HDArray: Parallel Array Interface for Distributed Heterogeneous Devices

Hyun Dok Cho[1][**], Okwan Kwon[1], and Samuel P. Midkiff[2]

[1] NVIDIA Corporation, Santa Clara, CA 95050, USA
{hyundokc,okwank}@nvidia.com
[2] Purdue University, West Lafayette, IN 47907, USA
smidkiff@purdue.edu

Abstract. Heterogeneous clusters with nodes containing one or more accelerators, such as GPUs, have become common. While MPI provides inter-address space communication, and OpenCL provides a process with access to heterogeneous computational resources, programmers are forced to write hybrid programs that manage the interaction of both of these systems. This paper describes an array programming interface that provides users with automatic and manual distributions of data and work. Using work distribution and kernel *def* and *use* information, communication among processes and devices in a process is performed automatically. By providing a unified programming model to the user, program development is simplified.

Keywords: Parallel Programming Model · Distributed Shared Memory · Heterogeneous Systems · MPI · OpenCL.

1 Introduction

As GPU programming becomes more mainstream, both large and small scale multi-node systems with one or more GPUs per node have become common. These nodes, however, complicate already messy distributed system programming by adding MPI [12] on top of proprietary host-GPU mechanisms. Developers must maintain two programming models: one for intra-process communication among devices and one for inter-process communication across address spaces.

Several systems have improved the programmability of multi-node systems with accelerators. SnuCL [15,17], dCuda [14], and IMPACC [16] support transparent access to accelerators on different nodes, and PARRAY [7] and Viñas *et al.* [31] provide high-level language abstractions and flexible array representations. Programmers can develop high-performance applications but must manage low-level details of accelerator programming or provide explicit communication code. Partitioned Global Address Space (PGAS) platforms for accelerators, XMP-ACC [21], XACC [24], and Potluri *et al.* [26], relieve programmers from dealing with data distribution, but data is strongly coupled to threads, making performance tuning more difficult. Finally, compiler-assisted runtime systems, Hydra [28] and

---

[**] This work was done while at Purdue University.

OMPD [19], propose a fully automatic approach that allows OpenMP programs to run on accelerator clusters, presenting an attractive alternative for developing repetitive, regular applications, but the distribution of work and data are limited by OpenMP semantics and expressiveness.

In this paper, we describe the Heterogeneous Distributed Array (HDArray) interface and runtime system. HDArray targets program execution on cluster-sized distributed systems with nodes containing one or more accelerators, i.e., devices. Work is done as OpenCL work items, and HDArray provides ways to explicitly and implicitly partition work onto devices.

HDArray also provides a way for the data used by the work on a device to be specified. The data read and written is typically relative to work items and can be specified either using offsets from the work item, or with an absolute specification of the data. HDArray then tracks the data defined and used by each work item, which allows communication to be generated automatically, since, in race-free programs, HDArray knows where the last written copy of a datum is, and who needs that value. Importantly, data is not explicitly distributed and is not bound to, or owned by, a work item, but flows from its defining process and device to the process and device where it is needed.

Finally, HDArray allows work to be repartitioned at any point in the program. This flexibility allows a programmer to optimize the work distribution and its necessary communication without any changes to the kernel code.

To summarize, our contributions are

1. A novel and easy programming model that can efficiently run on distributed heterogeneous devices, enabling flexible work distribution, with data flowing to the work that needs it.
2. A fully automatic runtime communication generation scheme and its implementation.
3. A flexible user interface that allows manual tuning of work distribution for high performance and enabling automatic communication.
4. Experimental results showing good performance and speedups on small clusters with eight nodes and 1 to 32 GPUs.

The rest of the paper is organized as follows. Sections 2, 3, and 4 describe the design and implementation of the HDArray interface. Section 5 presents a performance evaluation of the HDArray runtime system. Section 6 discusses related work, and conclusions and future work follow in Section 7.

2   Design of the HDArray Interface

We now present the design of the HDArray interface and its runtime system as shown in Fig. 1. The central structure, and concept, of the HDArray system is the HDArray itself. The HDArray encapsulates a *host buffer* and *device buffer* by keeping necessary states for communication among processes and kernel computation. The HDArray system provides a collection of APIs and annotations that the programmer uses to access the features of the HDArray system. These APIs
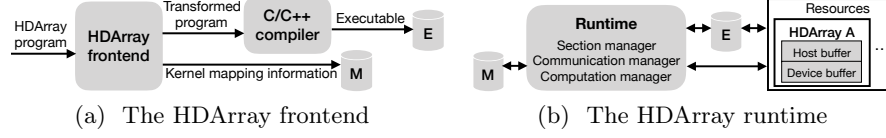
(a) The HDArray frontend   (b) The HDArray runtime

Fig. 1. An overview of the HDArray system within a single process.

and annotations are translated by the HDArray frontend into calls, arguments, and initialization files (M) for use by the HDArray runtime.

2.1 HDArray Structures

Each MPI process that maps to a single OpenCL device maintains HDArrays and their structures. Each HDArray contains the necessary state for the system to automatically generate communication and manage data distributions.

Host and device buffers contain the distributed data for an HDArray for a host and device in the HDArray program, respectively. Host buffers reside in the process memory of the host, and device buffers reside on a device.

An HDArray contains sets of array sections: *global*, i.e., across all kernels, definition sections (*GDEF*); *local*, i.e., for a particular kernel, definition sections (*LDEF*); and local use sections (*LUSE*). These sets are summarized by one or more sections of [LB:UB] that give the lower and upper bounds of the array sections for all processes. GDEF is a set of written sections not propagated to different processes, and two types of GDEFs are maintained: sGDEF and rGDEF. sGDEF for a process p describes HDArray elements that p has written, but not sent, to other processes q, i.e., the elements for which p describes the coherent copy that must be sent to other processes that use those values. rGDEF for process p describes elements of the HDArray that p has not received from q. LUSE/LDEF is the set of sections each process reads/writes in the kernel.

