

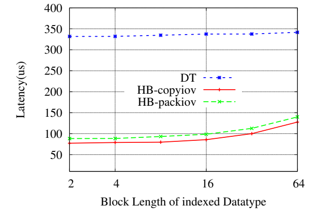


(b) count=32, various block lengths

Figure 6: Evaluation of indexed_block



(a) block_length=1, various counts



(b) count=32, various block lengths

Figure 7: Evaluation of Passing IOV Strategies

"HB-packiov" and "HB-copyiov". HB-packiov strategy allocates pinned device memory that can be accessed by both CPU and GPU and applies dataloop functions to directly pack iov structure onto pinned memory. Then, both CPU kernel wrappers and GPU kernels can access the iov structure. HB-copyiov strategy allocates general device memory so that only GPU kernels can access, yet with higher speed. Before launching the kernels, the iov structure stored on CPU memory should be copied to allocate device memory. HB-packiov avoids additional CUDA memory copy and performs better for datatypes with small block lengths. However, with larger block lengths, the benefit of faster memory access compensates the additional memory copy for iov structure in case of HB-copyiov.

In our experiment, we test the ping-pang latency on indexed datatype with non-uniform stride. Figure 7 evaluates the performance of these two strategies. Compared to direct transfer (DT), both strategies exhibit up to 99% improvement with count size larger than 4 for the case shown in Figure 7(a), and achieve an average of 70% improvement for the case shown in Figure 7(b). Rather than treating DT as the last choice in transformation scheme [9], the HAND framework introduces the experienced switch threshold of 4 to activate DT for dense datatypes. Furthermore, Compared to HB-packiov, HB-copyiov achieves an average improvement of 15% for the case of fixed block length. In contrast, HB-packiov gains an average of 12% benefit compared to HB-copyiov for the case of fixed count. Therefore, HAND dynamically chooses between HB-packiov and HB-copyiov for various shapes of datatypes.

Besides, there is startup overhead of allocating the device buffer used to store iov structure. In our experiment, this overhead is around 100us and 200us for HB-copyiov and HB-packiov, respectively. Considering the overhead, the threshold value of count will be 16 and 64 for HB-copyiov and HB-packiov, respectively, for the case of fixed block length. In fact, this startup overhead can be easily amortized with multiple runs of the application even with small counts. And this overhead can be ignored for datatypes with large count even the application executes only once.

B. Kernel Wrapper

First, kernel wrapper optimizes the performance of datatype with non-power-of-two counts. Since the maximum thread within one block is restricted to 1024, careful manipulation of the threads is necessary to achieve good performance. Rather than using external math library functions, we resort to fast bit operations to calculate the nearest power-of-two values for counts. Hence, we can minimize the thread block size for datatype with smaller counts. Also, the saving threads can be mapped to the dimension of block length, which further improves the parallelism. Combined with runtime dynamic thread block tuning based on the shapes of datatypes, these optimization techniques can be applied to other schemes and benefit different datatypes in various ways.

Figure 8 shows that, compared to the original static thread block (HB-Static), dynamic thread block tuning (HB-Dynamic) gains 2% to 12% improvement for datatypes with irregular counts. For datatypes with small counts, this technique shrinks the size of thread block and thus reduces

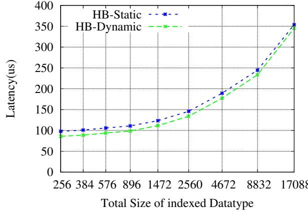


Figure 8: Evaluation of Adaptive Thread Block Tuning

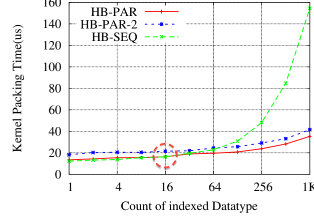


Figure 9: Evaluation of Automatic Kernel Selection

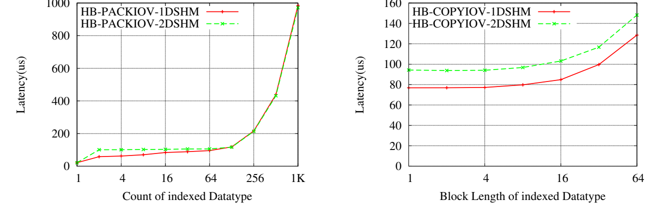

(a) block_length=1, various counts (b) count=32, various block length

Figure 10: Evaluation of IOV Device Shared Memory

the overhead of thread launch and synchronization with up to 12% improvement. And the minor improvement for datatypes with large counts comes from the speedup of last kernel launch with suitable thread block size.

