

## Purdue University Purdue e-Pubs

---

Department of Electrical and Computer  
Engineering Technical Reports

Department of Electrical and Computer  
Engineering

---

3-26-2018

# HArray: Parallel Array Interface for Distributed Heterogeneous Devices

Hyun Dok Cho

*Purdue University*, [cho111@purdue.edu](mailto:cho111@purdue.edu)

Okwan Kwon

*Nvidia Corporation*, [okwank@nvidia.com](mailto:okwank@nvidia.com)

Samuel Midkiff

*Purdue University*, [smidkiff@purdue.edu](mailto:smidkiff@purdue.edu)

Follow this and additional works at: <https://docs.lib.purdue.edu/ecetr>

---

Cho, Hyun Dok; Kwon, Okwan; and Midkiff, Samuel, "HArray: Parallel Array Interface for Distributed Heterogeneous Devices" (2018). *Department of Electrical and Computer Engineering Technical Reports*. Paper 487.  
<https://docs.lib.purdue.edu/ecetr/487>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.

# HArray: Parallel Array Interface for Distributed Heterogeneous Devices

Hyun Dok Cho  
Purdue University  
West Lafayette, IN 47907  
cho111@purdue.edu

Okwan Kwon  
NVIDIA Corporation  
Santa Clara, CA 95050  
okwank@nvidia.com

Samuel P. Midkiff  
Purdue University  
West Lafayette, IN 47907  
smidkiff@purdue.edu

**Abstract**—Heterogeneous clusters with nodes containing one or more accelerators, such as GPUs, have become common. While MPI provides a mechanism and management of inter-address space communication, and OpenCL provides a way to manage computation and communication within 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 or manual distributions of data and work. Using the distribution and information about what data is used and defined by kernels, communication among processes and among devices in a process is performed automatically. The interface provides a unified programming model to the user, thus simplifying program development.

**Keywords**—Parallel Programming Model; Distributed Shared Memory; Heterogeneous Systems; MPI; OpenCL

## I. 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. But these nodes complicate already messy distributed system programming by adding an additional layer of complexity. Traditional multi-node programming utilizes either *distributed shared memory* systems such as UPC [1] or *message passing* interfaces such as MPI [2]. Neither of these is well adapted to the problem of multiple address spaces on a node that requires proprietary communication mechanisms among the different CPU and GPU processing units. This complicates programming of these nodes because programs must incorporate, and 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 been devised to try to solve these problems. Many proposed language extensions [3]–[6] support transparent access to accelerators on different nodes. Other library-based approaches [7], [8] provide high-level language abstractions and flexible array representations. However, all of these systems require either low-level details of accelerator programming or explicit communication code. Other systems [9]–[11] implicitly expose the communication messages to programmer, but data is owned by a thread, which leads to less flexible data and work distribution. Finally, compiler-assisted runtime systems [12], [13] translate

and execute OpenMP programs on accelerator clusters, but the distribution of work and data are limited by OpenMP semantics and expressiveness.

To overcome the limitations, we introduce the Heterogeneous Distributed Array (*HArray*) interface and runtime system. *HArray* targets program execution on distributed systems whose nodes contain one or more accelerators. Parallelism is achieved using OpenCL [14] to create work on the *devices*, which can be accelerators or cores. *HArray* provides a global address space programming model with libraries and annotations to address two challenges: 1) more flexible distribution options and 2) avoiding low-level and explicit communication code.

First, annotations and APIs give programmers flexible distribution options: explicit and implicit partitionings across the devices, which overcome the intrinsic limitations of static compiler techniques [12], [13]. The programmers can then selectively use different partitioning methods for any computation, e.g., launching kernels with different work distributions. Programmers can also distribute data using an *offset* that refers to the data boundary each OpenCL work item accesses. This way, *HArray* separates work partitioning from the concept of data distribution so that programmers can change both work and data partitioning at any program point. Therefore, there is no data ownership, which addresses issues of [9]–[11] and leads to the flexible data and work distributions.

Second, *HArray* supplies an API which provides a layer of abstraction for the different underlying communication models. In addition, annotations allow the specifications of the data read and written during kernel invocations. The specifications also describe the offsets which can be easily derived from each kernel code. With the distribution, use/def, and offset information, the *HArray* runtime automatically generates efficient communication, overcoming the limitation of [3]–[8]. Therefore, the API and annotations avoid forcing programmers to handle explicit communication and allow them to focus on kernel programming.

To summarize, our contributions are

- 1) An easy programming model that can efficiently run on distributed heterogeneous devices. We introduce a novel programming model which separates work partitioning from data distribution, enabling flexible

work and data distribution.

- 2) Fully automated intra-node and inter-node communication. We introduce a runtime communication generation scheme and present the implementation of the runtime system.
- 3) A flexible user interface that can be tuned for high performance. The interface lets users utilize device resources to balance workloads and manually distribute data to reduce communication cost.
- 4) A runtime system that is maintainable and extensible. We also discuss the overheads of the runtime system and opportunities for improvement.
- 5) 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 II and III describe the design and implementation of the HDArray interface. Section IV presents a performance evaluation of the HDArray runtime system. Section V discusses related work, and conclusions and future work follow in Section VI.