HDArray programs are SPMD programs, and each process maintains coherent local copies of the aforementioned four sets for all processes, and thus each process knows the array access information of the other processes. All LUSE, LDEF, and GDEF sets are empty when an HDArray is created. LUSE (LDEF) is updated by HDArray annotations and APIs, as discussed in Section 3, and GDEF is updated as a function of itself, LUSE, and LDEF, which we describe in Section 2.2.

As well, each process maintains a history of local and global sections, and sections to communicate for each HDArray of a kernel. The runtime maintains the history to reduce the overhead of determining communication by avoiding an expensive data flow evaluation if possible, as described in Section 4.2.

2.2 Communication Generation using GDEF, LDEF, and LUSE

The runtime communicates elements of HDArrays immediately before a kernel launch. Intersecting the GDEF and LUSE sets allows a process to determine which processes have elements of an HDArray that it will use in a kernel call, and therefore which elements it must communicate with that process. Fig. 2 shows the example of the intersection with two processes. Let k be the index of a
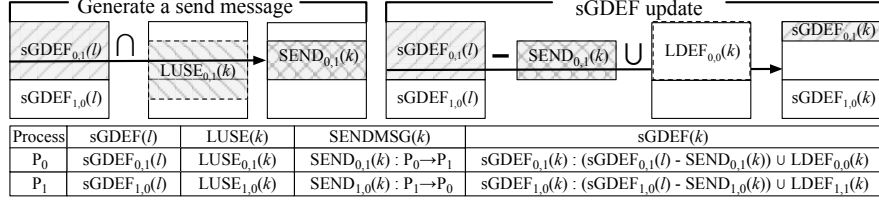
Fig. 2. sGDEF of *use* HDArray for Process 0 before and after kernel call ($\text{sGDEF}_{0,1}(l)$ and $\text{sGDEF}_{0,1}(k)$ respectively). The system intersects sGDEF with LUSE, sends the intersection from P_0 to P_1, and updates sGDEF. Note that $\text{LDEF}_{0,0}(k)$ is NULL.

kernel call and l the preceding kernel call of k. For process p, $\text{sGDEF}_{p,q}(l)$ and $\text{rGDEF}_{p,q}(l)$ are sets of sections that are live before the kernel call k. $\text{LUSE}_{p,p}(k)$ is local uses by process p for the kernel call k, whereas $\text{LUSE}_{p,q}(k)$ represents local uses of other process q for $0 \le q \le nprocs - 1, q \ne p$. The communication messages of p to send and receive for the kernel call k are then generated as:

$$SENDMSG_{p,q}(k) = sGDEF_{p,q}(l) \cap LUSE_{p,q}(k) \tag{1}$$

$$RECVMSG_{p,q}(k) = rGDEF_{p,q}(l) \cap LUSE_{p,p}(k) \tag{2}$$

After communication and kernel execution, GDEF sets for each HDArray for kernel call k must be updated to avoid redundant communication for used HDArrays, and detect new communication in the future kernel call $k + 1$ for defined HDArrays. These updates can be calculated together as Eqns. 3 and 4.

$$sGDEF_{p,q}(k) = (sGDEF_{p,q}(l) - SENDMSG_{p,q}(k)) \cup LDEF_{p,p}(k) \tag{3}$$

$$rGDEF_{p,q}(k) = (rGDEF_{p,q}(l) - RECVMSG_{p,q}(k)) \cup LDEF_{p,q}(k) \tag{4}$$

Fig. 2 shows the example of the sGDEF update. The SENDMSG and RECVMSG for the kernel call k are subtracted from sGDEF and rGDEF of kernel call l, respectively. These removes communicated data from sGDEF and rGDEF of the kernel call l. The results of the subtractions are then unioned with LDEF sets to update sGDEF and rGDEF for kernel call k. Similar to LUSE, $\text{LDEF}_{p,p}(k)$ and $\text{LDEF}_{p,q}(k)$ sets represent local definitions by process p and q for kernel call k, respectively.

3 HDArray Programming Interface

The HDArray programming interface has two types of specifications. First, a single pragma of the form `#pragma hdarray [clauses]` allows user-defined hints for generating LUSE and LDEF to find data to be accessed, and partitioning work item regions to distribute work. The functionality is contained in the clauses, described next. Second, HDArray provides library functions, hiding low-level details of distributed device programming, described below. Hereafter, we use the terms work item and thread interchangeably.

Table 1 lists the available annotation clauses. Five clauses exist: *use*, *def*, *use@*, *def@*, and *partition*. We now explain each of these.

Table 1. Core hdarray directive clauses

Clause	Description
use (array name, offset)	Declare offset(s) of an array to be used.
def (array name, offset)	Declare offset(s) of an array to be defined.
use@ (array name)	Declare absolute sections(s) of an array to be used.
def@ (array name)	Declare absolute sections(s) of an array to be defined.
partition (partition ID, dimension, dev:ID, region)	Manually partition work item regions to devices.

```
1 ...
2 #pragma hdarray partition(part0,          (10240,10240),\
3                           dev:0,   (0,3008),(0,10240),\
4                           dev:1,(3008,7232),(0,10240))
5 ...
6 HDArrayApplyKernel("corr_ker1", part0, ... );
7 HDArrayApplyKernel("corr_ker2", part0, ... );
8 ...
```

Listing 1.1. Manually partitioned Correlation host code.