Furthermore, kernel wrapper optimizes thread block mapping. To make it concise, we use $tidx$ and $tidy$ to illustrate first and second thread block dimensions. Previous design maps $tidx$ to count and $tidy$ to block length while new design reverses the mapping order. This reordered mapping facilitates successive memory accesses in $tidy$ direction to explore better data locality.

C. GPU Packing/Unpacking Kernels

The host bypass packing/unpacking kernels include two parts: prefix-sum calculation and thread level memory copy. All prefix-sum calculations [12] on an array of data are commonly known as *scan* and we will use *scan* for the remainder of the paper. Our new design includes two *scan* strategies: sequential and parallel. The sequential scan requires $O(N)$ steps assuming N is the datatype count. In contrast, the parallel one needs $O(logN)$ steps. Theoretically, parallel scan is better than the sequential one. However, the overhead of more instructions and thread synchronization counteracts the benefits of parallelism for datatypes with small counts. Therefore, host bypass kernel wrapper uses experienced threshold to automatically switch between two strategies. The second part of the kernels is the parallel memory copy and both kernels adopt the same 2D thread block where each GPU thread is responsible for the memory copy of one or more datatype primitives. Since GPU packing and unpacking kernels are symmetric, we only show the skeleton of GPU packing kernels in Algorithm 1 and 2.

The time complexity of parallel and sequential kernels can be expressed as indicated in Equations 1 and 2, respectively:

$$T_par = P * log(count) + iter + P_overhead \quad (1)$$

$$T_seq = S * count + iter + S_overhead \quad (2)$$

P and S represent the average number of operations for each GPU thread in parallel scan and sequential scan respectively. Correspondingly, $P_overhead$ and $S_overhead$ represents the overhead apart from scan and memory copy operations for each thread (e.g. synchronization overhead).

Algorithm 1: GPU PACKING KERNEL WITH PARALLEL SCAN

```
   Input: dstbuf, iov_structure, count
   Output: desbuf
1  __shared__size_t prefixsum[1024];
2  i ← threadIdx.x;
3  j ← threadIdx.y;
4  offset ← 1;
5  // exclusive parallel prefix-sum calculation
6  prefixsum[i * blockDim.y + j] ←
   iov_structure[j].iov_len;
7  for int k = (blockDim.y) >> 1; k > 0; k >>= 1 do
8      // build sum in place up the tree
9      offset <<= 1
10 end
11 for int k = 1; k < (blockDim.y); k <<= 1 do
12     offset >>= 1
13     __syncthreads();
14     // traverse down tree and build scan
15 end
16 __syncthreads();
17 if j < count then
18     iter ← iov_structure[j].srclen/blockDim.x;
19     for k = 0; k <= iter; k + + do
20         if i < iov_structure[j].srclen then
21             dstbuf[prefixsum[i * blockDim.y + j] + i]
22             ← *((iov_structure + j).iov_base + i);
23         end
24         i ← i + blockDim.x;
25     end
26 end
```

We then evaluate GPU packing kernels with sequential and parallel scans. In Figure 9, “HB-PAR” and “HB-SEQ” represent one-step packing kernel with parallel scan and sequential scan, respectively. “HB-PAR-2” represents two-step kernel in which the first kernel implements the parallel scan and the second one implements the memory copy. The HB-PAR achieves a constant 20% improvement than two-step kernel by avoiding the additional copy of intermediate scan results between global memory and shared memory. For