## II. DESIGN OF THE HDARRAY INTERFACE

We now present the overall design of the HDArray interface, including its runtime system, and how it provides an efficient and straightforward array programming model. As shown in Figure 1, HDArray is a logical representation of two arrays: a host buffer used for communication and device buffer used for computation. We follow a Single Program Multiple Data (SPMD) execution model [15], so every process has its own copy of HDArrays. This paper targets regular array programs, meaning that arrays' access patterns are same for all threads and rectangular shaped.

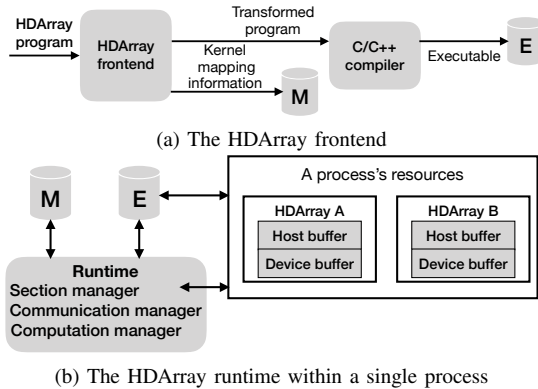


Figure 1. An overview of the HDArray system.

The HDArray system allows the user to add distribution information to the shared memory program to target accelerator clusters and write OpenCL code to exploit many-core parallelism. We choose OpenCL because of its strong portability across different computational units. Calls are made to HDArray library functions to execute the device code.

```

1 void main(int argc, char *argv[])
2 {
3     int ni, nj, nk, part0;
4     float a[ni][nk], b[nk][nj], c[ni][nj], alpha, beta;
5     HDArray_t *pA, *pB, *pC;
6
7     ni = nj = nk = 1024;
8     ... // initialize variables
9
10    HDArrayInit(argc, argv, "gemm.cl", NULL);
11    part0 = HDArrayPartition(ROW, 2, ni, nj, 0, 0, ni, nj)
12
13    pA = HDArrayCreate("a", "float", a, 2, ni, nk);
14    pB = HDArrayCreate("b", "float", b, 2, nk, nj);
15    pC = HDArrayCreate("c", "float", c, 2, ni, nj);
16
17    HDArrayWrite(pA, a, part0);
18    HDArrayWrite(pB, b, part0);
19    HDArrayWrite(pC, c, part0);
20
21    HDArrayApplyKernel("gemm", part0,
22                      pA, pB, pC, alpha, beta, ni, nj, nk);
23    HDArrayRead(pC, c, part0);
24    HDArrayExit();
25 }

```

Listing 1. GEMM host code.

The underlying HDArray runtime system then distributes data, manages communication, and runs the device code.

### A. A Case Study: Matrix Multiply

Listing 1 and 2 show an implementation of General Matrix Multiply (GEMM) using the HDArray programming model. The program uses C host code and OpenCL device code to perform the matrix multiply  $C = A \times B$  on three  $1024 \times 1024$  2D matrices.

The host code in Listing 1 initializes variables and the system, generates the *partition ID* used for data and work partitioning, registers user program arrays with the HDArrays, invokes the kernel with arguments, writes/reads arrays to/from the accelerator devices, and closes down the system.

Line 10 initializes the MPI and OpenCL environments and finds available devices to run the OpenCL kernel implemented in "gemm.cl". Line 11 evenly partitions the highest dimension of 2D array domain with regards to the number of devices. The function returns partition ID, *part0*, which represents the partitioned region and is used throughout the program.

On lines 13-15, the host creates HDArrays and allocates host and device buffers with the same size of user arrays. After the allocation, the host binds program array variables (a, b, c) to handles (pA, pB, pC) that point into structures in the HDArray runtime and allow users to access device buffers holding data for their respective program arrays.

Lines 17-19 write user arrays into the device buffer of HDArrays according to the *part0* specification. Therefore, the data is distributed to different devices. On lines 21-22, the host launches the "gemm" kernel using the *part0* and kernel arguments. *part0* is used for work distribution. The HDArray runtime then binds HDArray handles and host variables to the kernel arguments, handles necessary communication, and invokes the kernel (Listing 2), as described in Section III-B

Line 23 retrieves the result of the computation from the device memory and moves it into user array c. Similar to

the `HArrayWrite` call, the `HArrayRead` call uses `part0` to read each portion of the distributed array `pC` into user array `c`. Finally, the host frees all the resources, including `HArray`'s, and finalizes the parallel program in line 24.

```

1 #pragma harray use(A,(0,*)) use(B,(*,0)) def(C,(0,0))
2
3 __kernel void gemm(__global float *A, __global float *B,
4                   __global float *C, float alpha, float beta,
5                   int ni, int nj, int nk)
6 {
7     int i = get_global_id(1);
8     int j = get_global_id(0);
9
10    if ((i < ni) && (j < nj)) {
11        C[i * nj + j] *= beta;
12        for(int k=0; k < nk; k++)
13            C[i*nj+j] += alpha * A[i*nk+k] * B[k*nj+j];
14    }
15 }

```

Listing 2. GEMM device code.