*Offset Clauses: **use** and **def**.* Each HDArray accessed in a kernel can be a *use*, *def*, or both. These clauses specify elements of arrays, as offsets relative to work items, that will be read and/or written by a single work item. The offsets can be used when a kernel's array access pattern is relative to a work item, which is the most common kernel programming pattern. If the offsets are specified, the system derives LDEF and LUSE from the offset and partitioned work item region for each process. An offset can be any integer, e.g., "0" indicates the current position of an array element relative to the work item index. It also describes a direction with " + " or " − "; e.g., $(0, −1)$, referring to the previous elements of the same row. An " ∗ " denotes all elements of the array in the dimension of interest, e.g., $(0, ∗)$ denotes all elements in a row of a 2-dimensional array.

*Absolute Section Interface Clauses: **use@** and **def@**.* When it is difficult to represent offsets, e.g., a kernel's access pattern is not relative to a work item or non-rectangular, one can use the *absolute section interface* with *use@* and/or *def@* clauses and APIs. Unlike offsets, absolute sections are the coordinates of the [lowerbound, upperbound] of each dimension of an up to three dimensional item. The absolute section interface clauses inform the system to bypass LUSE/LDEF updates (Fig. 3). Instead, users call absolute section interface APIs, e.g., `HDArraySetAbsoluteUse`, described in Section 3.1, to set LDEF and LUSE. The APIs allow users to specify multiple absolute sections for each device, allowing non-rectangular regions or different access patterns for each device to be described, and enable fine tuning of communication.

*Partition Clause: **partition**.* HDArray provides an `HDArrayPartition` library call that allows a ROW, COL or BLOCK partition to be specified. HDArray will automatically partition the work evenly across processes and devices in the specified manner. A ROW partitioning is shown in Listing 1.2. The API then returns a unique partition ID, which is used, repeatedly if necessary, to execute kernels with the partitioned work item. This partition ID, along with *use* and *def* information, allows the system to know data needed by the work done by the kernel on each device.

Table 2. Core HDArray interface library functions

Prototype	Description
`int HDArrayInit(int argc, char *argv[], char *kpath, char *dpath)`	Initialize HDArray runtime environment and returns device ID. Take path to OpenCL kernel file (`kpath`) and optional device information file (`dpath`).
`void HDArrayExit(void)`	Terminate the HDArray runtime environment.
`void HDArrayShowDeviceInfo(int devID)`	Display device information. Take device ID.
`HDArray_t *HDArrayCreate(char *sym, char *type, void *uA, int dim, ... )`	Allocate host and device buffer and returns HDArray handle. Take name, type, address, and size of user array (`uA`).
`int HDArrayPartition(PART_T type, int dim, ... )`	Partition work item regions. Take type of partition and variable list for array size and region for each dimension. Supported types: `ROW`, `COL`, and `BLOCK`.
`void HDArrayApplyKernel(char *kName, int partID,...)`	Perform communication and kernel execution. Take the kernel name, partition ID, and kernel arguments.
`void HDArrayRead(HDArray_t *hA, void *uA,int partID)` ⇒ Same for HDArrayWrite	Read(Write) array section specified by partition ID from(to) HDArray(`hA`) to(from) user array (`uA`).
`void HDArrayReduce(HDArray_t *hA, void *res, REDUCE_OP op, int partID)`	Reduce specific array sections of HDArray(`hA`) to a scalar value(`res`). Supported ops: `SUM`, `PROD`, `MAX`, and `MIN`.
`void HDArraySetAbsoluteUse(char *kName, int partID, HDArray_t *hA, int devID, int dim, ... )` ⇒ Same for HDArraySetAbsoluteDef	Set absolute section used(defined) for each device. The absolute section becomes LUSE(LDEF).
`void HDArraySetTrapezoidUse(char *kName, int partID, HDArray_t *hA, int devID, int dim, ... )` ⇒ Same for HDArraySetTrapezoidDef	Set predefined shape for LUSE(LDEF). Specify four positions of upper-left, upper-right, below-left, and below-right.

The HDArray pragma enables manual partitions to be used as well, as shown in Listing 1.1, which specifies two work regions. The annotation is expanded to an internal function call that performs the partitioning and returns a unique partition ID. In lines 6-7, the partition ID (`part0`) is used to determine the work distributions of the two kernels. This annotation allows more programmer control for optimal communication tuning and load balancing.

3.1   HDArray Library Functions

The library functions, with core APIs described in Table 2, encapsulate low-level details of the programming model.

`HDArrayInit` initializes the MPI and OpenCL systems and reads in data generated by the frontend. A table is maintained for each HDArray's use in a kernel call, which is initialized with information gathered by the frontend from *use* and *def.* This use/def and partitioning information are used to generate inter-process and host-device communication. The last parameter of the `HDArrayInit` function provides a device information file containing tuples *(MPI rank, device ID)* to the kernel. Note that MPI rankfile allows the MPI rank to be known before a program run. The device ID can be used to specify which device(s) to use when multiple devices are available to a process.

`HDArrayCreate` creates a host and device buffer for the HDArray on each process using `malloc()` and `clCreateBuffer()` respectively, and allocates and initializes the HDArray's GDEF.

`HDArrayPartition` evenly partitions work item regions for each device for a given partitioning type. It then defines the partitioned work item region entry in a partition table maintained for the HDArray and return a unique partition ID that is used for work and data distribution.

`HDArrayApplyKernel` manages communication and launches a kernel. It performs the following actions:

- Bind arguments to the kernel call. The system finds the device buffer from HDArray handle, binds all other arguments directly, and calls the OpenCL function `clSetKernelArg()`.
- Determine and perform communication. *Use* HDArrays trigger communication. LUSE is updated by composing *use* offset (in the kernel table) with partitioned work item regions, and the LUSE is intersected with GDEF to determine communication (Eqns. 1 and 2). If the intersection resides in the device, it is transferred to the host using `clEnqueueReadBufferRect()`. Then, any needed inter-process non-blocking (e.g., point-to-point or collective) communication is done, followed by transferring data to the device using `clEnqueueWriteBufferRect()`. Finally, the system updates the GDEF sets, as described in Eqns. 3 and 4.
- Execute the kernel. The preferred work-group size is found using `clGetKernelWorkGroupInfo()`, the work-group size is set, and the kernel is called using `clEnqueueNDRangeKernel()`. Modified device buffers are known because of the *def* offset from the kernel table, and the system updates LDEF and GDEF (Eqns. 3 and 4) so that the host has the coherent copy.