Algorithm 2: GPU PACKING KERNEL WITH SEQUENTIAL SCAN

Input: $dstbuf$, $iov_structure$, $count$
Output: $desbuf$

1 $i \leftarrow threadIdx.x$;
2 $j \leftarrow threadIdx.y$;
3 // Sequential prefixsum calculation
4 **if** $j < count$ **then**
5   // initialize block_length,
6   // source and destination buffer displacements
7   **for** $k = 0; k < j; k++$ **do**
8   // calculate $prefixsum$ for each subblock
9   **end**
10   $iter \leftarrow iov_structure[j].srclen/blockDim.x$;
11   **for** $; k <= iter; k++$ **do**
12   **if** $i < iov_structure[j].srclen$ **then**
13       $dstbuf[prefixsum + i]$
14       $\leftarrow *((iov_structure + j).iov_base + i)$;
15   **end**
16     $i \leftarrow i + blockDim.x$;
17   **end**
18 **end**

large count sizes, HB-PAR gains an average of 41% benefit compared to HB-SEQ. However, for count sizes smaller than 16, HB-SEQ achieves an average of 9% benefit compared to HB-PAR. This is mainly due to the fact that P in equation 1 is larger than S in equation 2, and the benefit of logarithmic scan is hidden.

Further, our design considers two strategies to store the intermediate prefix-sum values used in the kernel. This intermediate shared memory space can be organized in either 1D array or 2D array format. To compare their performance, we use the packing kernel with parallel scan and combine it with one or two dimensional shared memory which stores the intermediate scan values. As shown in Figure 10(a) and 10(b), one dimensional shared memory (HB-PACKIOV-1DSHM) gains an average of 26% and 17% performance improvement for respective cases of datatype. This comes from the fact that indexing on 1D shared memory facilities shared memory access.

V. N-BODY APPLICATION ENHANCEMENT BY HAND

In this section, we propose a case study of N-Body application. We apply datatype aware design for large scale N-Body particles simulation [13] implemented with MPI+GPGPU programming model which is widely used HPC applications. A familiar example is an astrophysical simulation in which each particle represents a galaxy or an individual star, and the particles attract each other through the gravitational force.

The simulation usually executes multiple times and we focus on the analysis of one iteration. Initially, each GPU node gets a subset of particles and uses the GPU kernels to calculate the forces for local particles. Then all GPU nodes exchange data with others by MPI_Allgather communication as shown in Figure 11. After that, each GPU node updates positions and velocities of local particles.

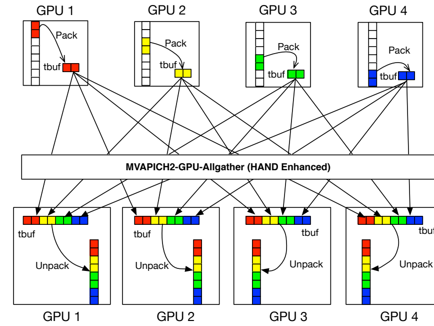


Figure 11: Proposed Datatype Aware Design of N-Body Particles Simulation

To illustrate the benefit of HAND, we extend the definition of particle structure to simple particle and complex particle, and define two corresponding non-contiguous datatypes above them. Both the transformation and HAND help to pack/unpack the non-contiguous datatypes before/after the GPU Allgather communications. In contrast, direct transfer copies the whole structure of particles between CPU and GPU, and relies on CPU Allgather communications.

```
typedef struct {
        double position_x, position_y, position_z ;
        double velocity_x, velocity_y, velocity_z;
        double force_x, force_y, force_z;
        double mass;
        float properties[20];  //complex struct only
        int   charge;
} Particle;
```

We declare struct datatype including all data members above simple particle, and indexed datatype for $force_x$, $force_y$, $force_z$ and $charge$ data members above complex particle, which are the minimum datasets that need to be exchanged. We will show experiment results of our modified design in Section VI-D.

VI. PERFORMANCE EVALUATION

In this section, we describe our experimental testbed and evaluate HAND design with micro application kernels and large scale applications.