The device code in Listing 2 shows the kernel to be called, which is an ordinary OpenCL kernel with an annotation on line 1. The annotation is a `#pragma harray` statement with *use* and *def* clauses. These specify slices of the `A`, `B`, and `C` arrays that are used and defined by the kernel call. The slice of the array is specified using an offset relative to an OpenCL work item index that operates on the array in the kernel. Hereafter, we use the terms work item and thread interchangeably. The code informs the runtime system that a single thread reads all elements of the row of array `A` and all elements of the column of array `B`. The zero offset indicates that each thread writes the result of the multiplication to its work item index of array `C`. With the per-thread array element access (offset) and work partitioning (`part0`) information, the runtime generates communication and launches the kernel.

## B. The HArray Programming Interface

We now discuss the details of the `HArray` interface. The interface has three types of specifications: (1) pragmas of the form `#pragma harray [clauses]` (described in the previous section); (2) clauses for pragmas; and (3) library functions.

1) *Pragma Clauses*: Table I lists all available clauses for `HArray` annotations. The *use* and *def* clauses are required for arrays that are accessed in kernel calls; therefore, each `HArray` accessed in a kernel must have a type of *use*, *def*, or both. These clauses specify elements of arrays that are to be read and written by a single work item. The supported offsets are shown in Table II. The zero value indicates the current position of an array element relative to the work item index. The direction of the offset is specified by  $+$  and  $-$  symbols; e.g.,  $(0, -1)$  refers to previous element of the same row. The asterisk denotes all elements of the array in the dimension of interest. For example,  $(0, *)$  denotes all elements in a row of a 2-dimensional array, which corresponds to the offset of array `A` in Listing 2. The *partition* clause, used for manual partitioning, is discussed in Section II-D.

Table I  
HARRAY DIRECTIVE CLAUSES

Clause	Description
<i>use</i> (array name, offset)	Declare offset(s) of an array to be used.
<i>def</i> (array name, offset)	Declare offset(s) of an array to be defined.
<i>partition</i> (partition ID, dev:ID, region)	Manually distribute work to devices.

Table II  
TYPES OF SUPPORTED OFFSETS FOR *use* AND *def* CLAUSES

Dimension	Offset	Description.
1D	$(-1)$	Access previous element in dimension 1 (dim1).
2D	$(0, +1)$	Access next element of the row (dim1).
3D	$(0, 0, *)$	Access all elements in dim3.

2) *HArray Library Functions*: Table III describes the library functions in detail. These further support writing array-based parallel programs. The library functions encapsulate low-level details of the programming model. For example, `HArrayInit` initiates MPI processes, and each process is assigned a single OpenCL device. `HArrayExit` cleanly shuts down the MPI and OpenCL systems. Our library functions also allow users to tune a program's performance. For example, a user can partition a work item region in several different ways to distribute both work and data when calling `HArrayPartition`. The appropriate region information with the offset may result in ideal data affinity among processes, which minimizes communication cost. `HArrayApplyKernel` manages communication and computation. Other functions are utility functions: `HArrayRead` and `HArrayWrite` perform I/O operations; and `HArrayReduce` performs a reduction and returns a scalar value. We discuss the implementation details of all the library functions in Section III-B.

## C. Communication Handling

A major advantage of using `HArrays` is that communication between host processes, and data transfer between host and device, are managed automatically. This is possible because the `HArray` runtime system knows that each device will access distributed arrays through the partition ID and the `HArray` pragma. From this information, each process knows what array sections it and other processes own; i.e., it has the coherent copy of `HArrays`, and what array sections it and the other processes are using in a kernel call. From this information, what needs to be communicated can be calculated.

`HArray` does *greedy* communication. The `HArray` runtime determines and performs the communication before launching kernels in the `HArrayApplyKernel` call. This is done by using three 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*). The three 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. To be more precise, *GDEF* is a set of written sections not

Table III  
HDARRAY INTERFACE LIBRARY FUNCTIONS

int HDArrayInit(int argc, char *argv[], char *kpath, char *dpath)	Initialize HDArray runtime environment. Returns device ID. kpath is the path to the OpenCL kernel file and dpath (optional) is the path to the device information file.
void HDArrayExit(void)	Terminate the HDArray runtime environment.
void HDArrayShowDeviceInfo(int deviceId)	Display a mapping from each MPI rank to devices and device information. Takes a deviceId which is the same as the MPI rank.
HDArray_t *HDArrayCreate(char *symbol, char *type, void *buf, int dim, ...)	Allocate array to host and device buffer and returns HDArray handles. The 1st and 2nd parameters are the name and type of an array. Takes a variable argument list for the array size.
int HDArrayPartition(HDARRAY_PARTITION part, int dim, ...)	Partition work item regions. The 1st parameter is type of partitioning. Takes a variable argument list for the array size and regions for each dimension. The list of currently supported part: ROW, COL, and BLOCK.
void HDArrayApplyKernel(char *kernelName, int partitionID, ...)	Perform communication and kernel execution. Takes the kernel name, partition ID, and kernel arguments. The order of the kernel arguments must be identical to the order of kernel parameters.
void HDArrayRead(HDArray_t *HDArr, void *userArr, int partitionID)	Read an array section from the HDArray object (HDArr) to host memory (userArr). Takes partition ID that specifies the array section to be read.
void HDArrayWrite(HDArray_t *HDArr, void *userArr, int partitionID)	Write an array section to an HDArray object (HDArr) from host memory (userArr). Takes partition ID that specifies the array section to be written.
void HDArrayReduce(HDArray_t *HDArr, void *res, HDARRAY_REDUCE_OP op, int partitionID)	Reduce specific array sections to a scalar value and copy it to res. The list of currently supported ops: HDARRAY_REDUCE_SUM, HDARRAY_REDUCE_PRODUCT, HDARRAY_REDUCE_MAX, and HDARRAY_REDUCE_MIN.