`HDArraySetAbsoluteUse` *and* `HDArraySetAbsoluteDef` specify the absolute sections of the HDArrays annotated with *use@* or *def@* clauses (Section 3) which are then used to define LUSE and LDEF for each device. LDEF updates must be precise because they affect GDEF, which defines who owns the coherent copy of a value. Multiple `HDArraySetAbsoluteDef` calls can be used to give precise updates, but for ease of programming and avoiding errors in entering each absolute section, HDArray supports predefined shapes for absolute section updates for LDEF (and LUSE). For example, the `HDArraySetTrapezoidDef` function supports LDEF updates for two-dimensional trapezoidal or triangular shapes, which avoids multiple `HDArraySetAbsoluteDef` calls to update the LDEF.

Utility library functions In addition to the core API functions, utility library functions are provided. I/O utility functions allow the programmer to move data between user space arrays and HDArrays. The HDArray runtime updates the GDEF, LDEF, and LUSE information to reflect the data movement, thus keeping these consistent with the actual memory state. Reduction functions are also provided. If all data for an HDArray is in the host memories, a local reduction followed by an MPI reduction is performed. If some or all data is in the device memory, a device reduction is performed followed by an MPI reduction.

3.2   A Case Study: Matrix Multiply

Listing 1.2 and 1.3 show a General Matrix Multiply (GEMM) implemented using HDArray. The program uses C host code and OpenCL device code to perform the matrix multiply $C = A \times B$ on three 1024×1024 2D matrices.

```
1  void main(int argc, char *argv[]) {
2    int ni = 1024, nj = 1024, nk = 1024;
3    float a[ni][nk], b[nk][nj], c[ni][nj], alpha, beta;
4    ... // initialize variables
5
6    HDArrayInit(argc, argv, "gemm.cl", NULL);
7    int part0 = HDArrayPartition(ROW, 2, ni, nj, 0, 0, ni, nj)
8
9    HDArray_t *hA = HDArrayCreate("a", "float", a, 2, ni, nk);
10   HDArray_t *hB = HDArrayCreate("b", "float", b, 2, nk, nj);
11   HDArray_t *hC = HDArrayCreate("c", "float", c, 2, ni, nj);
12   HDArrayWrite(hA, a, part0);
13   HDArrayWrite(hB, b, part0);
14   HDArrayWrite(hC, c, part0);
15
16   HDArrayApplyKernel("gemm", part0, hA, hB, hC, alpha, beta, ni, nj, nk);
17   HDArrayRead(hC, c, part0);
18   HDArrayExit();
19 }
```

Listing 1.2. GEMM host code.

```
1  #pragma hdarray use(A,(0,*)) use(B,(*,0)) def(C,(0,0))
2  __kernel void gemm(__global float *A, __global float *B, __global float *C,
3                     float alph, float beta, int ni, int nj, int nk) {
4    int i = get_global_id(1), j = get_global_id(0);
5    if ((i < ni) && (j < nj)) {
6      C[i * nj + j] *= beta;
7      for(int k=0; k < nk; k++)
8        C[i*nj+j] += alph * A[i*nk+k] * B[k*nj+j];
9    }
10 }
```

Listing 1.3. GEMM device code.

Line 6 of the host code initializes the MPI and OpenCL environments and finds available devices to run the OpenCL kernel implemented in "gemm.cl". Line 7 evenly partitions the highest dimension of 2D array domain with regards to the number of devices. The function returns a partition ID, part0, which represents the partitioned region and is used throughout the program.

On lines 9-11, the host creates HDArrays and allocates host and device buffers with the same size of user-space arrays. After the allocation, the host binds program array variables (a, b, c) to handles (hA, hB, hC) that point into structures in the HDArray runtime and allow users to access device buffers holding data for their respective program arrays. Lines 12-14 write user arrays into the device buffer of HDArrays according to the part0 specification. Therefore, the data is distributed to different devices.

On line 16, the host launches the "gemm" kernel using the part0 and kernel arguments. part0 is used for work distribution. As shown in Fig. 3, the runtime then binds HDArray handles and host variables to the kernel arguments, handles necessary communication, and invokes the kernel (Listing 1.3). Line 17, reads the result of the computation from the device memory into user array c. Finally, the host frees all the resources, including HDArray's, and finalizes the parallel program in line 18.

The device code in Listing 1.3 shows an ordinary OpenCL kernel to be called, with an annotation added on line 1. The annotation is a #pragma hdarray statement with offset clauses discussed in Section 3. These offsets, relative to a work item index, specify slices of the A, B, and C arrays that are used and defined. The code informs the runtime system that a single thread reads all elements of

the row of the array A and all elements of the column of the array B. The zero offset indicates that each thread writes the result of the multiplication to its work item index of the array C.

With the work partitioning (part0) and per-thread array element access (offset) information provided by the host and device code, respectively, the runtime is able to generate LUSE and LDEF, and distribute work for the kernel.

4   Implementation

This section describes some implementation details not covered in Section 3.

4.1   Frontend and Execution Phase

The frontend phase, shown in Fig. 1a, uses a simple parser that performs three tasks: (1) parse OpenCL kernel functions and HDArray pragmas, (2) collect information, including *use* and *def* offset information, that is written to the file M (Fig. 1) used to initialize the HDArray table, and (3) generate code for HDArray pragmas and directives that pass partitioning information to the runtime.