A. Experiment Setup

We use two clusters in our experiments and their specifications are in Table III. Most experiments are conducted on Cluster A, and we run N-Body particle simulation on Oakley Cluster with up to 32 GPU nodes where each node has two Fermi GPU accelerators.

Cluster Specs	A	Oakley
CPU Processor	Intel Xeon E5630	Intel Xeon X5650
CPU Clock	2.53GHz	2.66GHz
Node Type	two 4-core sockets	two 6-core sockets
CPU Memory	12 GB	46 GB
GPU Processor	NVIDIA Tesla C2050	NVIDIA Tesla M2070
GPUs per Node	1	2
GPU Memory	3 GB	6 GB
Compilers	gcc 4.4.7	gcc 4.4.7
MPI Library	MVAPICH2 2.0b	MVAPICH2 2.0b
Interconnect	Mellanox IB QDR	Mellanox IB QDR

Table III: Experimental Environment

B. *Evaluation with DDTBench Micro Application*

In this section we evaluate overall HAND design by comparing with direct transfer (DT) and transformation (TR). DT uses MPI_Send/Recv on CPUs and explicitly moves data between CPU memory and GPU memory. We normalize the latency of HAND to 1 and calculate the relative latency for other designs. We choose five representative micro application kernels from DDTBench Suite [14], as summarized in Table I. DDTBench is a suite of micro applications that characterizes parallel scientific applications from different fields of science. By performing a ping-pong benchmark, the DDTBench illustrates the usage of datatype in real applications. We modify these DDTBench micro applications by allocating the data on device memory and use them directly in MPI communication routines. Since many communication patterns and datatypes are widely used in GPGPU applications, our evaluation provides good perspective for real GPGPU applications as well.

To verify the benefit of HAND, we evaluate both one layer and multi-layer datatypes and combine the results in Figure 12. In general, HAND achieves equal or better performance than other schemes. Compared to DT, without additional explicit data movement between CPU and GPU, HAND achieves an average of 66%, 99%, 97%, 43% and 97% performance improvement for NAS_MG_y, SPECFEM3D_oc, WRF_y_sa, MILC_su3_zd and SPECFEM3D_cm, respectively.

Compared to TR, HAND performs equally on WRF_y_sa and MILC_su3_zd and gains an average of 18% improvement on NAS_MG_y. In the cases of SPECFEM3D_oc and SPECFEM3D_cm, TR degrades to DT. This is largely caused by the non-uniform displacements of the arbitrary datatypes. In addition, in case of NAS_MG_y, we use 2D cudaMemcpy for DT, yet still delivers worse performance. This is also true for stencil2D in Scalable Heterogeneous Computing benchmark (SHOC) [15], in which the east and west data are non-contiguous and use 2D strided cudaMemcpy to explicit move the data. Therefore, HAND can improve the performance of stencil2D in SHOC as well.

C. *Evaluation with Stencil Computations*

Besides the modified DDTBench suite, we choose two stencil computation kernels to further evaluate the optimization for targeted subarray kernels. First, we evaluate Stencil2D communication kernel on 4 GPUs with two cases: fixed size of 2D array with increasing boundary size, and fixed boundary size of 1 with increasing total size of 2D array. As shown in Figure 13, both transformation (TR) and HAND achieve significant performance improvement compared to direct transfer (DT). Furthermore, the HAND achieves up to 70% improvement compared to TR.

Then, we evaluate the stencil3D computation kernel by varying the size on each dimension of the 3D subarray. As shown in Figures 14(a), 14(b) and 14(c), compared to transformation (TR), HAND gains up to 15%, 15% and 22% reduction in latency with various dimensional sizes, respectively. This is largely due to the reverse thread block mapping and automatic tuning based on different datatype sizes. Also, both designs achieve more than 86% latency reduction compared to application level packing/unpacking routines (DT). This is because Stencil3D kernel uses MPI_Isend and MPI_Irecv asynchronous functions and HAND has been fully incorporated into the pipeline design [7], which pipelines the packing and unpacking with asynchronous data chunk communication.