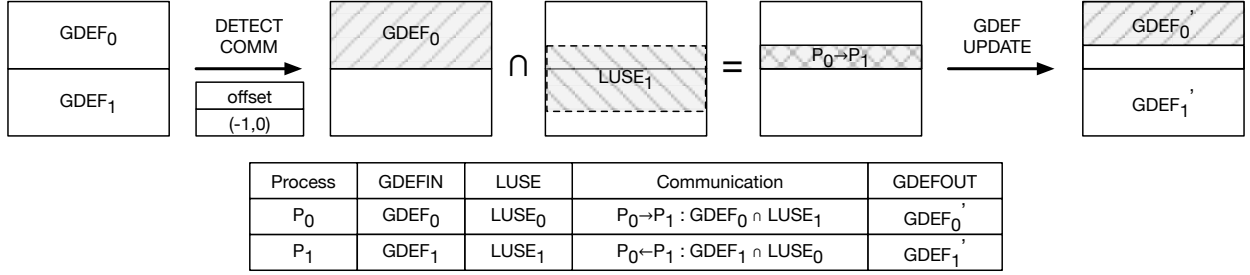


Figure 2. Process 0's array sections for *use* type HDArray before and after communication (GDEFIN and GDEFOUT respectively). The HDArray runtime system updates LUSE with offset information and detects the communication pattern. The system then finds array sections to be communicated among the processes by intersecting globally defined and locally used elements, and then performs the communication. Finally, it updates the GDEF information on each process to reflect the globally defined data that is not present in a process 0.

propagated to different processes, and LUSE/LDEF is the set of sections each process reads/writes in the kernel. Each process maintains coherent local copies of the three sets without communicating with other processes. This is possible because each process knows its rank and the total number of processes and executes the same HDArray functions.

When the program invokes an HDArrayApplyKernel, the runtime analyzes LDEF and LUSE by composing the partitioned work item region with the offset provided by each kernel. The runtime then communicates only necessary array sections by intersecting LUSE with the GDEF. The data to be sent is found by intersecting the process's GDEF with the LUSE of every other process for the kernel. The data to be received is found by each process intersecting its LUSE with the GDEF of every other process. The system also uses the intersection to perform a necessary data transfer between host and device buffer of HDArray.

After the communication and kernel execution, the system updates GDEF to reflect the changes in the HDArrays. Each

process updates its copy of GDEF for all  $HDArray_k$  that are used with Eqn. 1:

$$GDEFOUT(k) \leftarrow GDEFIN(k) \cap LUSE(k) \quad (1)$$

This update removes the data a process received from its copy of GDEF and prevents it from being received again at the next kernel call, unless it is redefined by another process. Figure 2 shows how the system maintains the GDEF with two processes when HDArrays are used in a kernel. Then, for all  $HDArray_k$  that are defined in the kernel, each process computes:

$$GDEFOUT(k) = GDEFIN(k) \cup LDEF(k) \quad (2)$$

The LDEF is merged into GDEF, indicating that the updated GDEF may have new candidates for the communication. If a process  $p_d$  defined data that a process  $p_r$  had received from it, and that  $p_r$  had subtracted from its set in Eqn. 1, this will add the redefined elements into  $p_r$ 's copy of GDEF so that

the new values will be communicated should they be used in a later kernel call.

#### D. Multi-node Accelerator Resource Management

The HDArray interface lets users pass a device information file as the last parameter of the HDArrayInit function. The file contains tuples of (MPI rank, device ID). The MPI rank can be fixed and known before the program runs, so the rank becomes the unique ID of each device. Once the programmer provides a known unique device ID to the HDArray interface, the system is able to select the user-preferred device and let users manage device resources. The users also can look up the device ID and other information through the HDArrayShowDeviceInfo library function.

```

1  ...
2  #pragma hdarray partition(part0,dev:0,(0,1024),(0,512),\
3                                dev:1,(0,1024),(512,512))
4  ...
5  HDArrayApplyKernel("2mm_ker1", part0, pA, pB, pD);
6  HDArrayApplyKernel("2mm_ker2", part0, pC, pD, pE);
7  ...

```

Listing 3. Manually partitioned 2MM ( $D=A \times B$ ,  $E=C \times D$ ) host code.