Fig. 1b shows the execution phase. The main tasks of the HDArray execution phase are (1) maintaining information about HDArray sections residing on hosts and in devices, (2) determining and scheduling communication to ensure up-to-date data is available for computations, and (3) launching kernel executions. Fig. 3 sketches the logic of HDArrayApplyKernel which is the core part of the execution phase.
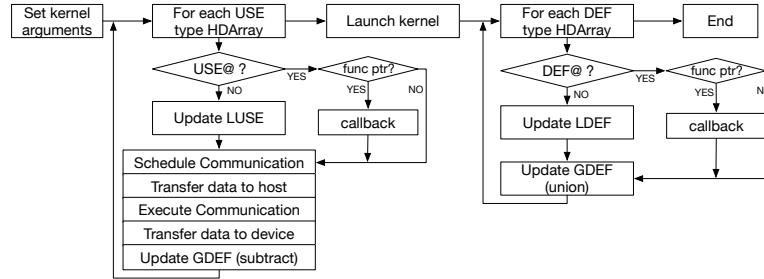

Fig. 3. Logic of HDArrayApplyKernel function.

4.2   Reducing Runtime Overhead

The major overhead incurred by our baseline runtime system comes from intersecting and updating array sections to determine communication. Data to send or receive are found by intersecting GDEF with LUSE sets, as shown in Eqns. 1 and 2, which requires the number of computations on each process p that is linear in the number of processes. To reduce the overhead, the HDArray system attempts to evaluate the intersections only when it is necessary to do so. To assist in this, the system caches the last GDEF, LDEF, and LUSE sets as well as intersections per kernel call/partition ID. If LUSE and GDEF are

unchanged for repeated kernel calls with the same partition ID, the system reuses the intersections from the last kernel call.

LUSE and LDEF sets, accessed in a repetitive kernel call with the same partition ID, can be reused. However, GDEF sets for that kernel call can change as they are a function of the LUSE and LDEF sets *and* all previous GDEFs. Therefore, the system must check whether the GDEF sets have changed. An exact comparison of GDEFs takes $O(n^2)$ time, but the GDEF comparison overhead is reduced in two steps. First, the system maintains history buffers of the IDs of LDEF and LUSE sets for each HDArray that tracks LDEF/LUSE IDs for the entire program. It then evaluates the def-use chains in the buffer to determine changes in GDEF. For example, the system compares the last LDEF and LUSE IDs with the current LDEF and LUSE IDs. If the two pairs are the same, GDEF sets for last and current kernels are also the same because a GDEF update is a function of LDEF and LUSE, thereby the system bypasses the GDEF comparison. If the history buffer does not provide enough information, as the second step, the system performs an $O(n)$ comparison of GDEF sections, enabled by keeping the GDEF sections in sorted order when updated in Eqns. 3 and 4. The sorted GDEFs allow simple and linear-time GDEF comparisons.

Finally, the system hides the overhead of section updates by overlapping the updates with host communication and device computation. It updates GDEF sets for HDArray used during the communications which are all non-blocking. For a defined HDArray, the system updates LDEF and GDEF sets during the non-blocking kernel execution.

5   Experimental Results

In this section, we evaluate the effectiveness of the proposed techniques with six publicly available benchmarks. Our evaluation is done using up to 32 OpenCL devices (limited by Xsede job submission policies) on the Xsede Comet cluster [23, 29]. Comet has 1,944 compute nodes and 72 GPU nodes, connected by a 56 Gbps FDR Infiniband. Each compute node consists of two 12 core Intel Xeon CPU E5-2680 processors running at 2.50 GHz, 128 GB of main memory. The GPU nodes consist of 36 NVIDIA P100 nodes and 36 NVIDIA K80 nodes, and each node has 4 GPUs. We use both P100 and K80 GPU nodes, each of which has total 40GB of device memory and utilize 4 GPU devices per node.

We use six micro-kernel benchmarks: GEMM, 2MM, 2D Convolution, Jacobi, Covariance, and Correlation from the Polyhedral Benchmark Suite for GPUs and accelerators (PolyBench/ACC) [11]. We compile the benchmarks with gcc 4.9.2 with -O3 and use OpenMPI version 1.8.4. OpenCL version 1.2 is used to support NVIDIA devices. Baseline numbers are found using the implementations provided by the benchmarks. For HDArray system numbers, the OpenCL device code is augmented with HDArray pragmas with *use* and *def* clauses, and C host code that includes HDArray library calls and *partition* clauses are added.
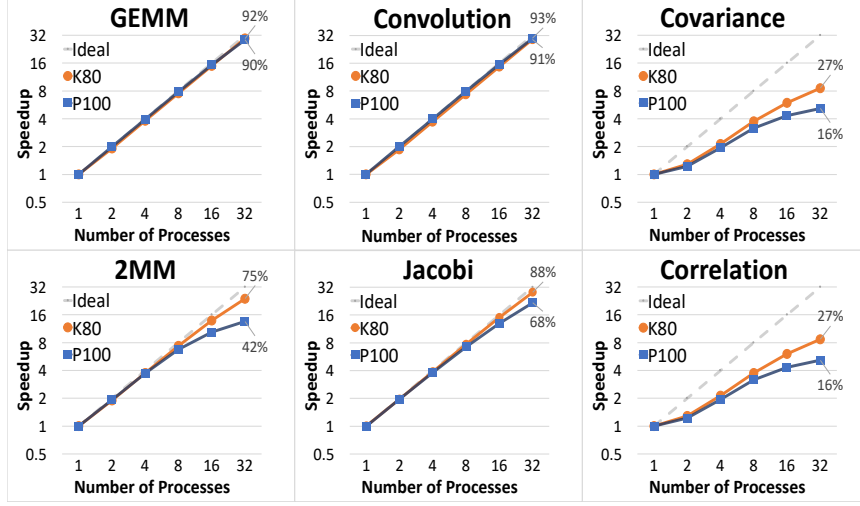
Fig. 4. Scalability for the HDArray runtime system on P100 and K80 nodes. We show the speedup for each benchmark, which is the ratio of the execution time of a single device to the execution time of the number of devices indicated on the x-axis. All the benchmarks use an automatic row-wise partitioning for data and work distribution.