D. *Evaluation with N-Body Application*

In this section, we evaluate large scale N-Body application proposed in section V. Figure 15(a) and Figure 15(b) show the total execution time of N-Body simulation with 200 iterations, in terms of simple particle and complex particle, respectively. We fix the number of particles to 131,072 (128K) on each GPU node, and evaluate the weak scalability of two cases among direct transfer (DT), transformation (TR) and HAND. For simple particles using struct datatype, both TR and HAND outperform DT, and on average, HAND exhibits 2% benefit compared to TR. However, due to large number of kernel invocations, TR shows poor performance for complex particle using indexed datatype with non-uniform stride. In contrast, HAND exhibits up to 55% performance improvement compared to DT.

VII. Related Work

In HPC area, efficient processing and use of MPI derived datatypes have been explored by many researchers [16–18]. Making MPI libraries GPU-aware and extending MPI to support communication on GPU clusters have also been in focus for several years [6, 7]. Some researchers have also explored optimization of MPI datatypes processing when data is stored on GPUs. The Physis approach proposed by Maruyama et. al [19] defines an implicit parallel programming model specifically for stencil computations on GPU clusters. Similarly, Bianco et. al. [20] proposes a generic library for stencil computations by optimizing targeted nested vector and subarray datatype packing and unpacking routines at the application level. cudaMPI and glMPI proposed by Lawlor et. al [21] define MPI-like message passing interface to support both point-to-point and collective operations on contiguous and non-contiguous data. However, these work

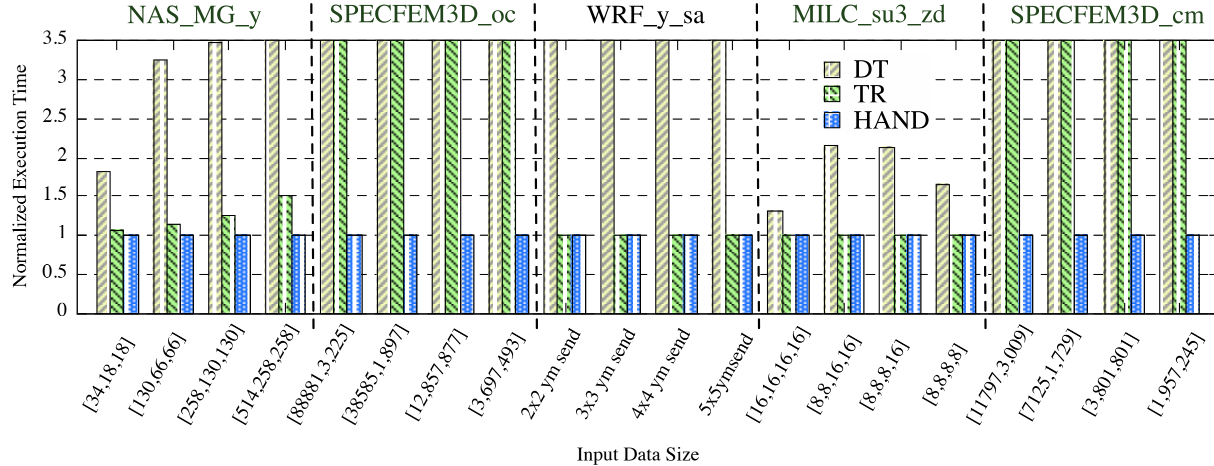
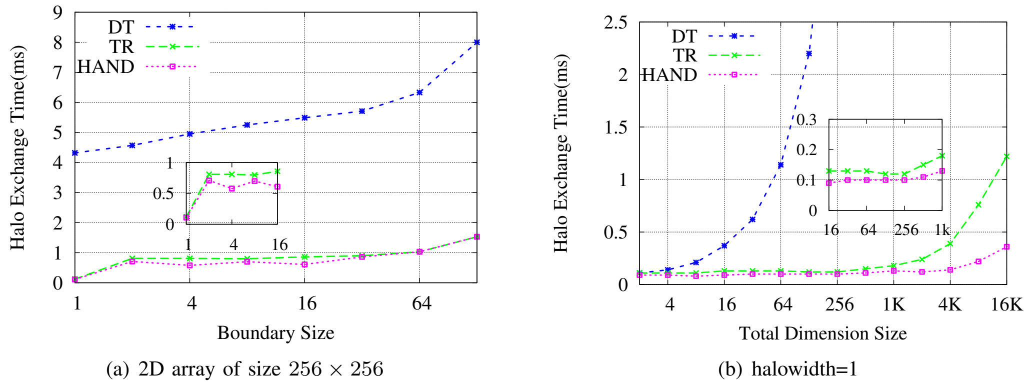
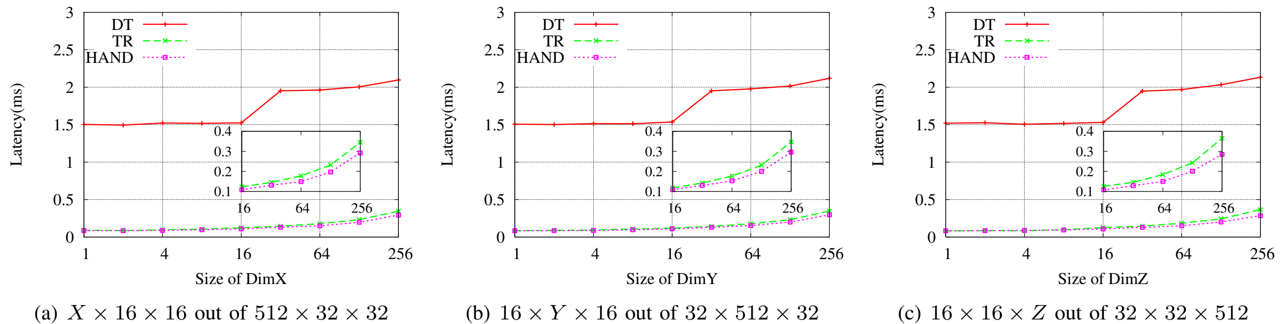