The HDArray interface also lets users partition arrays at any program point using the *partition* clause shown in Table I. This is beneficial because different kernels may need different partitions, and the default partitioning options may not be optimal. In the experimental results shown in Section IV-A, the difference of the communication cost is negligible between different types of partitioning for the code in Listing 2. However, with two matrix multiplications (2MM),  $D = A \times B$  followed by  $E = C \times D$ , the column-wise partitioning of array  $D$  (Listing 3) gives us a lower communication cost than the row-wise partitioning. This is because each process needs all elements of the column of array  $D$  when doing  $E = C \times D$ , and the program always defines the array before using it.

Other examples where manual partitioning can be useful are distributing the different amount of work to devices depending on the devices' capabilities for load balancing, or making some device computationally idle to avoid communication when communication would be more expensive than un-utilizing computational resources. Although manual partitioning requires understanding of the communication patterns of the kernels, it gives more control and flexibility to programmers.

#### E. Discussion: Comparing with Compiler-assisted Communication Generation

Unlike, for example, Hydra [12], the HDArray runtime handles communication at runtime without any static analysis. This makes the compilation simpler and can give more precise information, e.g., when data is manually partitioned or when *use* and *def* clauses are symbolically expressed, leading to lower communication costs than with static analyses. Another benefit of our approach is portability. Unlike with compiler

analyses, programmers can use separate compilation and external libraries as long as they update arrays through HDArrayRead and HDArrayWrite calls. While using the interface requires more work from the programmer than a fully automatic solution, we believe that the benefits of enabling performance tuning and higher portability, as well as the ease of maintaining the HDArray system, outweigh the cost.

### III. IMPLEMENTATION OF RUNTIME SYSTEM

This section describes the implementation details of the HDArray runtime system which consists of a *frontend* and an *execution* phase.

#### A. Frontend Phase

The frontend phase, shown in Figure 1a, uses a simple parser, written in Python, that performs three tasks: (1) parse OpenCL kernel functions and HDArray pragmas, (2) collect information that is written to a file and passed to the runtime, (3) generate code for HDArray pragmas that pass partitioning information to the runtime. The frontend phase creates the file that stores information for all kernel functions and all HDArrays of each kernel function. The information for each HDArray handle includes the offset information and the types (either USE or DEF).

If the frontend phase encounters manual partitioning directives, as discussed in Section II-D, the parser expresses the directives as calls to internal HDArray runtime routines. The call is essentially same as HDArrayPartition and returns partition ID to the variable defined in the *partition* clause. Because the information is passed at runtime, variables can be used to specify regions, etc., and their value will be used to update GDEF, LDEF, and LUSE.

#### B. Execution Phase

Figure 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.

1) *Program Startup*: With the HDArrayInit call, the runtime reads in the file the frontend generated and creates a kernel table which maps each kernel argument to the HDArrays. The table also contains the use/def and offset information for each HDArray, and this information is used when updating LDEF and LUSE, scheduling communication, and launching the kernel. The selective device feature is implemented using `clGetDeviceInfo()` to find a unique device ID that the device information file specifies. The unique ID is a combination of the PCI bus and slot ID for each device.

---

**Algorithm 1** Pseudo code of HDArryApplyKernel

---

```
1: procedure HDArryApplyKernel(kernel name, region, arguments)
2:   get kernel ID from kernel table
3:   set kernel arguments
4:   for each HDArry in kernel arguments do
5:     if type of HDArry is USE then
6:       update LUSE
7:       detect and schedule communication
8:       if scheduled then
9:         data transfer from device to host
10:        execute communication
11:        data transfer from host to device
12:        update GDEF (subtraction)
13:      end if
14:    end if
15:  end for
16:  kernel execution
17:  for each HDArry in kernel arguments do
18:    if type of HDArry is DEF then
19:      update LDEF
20:      update GDEF (union)
21:    end if
22:  end for
23:  clear LDEF and LUSE
24: end procedure
```

---

2) *Creating HDArry and Partitioning Data and Work:* An HDArryCreate call creates two internal buffers: a host buffer using malloc() and a device buffer using clCreateBuffer(), and then allocates GDEF sections for each process. HDArryPartition calls create the work item region maintained by the partition table where a unique partition ID identifies each entry.

3) *Binding Arguments to Kernel Calls:* Algorithm 1 sketches the logic of HDArryApplyKernel. On lines 2-3, the runtime sets argument values for a kernel and passes the argument values to the OpenCL function clSetKernelArg(). The runtime system finds the device buffer from the argument of the HDArry handle and calls the OpenCL function. The system binds all other arguments directly.