5.1   Scalability

Fig. 4 shows strong scaling on both P100 and K80 nodes. The baseline time is for running one OpenCL device without HDArray. All the benchmarks perform a row-wise partition using the `HDArrayPartition` function with a ROW argument for work and data distribution. Most benchmarks running on K80 nodes scale better than on P100 nodes because the P100 is faster than the K80, and thus the communication overhead on P100 nodes is a larger fraction of the computation time.

GEMM, shown in Section 3.2, uses 10,240×10,240 matrices with 100 iterations. The HDArray runtime system detects and generates all-gather collective communication because each OpenCL work-item needs row and column elements of arrays for computation. Scaling is good to 32 processes, with similar efficiencies on the K80 (92%) and P100 (90%), due to the low ratio of communication to kernel execution time. 2MM performs two matrix multiplications, $D = A \times B$ followed by $E = C \times D$. It differs from GEMM in that 2MM runs two kernel functions within a loop and exhibits a data dependency because one kernel defines the array D used by the other kernel. With the row-wise partitioning, the efficiency drops off to about 75% (42%) on the K80 (P100) at 32 processes because of the communication cost. The cost is proportional to the number of processes, and every iteration requires the communication: once for the array B, and 100 times for the array D.

A different partitioning can be used to reduce the communication cost. 2MM with column-wise partitioning, as shown in Fig. 5, only communicates twice for arrays A and C, and the efficiency is about 98% (96%) on the K80 (P100) at 32 processes. Table 3 shows communication volumes of all 32 processes and
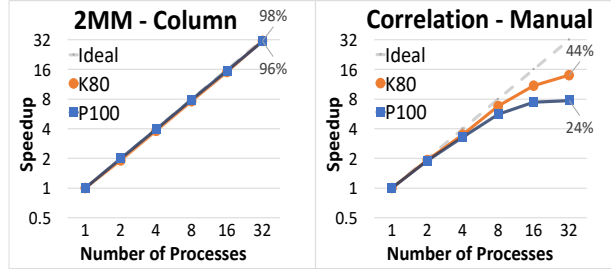
Fig. 5. Scalability for the HDArray runtime system with different partitioning methods. 2MM uses automatic column-wise partitioning and Correlation uses manual row- and column-wise partitionings.

Table 3. Total communication volume for 32 processes

Partition	Convolution	JACOBI	GEMM	2MM	Covariance	Correlation
Default (Row)	5 MB	473 GB	12 GB	1262 GB	1268 GB	1268 GB
Customized	5 MB	473 GB	12 GB	25 GB	811 GB	811 GB

noticeable volume difference for 2MM. This performance tuning was done by simply changing the `PART_T` argument of `HDArrayPartition` function.

Both Jacobi and Convolution kernels are iterative stencil codes, with four and eight neighbors, respectively. The offsets of *use* and *def* clauses have similar patterns for both benchmarks. For Jacobi, the device code consists of two kernels. One kernel processes the following computation:

$$A[i][j] = (B[i][j-1] + B[i][j+1] + B[i-1][j] + B[i+1][j])/4$$

The *use* clauses are specified for the kernel with four offsets, (0,-1), (0,+1), (-1,0), (+1,0), for an array B. The other kernel performs $B[i][j] = A[i][j]$, and zero offsets are used. The host code allocates user space arrays to have ghost cells at the array boundary, and then generates two partition IDs: one that partitions the entire region for data distribution and the other that excludes the cells for work distribution. Two kernels have a data dependency on array B in the iteration space. Convolution has four additional offsets added, but there is no data dependency. Both kernels use 20,480×24,080 matrices with 100,000 iterations, and the runtime detects and schedules a point-to-point communication. Both benchmarks scale well with an efficiency of 88% (68%) on the K80 (P100) for Jacobi, and 91% (93%) on the K80 (P100) for Convolution at 32 processes. Similar to GEMM, two nodes for Convolution show similar efficiencies due to the small communication overhead.

Covariance and Correlation are data mining benchmarks that compute a measure from statistics that show how linearly related two variables are. These benchmarks have triangular-shape array accesses, requiring the absolute section interface discussed in Section 3. Both use 10,240 vectors and 10,240×10,240 matrices with 100 iterations, and the system detects point-to-point and all-gather communication. Scaling is poor with the default row-wise partitioning with an efficiency of 27% (16%) on the K80 (P100) for Correlation (similar to Covariance). This is because evenly distributing work using `HDArrayPartition` causes
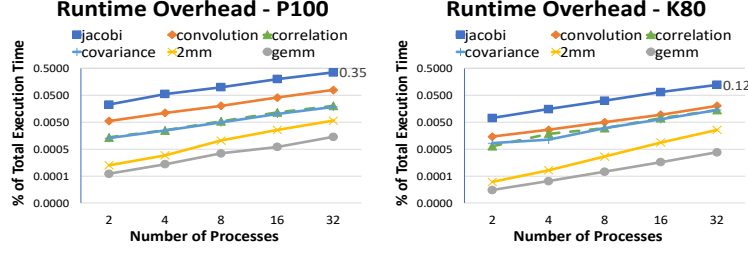
Fig. 6. Total runtime overhead for all six benchmarks on both P100 and K80. The highest overhead is 0.35% from Jacobi on P100.
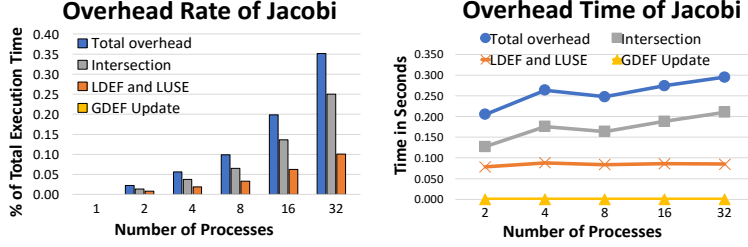

Fig. 7. Breakdown of runtime overhead for Jacobi on P100. There is no overhead of GDEF updates because these updates are overlapped with communication and computation. Running a single device also does not incur any overhead.

poor work and communication load balancing for kernels that have triangular access patterns. The most time-consuming computation is done from the upper-triangular section of an array which later requires communication to make the array symmetric. As a result, each device gets a different amount of work, and a device with the most computation also has the most communication, which leads to the imbalance of computation and communication across the devices.