Figure 12: Evaluation of DDTBench Micro-Application Kernels



(a) 2D array of size 256×256 (b) halowidth=1

Figure 13: Evaluation of Stencil2D Computation Kernel



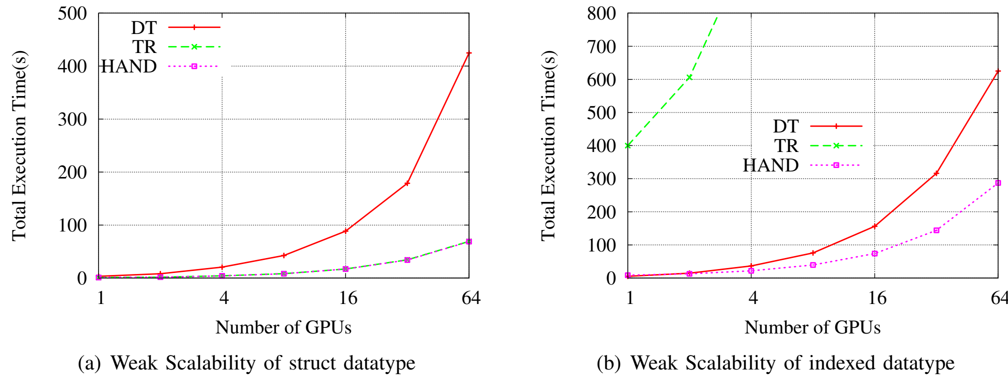
(a) $X \times 16 \times 16$ out of $512 \times 32 \times 32$ (b) $16 \times Y \times 16$ out of $32 \times 512 \times 32$ (c) $16 \times 16 \times Z$ out of $32 \times 32 \times 512$

Figure 14: Evaluation of Stencil3D Application Kernel

focus on particular application scenarios and do not cover the wide range of non-contiguous data transfer patterns found in applications. Jenkins et. al. proposes a tree-based representation that enables efficient storage and access of datatype information on GPUs. They also propose a paralleled datatype packing algorithm [10]. Wang et al. proposes transformation approach, which converts every datatype into a group of vectors and uses kernels to parallelize the packing of each of these vectors [9]. However, both approaches incur inefficiency in handling different shapes of non-contiguous data as explained in Section I.