4) *Communication Generation:* On lines 4-15, HDArrys designated as *use* trigger communication. The LUSE of the HDArry is updated using the mapped offset in the kernel table and the work item region in the partition table. As described in Section II-C, the LUSE is intersected with the GDEF for an HDArry to determine what should be communicated and what communication pattern is needed, i.e., point-to-point or collective communication. If the intersections are most recently updated in the device memory, they are first transferred to the host memory by the non-blocking OpenCL function, clEnqueueReadBufferRect(). Once the data is available in the host memory, MPI communication is performed. The HDArry runtime supports asynchronous point-to-point MPI\_Isend(), MPI\_Irecv(), and MPI\_Waitall() communications, and MPI\_Allgatherv() and MPI\_Alltoallw() collective communications. As a result of the communication, the host buffer has up-to-date sections while the device buffer does not. Therefore, the execution phase writes the sections from the host buffer

to the device buffer using the non-blocking OpenCL function, clEnqueueWriteBufferRect(). Finally, GDEF is updated by subtracting the intersection from GDEF, as described in Eqn. 1.

5) *Executing the Kernel:* On line 16, after all HDArrys have been communicated, the OpenCL kernel function is executed using the OpenCL function, clEnqueueNDRangeKernel(). The execution phase finds the preferred work-group size to accommodate multiple architectures and receive the benefit of accessing OpenCL local memory. It queries the device for the CL\_KERNEL\_PREFERRED\_WORK\_GROUP\_SIZE\_MULTIPLE parameters by calling clGetKernelWorkGroupInfo() and sets the work-group size, e.g., the warp size of NVIDIA GPU, accordingly.

On lines 17-23, the kernel execution can modify device buffers, and from the *def* clauses, the runtime knows which HDArrys are defined. If the HDArry is defined, the LDEF is updated using the mapped offset in the kernel table and the work item region in the partition table. The LDEF is then used to update the GDEF of the defined HDArrys as seen in Eqn. 2. It also updates the state bit of the defined section to show that the device, and not the host, has the latest data.

6) *Utility Library Functions:* The HDArry runtime provides two kinds of utility functions that operate on HDArrys: (1) I/O functions that read and write data from and to HDArrys, and (2) a reduction operation on HDArrys.

The I/O utility functions allow the user to explicitly move data between the user array and the device buffer. LUSE and LDEF are updated with partition ID. HDArryRead updates the LUSE according to the requested section, with the LUSE being intersected with the GDEF set to perform the necessary host/device data transfer and MPI communication. The runtime then copies the HDArry's internal buffer to the designated user space buffer using memcpy() or clEnqueueReadBufferRect(). HDArryWrite updates the LDEF and writes the section directly to the device memory using clEnqueueWriteBufferRect(). For both functions, the GDEF is updated to reflect the communication in the same way as was done with the HDArryApplyKernel call; therefore, the GDEF is consistent with the actual state of the memory.

The reduction utility function HDArryReduce produces a scalar value by applying local and global reductions to HDArrys. First, the local reduction is performed on partitioned data within an MPI process. The runtime executes an OpenMP reduction if the latest data is only in the host memory; otherwise, the runtime executes an OpenCL implementation of Two-stage Reduction [16]. Second, MPI\_Allreduce() is used for the global reduction on the results of the local reductions each process has performed.

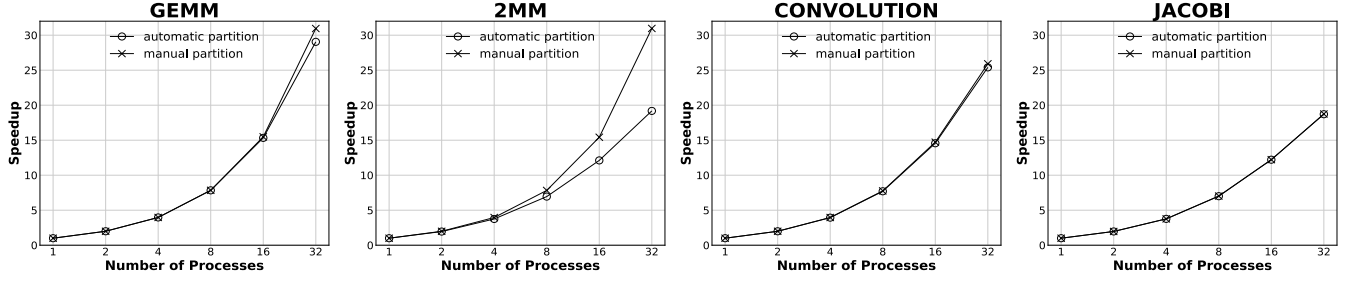


Figure 3. Scalability for the HDArray runtime system with different partitioning methods. We show the speedup for each benchmark, which is the ratio of the execution time of a single device to the execution time of the number of devices indicated on the x-axis.

### C. Discussion: Improving Runtime Overhead

A major runtime overhead comes from computing the intersections to perform communication. If the LUSE and GDEF for a kernel call are unchanged from the last call, the intersection from the last call can be used. If *use* and *def* for a kernel call is constant and the intersection is unchanged, it can be shown that the intersection will never change unless an HDArray is repartitioned. When an HDArray is repartitioned, all LUSE, GDEF, and intersections involving the array are set to NULL, forcing their re-computation. Our system uses this optimization to improve performance.

## IV. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness of the proposed techniques with four publicly available benchmarks. Our evaluation is done using up to 32 OpenCL devices on the Xsede Comet cluster [17]. Comet has 1,944 compute nodes and 72 GPU nodes, connected by a 56 Gbps FDR Infiniband and 10 Gbps Ethernet network. 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 P100 GPU nodes, each of which has total 40GB of device memory.