Manual partitioning (Listing 1.1) with optimized absolute section updates to balance the work and communication among devices, gives better scalability with an efficiency of 44% (24%) on the K80 (P100) with the reduced communication volume as shown in Fig. 5 and Table 3, respectively. This result highlights the value of integrating manual and automatic partitioning. Also, the performance tuning does not require any changes in kernel code, but only a few lines are changed in absolute section updates and partitioning in the host code.

5.2   Runtime Overhead

Fig. 6 shows the percentage of runtime overhead based on total execution time. All of the benchmarks have less than 0.36% overhead on up to 32 processes. The benchmarks on a P100 node have more overhead than on a K80 as the P100 is faster than the K80, similar to the scalability difference. Our effort to minimize the runtime overhead, as discussed in Section 4.2, significantly reduced the cost of calculating GDEF, LDEF, LUSE, and intersection for all the benchmarks. Without the optimization, the HDArray baseline system suffers from the overhead of section calculations, which increases proportional to the number of processes.

Breaking down the highest overhead benchmark in Fig. 7, the overhead of GDEF updates are zero because they overlap with, and finish before, the communication and kernel computation. This is a large improvement because every kernel call requires the GDEF update. The intersection overhead, which includes both intersection time and caching time, is much smaller, e.g., by a factor of 19 for Jacobi on P100, than the baseline system. The result shows the benefit of using the LDEF/LUSE history buffer and linear-time GDEF comparison. Caching LDEF and LUSE for each kernel call is also beneficial. The LDEF and LUSE update overhead does not linearly increase in time because the local sections are reused after the first iteration of the kernel call. Finally, although not shown in the figure, the system reduces the number of sections by merging adjacent or redundant sections, further reducing the overheads of intersecting.

6 Related Work

Our paper is related to previous efforts to simplify distributed accelerator programming with runtime support for efficient communication. Hydra [28] is a compiler-assisted runtime system which extends OMPD [19]'s hybrid compiler-runtime communication analysis to translate and execute OpenMP programs on accelerator clusters. One difference is that HDArray handles communication without any static analysis, allowing programmers to use separate compilation and external binary libraries.

PGAS languages [6, 8, 25] have been extended to support accelerator clusters [21, 24, 26]. PGAS languages expose a global shared array to relieve programmers from data distribution and communication handling but require the specification of the affinity between data and threads, which makes data owned by a thread. As the data ownership is strongly coupled with computation, changing data distribution may require the modification of computation code. Our approach gives more freedom to the programmer to re-distribute data at any parallel program point without changes in kernel code. High Performance Fortran (HPF) [13, 27] does not support accelerators.

Researchers have proposed language extensions for the programmability of heterogeneous clusters. SnuCL [17] and SnuCL-D [15] enable OpenCL applications to run in a distributed manner without any modification. dCUDA [14] automatically overlaps on-node computation and inter-node communication with hardware support and device-side remote memory access operations. It combines the MPI and CUDA programming models into a single CUDA kernel. IMPACC [16] integrates MPI and OpenACC [2] while exploiting shared memory parallelism. It reduces the communication cost through unified MPI communication routines, a unified node virtual address space, node heap aliasing technique, etc. Despite their optimized communication with little or no code changes, programmers are forced to manage numerous low-level details of the accelerator or MPI programming because these tools provide an abstraction level analogous to OpenCL/CUDA or MPI, and require explicit data transfer or communication code.

OmpSs [4] supports task parallelism and directives for computation offloading and communication handling. Programmers specify accessed regions of shared data, but no convenient way to define and operate on subarrays is provided. We differ by supporting data parallelism and allowing users to specify per-thread offset information, and both work and data partitioning can be done automatically or manually. HOMP [32] proposes an extension of OpenMP for distributing and binding computation and data, which gives users more control of managing data and computation, but lacks cluster support and manual partitioning for specific devices.

Viñas *et al.* [31] proposed the hybrid use of Hierarchically Tiled Array (HTA) [3] for globally distributed arrays and Heterogeneous Programming Library (HPL) [30] for accelerators. Both HTA and HPL C++ libraries provide implicit parallelism and communication and hide many low-level details of MPI and OpenCL; however, there exist two different arrays: an HTA and an HPL Array, which programmers need to define and maintain. Explicit data transfer from the HPL Array to an HTA is also necessary. PARRAY [7,9] is a C language extension that introduces novel array types to separate the logical and physical structure and what kind of process/thread will operate on a dimension. Unlike HDArray, users need to specify communication mechanisms for every array and explicitly insert communication code.

Skeleton libraries [10,22] differ from us in that they can only support applications in which all the computational patterns are covered by the skeletons. Other popular platforms such as NumbaPro [20], Arrayfire [1], PyCUDA [18], Copperhead [5], OpenACC and OpenMP 4.0 all aim to make accelerator programming easier, but only target a single node.

7 Conclusions and Future Work

We have presented the HDArray interface and runtime system for accelerator clusters. The interface features a novel global programming model which separates work partitioning from the concept of data distribution, thus enabling straightforward and flexible work distribution.