VIII. Conclusion and Future Work

This paper proposes a novel HAND framework to efficiently pack and unpack non-contiguous data on GPUs. With the new host bypass design, the HAND framework can handle irregular non-contiguous datatypes more efficiently. Also, the HAND framework seamlessly integrates other schemes so as to exhibit equal or better performance. The



(a) Weak Scalability of struct datatype

(b) Weak Scalability of indexed datatype

Figure 15: Evaluation of N-Body Particles Simulation

experimental results exhibit sustained scalability of HAND. Our experimental results across two platforms with different GPU configurations validate the portability and stability of HAND. In the future, we plan to explore HAND with new GPU architectures (e.g. Kepler GPUs) and AMD GPUs with OpenCL programming model. Support for the HAND framework will be available in future MVAPICH2 releases.

REFERENCES

[1] Top500, http://www.top500.org.

[2] A. Nukada, Y. Maruyama, and S. Matsuoka, "High performance 3-d fft using multiple cuda gpus," in *Proceeding of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*, London, UK, 2012.

[3] R. Shi, S. Potluri, K. Hamidouche, X. Lu, K. Tomko, and D. Panda, "A scalable and portable approach to accelerate hybrid hpl on heterogeneous cpu-gpu clusters," in *Proceeding of CLUSTER (CLUSTER'13)*, Indianapolis, Indiana, USA, 2013, pp. 1–8.

[4] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/.

[5] OpenMPI: Open Source High Performance Computing, http://www.open-mpi.org/.

[6] 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 *Proceeding of International Conference on Parallel Processing (ICPP'12)*, Pittsburgh, PA, USA, 2012.

[7] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters," *Comput. Sci.*, no. 3-4, Jun. 2011.

[8] 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 *Proceeding of Cluster Computing (CLUSTER'11)*, Austin, Texas, USA, 2011.

[9] 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*, 2013.

[10] 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*, 2013.

[11] R. B. Ross, N. Miller, and W. Gropp, "Implementing Fast and Reusable Datatype Processing," in *PVM/MPI*. Springer Berlin Heidelberg, 2003.

[12] M. Harris, S. Sengupta, and J. D. Owens, "Parallel Prefix Sum (Scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, 2007.

[13] BWSIP, http://www.shodor.org/petascale/materials/hybrid/code.

[14] T. Schneider, R. Gerstenberger, and T. Hoefler, "Micro-Applications for Communication Data Access Patterns and MPI Datatypes," in *Proceeding of 19th European conference on Recent Advances in the Message Passing Interface (EuroMPI'12)*, Vienna, Austria, 2012.

[15] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *Proceeding of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*, Pittsburgh, PA, USA, 2010.

[16] T. Schneider, F. Kjolstad, and T. Hoefler, "MPI Datatype Processing Using Runtime Compilation," in *Proceeding of the 20th European MPI Users' Group Meeting (EuroMPI'13)*, Madrid, Spain, 2013.

[17] W. Gropp and T. Hoefler and R. Thakur and J. L. Traeff, "Performance Expectations and Guidelines for MPI Derived Datatypes," in *Proceeding of 18th European conference on Recent Advances in the Message Passing Interface (EuroMPI'11)*, Santorini, Greece, 2011.

[18] T. Hoefler and S. Gottlieb, "Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes," in *Proceeding of 17th European MPI users' group meeting conference on Recent advances in the message passing interface (EuroMPI'10)*, Stuttgart, Germany, 2010.

[19] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers," in *Proceeding of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, Seattle, WA, USA.

[20] M. Bianco and U. Varetto, "A Generic Library for Stencil Computations," *CoRR*, 2012.

[21] O. Lawlor, "Message Passing for GPGPU Clusters: CudaMPI," in *Proceeding of Cluster Computing and Workshops (CLUSTER '09)*, New Orleans, Louisiana, USA, 2009.