We compile all the programs with gcc 4.9.2 with -O3 and use OpenMPI version 1.8.4. OpenCL version 1.2 is used to support NVIDIA devices. We use four benchmarks that compute on 2-dimensional arrays: GEMM, 2MM, 2D Convolution, and Jacobi that are micro-kernels from the Polyhedral Benchmark Suite for GPUs and accelerators [18]. The OpenCL device code is modified to embed the HDArray pragma for *use* and *def* clauses, and C host code that includes HDArray library calls and HDArray partitioning pragmas are added. Our first evaluation (Section IV-A) utilizes up to 8 nodes with 4 GPU devices per node. The second evaluation (Section IV-B) uses up to 6 nodes with 2 devices per node.

### A. Scalability

Figure 3 shows strong scaling for both automatic and manual distributions. The baseline time is for one process running on one core with one device (GPU). Automatic partitioning performs a row-wise partition, and manual

partitioning performs a column-wise partition. We use 8 nodes with up to four devices per node, with additional nodes added only after four processes are executing on a node.

GEMM, shown in Section II-A, uses  $10,240 \times 10,240$  matrices with 100 iterations. With automatic partitioning, 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 for both the manual and automatic partitioning, with an efficiency of over 90%.

2MM performs two matrix multiplications, as discussed in Section II-D. It differs from GEMM in that 2MM runs two kernel functions within a loop and exhibits a data dependency because one kernel defines an array used by the other kernel. With automatic partitioning, the efficiency drops off to about 60% at 32 processes because the communication is proportional to the number of iterations. For manual partitioning, shown in Listing 3, communication occurs only twice for arrays *A* and *C*, and the efficiency is about 95% at 32 processes. Table IV shows communication volumes of all 32 processes and noticeable volume difference for 2MM. This result shows the value of integrating manual and automatic communication.

Table IV  
TOTAL COMMUNICATION VOLUME FOR 32 PROCESSES

Partition	GEMM	2MM	CONV	JACOBI
Automatic	0.41 GB	42.36 GB	5.07 GB	507.85 GB
Manual	0.41 GB	0.83 GB	5.07 GB	507.85 GB

Both Convolution and Jacobi kernels are iterative stencil codes, with eight and four 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 function 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 inserted. The host code allocates user space arrays to have boundary or ghost cells then distributes data and



work. Convolution has four additional offsets added. Both kernels use  $20,480 \times 24,080$  matrices with 100,000 iterations, and the runtime detects and schedules a point-to-point communication. The performance of manual and automatic partitioning is essentially identical with the efficiency of 80% for Convolution and 56% for Jacobi at 32 processes. Scaling is poor primarily because of HDArray runtime system overhead, not communication overhead.

Table V  
RUNTIME OVERHEAD OF JACOBI AS A PERCENTAGE OF TOTAL EXECUTION TIME: CACHING DISABLED AND ENABLED

Number of Processes	2	4	8	16	32
No Caching	0.261	0.779	2.147	5.539	14.120
Caching	0.109	0.351	0.923	2.882	8.472

The HDArray system’s largest runtime overhead is performing intersections. Table V shows that the overhead is reduced by the caching strategy of Section III-C. Other overheads are primarily from LDEF, LUSE, and GDEF computations, which linearly increase as the number of processes increases. Both Convolution and Jacobi suffer from the overhead because the runtime generates many LUSE sections. Worse, the runtime updates and clears LUSEs and LDEFs for each kernel invocation, as shown in Algorithm 1.

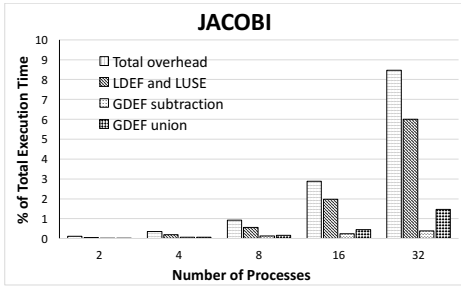


Figure 4. Breakdown of runtime overhead for Jacobi

Figure 4 shows the percentage of each runtime component’s overhead, based on total execution time. All the overheads of LDEF, LUSE, and GDEF updates are linearly increasing. Improving the scalability of the HDArray runtime is part of our ongoing work. One opportunity is to expand our caching techniques to the section updates. The system already avoids unnecessary intersections by caching each HDArray’s intersection history for each kernel, and the system at this point has enough information to skip unnecessary section updates.