The interface abstracts away many low-level details of multiple address space programming, yet supports a low-level array programming environment through the HDArray annotations and APIs for performance tuning. We showed how the HDArray interface could help programmers to write and tune array-based programs for distributed devices. The offsets provide an intuitive and simple way to describe the access patterns of kernels, and the patterns can be easily changed by simply adjusting partitions without the modification of kernel code.

The HDArray runtime system performs efficient and fully automatic communication by managing the array sections. We presented optimizations including the caching mechanism and communication and computation overlap, which reduce or hide much of the overheads of communication detection.

Future work being considered is the ability to adjust work partitions assigned to devices. This capability would allow splitting up computations to fit them in

small device memories, and adjusting the sizes of work assigned to heterogeneous devices to provide load balancing.

Acknowledgments

References

1. ArrayFire, https://arrayfire.com/
2. The OpenACC Application Programming Interface Version 2.5, 2015, http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf
3. Bikshandi, G., et al.: Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. PPoPP '06
4. Bueno, J., et al.: Productive Programming of GPU Clusters with OmpSs. In: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. pp. 557–568
5. Catanzaro, B., Garland, M., Keutzer, K.: Copperhead: Compiling an Embedded Data Parallel Language. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. PPoPP '11, New York, NY, USA. https://doi.org/10.1145/1941553.1941562
6. Charles, P., et al.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In: Acm Sigplan Notices. vol. 40, pp. 519–538. ACM (2005)
7. Chen, Y., Cui, X., Mei, H.: PARRAY: A Unifying Array Representation for Heterogeneous Parallelism. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '12
8. Consortium, U., et al.: UPC Language Specifications V1.2. Lawrence Berkeley National Laboratory (2005)
9. Cui, X., Li, X., Chen, Y.: Programming Heterogeneous Systems with Array Types. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing
10. Ernsting, S., Kuchen, H.: Data Parallel Algorithmic Skeletons with Accelerator Support. International Journal of Parallel Programming **45**(2), 283–299 (2017)
11. Grauer-Gray, S., et al.: Auto-tuning a High-level Language Targeted to GPU Codes. In: 2012 Innovative Parallel Computing (InPar). pp. 1–10 (May 2012). https://doi.org/10.1109/InPar.2012.6339595
12. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface, vol. 1. MIT press (1999)

13. Gupta, M., et al.: An HPF Compiler for the IBM SP2. In: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing. Supercomputing '95, ACM (1995). https://doi.org/10.1145/224170.224422

14. Gysi, T., Bär, J., Hoefler, T.: dCUDA: Hardware Supported Overlap of Computation and Communication. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '16

15. Kim, J., Jo, G., et al.: A Distributed OpenCL Framework using Redundant Computation and Data Replication. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '16

16. Kim, J., et al.: IMPACC: A Tightly Integrated MPI+ OpenACC Framework Exploiting Shared Memory Parallelism. In: International Symposium on High-Performance Parallel and Distributed Computing. HPDC '16

17. Kim, J., et al.: SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing. ICS '12

18. Klöckner, A., et al.: PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. Parallel Comput. **38**(3), 157–174 (Mar 2012)

19. Kwon, O., et al.: A Hybrid Approach of OpenMP for Clusters. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 75–84. PPoPP '12 (2012). https://doi.org/10.1145/2145816.2145827

20. Lam, S.K.: NumbaPro: High-Level GPU Programming in Python for Rapid Development, http://on-demand-gtc.gputechconf.com/

21. Lee, J., et al.: An Extension of XcalableMP PGAS Language for Multi-node GPU Clusters. In: Proceedings of the 2011 International Conference on Parallel Processing. pp. 429–439. Euro-Par'11, Springer-Verlag (2012). https://doi.org/10.1007/978-3-642-29737-3_48

22. Majeed, M., et al.: Cluster-SkePU: A Multi-Backend Skeleton Programming Library for GPU Clusters. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) (2013)

23. Moore, R.L., et al.: Gateways to Discovery: Cyberinfrastructure for the Long Tail of Science. In: Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment. XSEDE '14, ACM (2014). https://doi.org/10.1145/2616498.2616540

24. Nakao, M., et al.: XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters. In: Workshop on Accelerator Programming using Directives (WACCPD) (2014). https://doi.org/10.1109/WACCPD.2014.6

25. Numrich, R.W., Reid, J.: Co-Array Fortran for Parallel Programming. In: ACM Sigplan Fortran Forum. vol. 17, pp. 1–31. ACM (1998)

26. Potluri, S., et al.: Extending openSHMEM for GPU Computing. In: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium. pp. 1001–1012

27. Rice University, C.: High Performance Fortran Language Specification. SIGPLAN Fortran Forum (Dec 1993). https://doi.org/10.1145/174223.158909

28. Sakdhnagool, P., Sabne, A., Eigenmann, R.: HYDRA: Extending Shared Address Programming for Accelerator Clusters. In: International Workshop on Languages and Compilers for Parallel Computing. Springer (2015)

29. Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G.D., et al.: Xsede: accelerating scientific discovery. Computing in Science & Engineering **16**(5), 62–74 (2014)

30. Viñas, M., Bozkus, Z., Fraguela, B.B.: Exploiting Heterogeneous Parallelism with the Heterogeneous Programming Library. Journal of Parallel and Distributed Computing **73**(12), 1627–1638 (2013)
31. Viñas, M., et al.: Towards a High Level Approach for the Programming of Heterogeneous Clusters. In: Parallel Processing Workshops (ICPPW), 2016 45th International Conference on. pp. 106–114. IEEE (2016)
32. Yan, Y., et al.: HOMP: Automated Distribution of Parallel Loops and Data in Highly Parallel Accelerator-Based Systems. In: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International. pp. 788–798. IEEE (2017)