#### B. Network and Process Effect on Communication Cost

With multiple nodes and four GPUs per node, executions on a small number of processors can be placed on one or two nodes or spread across multiple nodes with one process per node. The former takes advantage of faster intra-node communication and should be the fastest as long as data fits in the memory of the nodes. Our experiments show that with up to 12 processes on 6 nodes and use of Infiniband,

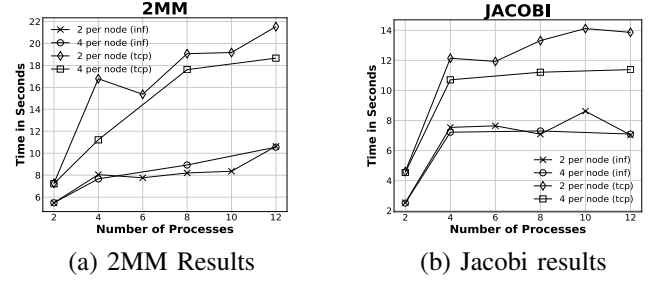


Figure 5. The effects on process scheduling with Infiniband and Ethernet.

process placement makes little difference. Some difference is noticeable when using Ethernet. Figure 5 gives the results for Jacobi and 2MM, which show the most dramatic differences between Infiniband and TCP for jobs with 2 processes per node and 4 per node.

#### V. RELATED WORK

Our paper is related to previous efforts to simplify distributed accelerator programming with runtime support for efficient communication.

##### A. Programming Models for Accelerator Clusters

UPC [1], Co-array Fortran [19], and X10 [20] are Partitioned Global Address Space (PGAS) languages that expose a global shared array so that a programmer can partition the array without worrying about the communication. These programming paradigms have been extended to support accelerator clusters [9]–[11]. PGAS languages require programmers to specify the affinity between data and threads, which is difficult to change during program execution. Our approach gives more freedom to the programmer to re-distribute data at any parallel program point. High Performance Fortran (HPF) also provides a global address space programming model that runs on clusters [21], [22]. Programmers explicitly distribute data using HPF directives, which guides computation partitioning and communication handling. HPF does not support accelerators.

Researchers have proposed language extensions for the programmability of heterogeneous clusters. SnuCL [3] and SnuCL-D [4] enable OpenCL applications to run in a distributed manner without any modification by making all the devices on a cluster logically appear on a single local node. dCUDA [5] 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 [6] integrates MPI and OpenACC [23] 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 [24] supports task parallelism and provides directives for computation offloading and communication handling. Programmers need to specify accessed regions of shared data, and OmpSs does not provide a convenient way to define and operate on subarrays. Our approach supports data parallelism and is different from OmpSs in that users specify per-thread offset information which can be easily derived for regular applications, and both work and data partitioning can be done automatically or manually.

HOMP [25] proposes an extension of OpenMP for distributing and binding computation and data, which gives users more control of managing data and computation. It also provides a number of loop distribution algorithms for better load-balancing. Unlike HArray, HOMP lacks cluster support and manual partitioning for specific devices.

Viñas *et al.* [8] proposed the hybrid use of Hierarchically Tiled Array (HTA) [26] for globally distributed arrays and Heterogeneous Programming Library (HPL) [27] 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], [28] is a C language extension that introduces novel array types using a very flexible notation with a Single-Program-Multiple-Codeblock (SPMC) programming style. The data array type separates the physical storage and logical structure of data to support logical multi-dimensional array operations. The thread array type indicates what kind of process/thread will be used for the array dimensions to unify various communication mechanisms (MPI, Pthread, CUDA, or other optimized libraries, e.g., PGAS calls). Unlike HArray, users need to specify communication mechanisms for every array and explicitly insert communication code.

Skeleton libraries [29], [30] are another approach to exploit accelerator clusters with less programming effort. One limitation is that they can only support applications in which all the computational patterns are covered by the skeletons.

### B. Easier Accelerator Programming

Due to the difficulty of programming accelerators, systems to make this easier have been proposed. One approach is enabling accelerator programming in simple scripting languages. NumbaPro [31], Arrayfire [32], PyCUDA [33], and Copperhead [34] support Python libraries for accelerators. OpenACC is an OpenMP-like directive-based accelerator programming model. OpenMP 4.0 includes the OpenACC feature. Unlike HArray, all of these APIs and programming models target a single node.

## VI. CONCLUSIONS AND FUTURE WORK

We have presented the HArray 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 flexible work and data distribution.

The interface abstracts away many low-level details of multiple address space programming, yet supports a low-level array programming environment through the HArray annotations and APIs for performance tuning. We showed how the HArray interface could help programmers to write and tune array-based programs for distributed devices.

The HArray runtime system performs efficient and fully automatic communication by managing the array sections. As a trade-off, the system incurs runtime overhead depending on communication patterns; however, our caching mechanism can overcome the overhead.

### A. Future Work

With additional enhancements, the HArray can be more efficient and reliable with a wider range of array-based programs. We discuss these now.

1) *The Expressiveness of the Offset Clause:* The offset helps users to write kernel code without considering communication; however, it cannot express non-regular, e.g., triangular, array access patterns of kernels. As a result, the users can only make conservative offsets for those kernels, causing unnecessary communication. Instead of using “relativeness” of the offset, we can have users directly specify LUSE and LDEF for each device, similar to manual partitioning. To enhance the programming model, we can provide additional directive clauses and APIs for an easier way of expressing the absolute array sections for non-regular applications.

2) *Handling Applications that Exceed Device Memory:* Unlike CPUs, accelerators typically have small physical memories and no virtual addressing. The HArray program will fail if the available device memory size is smaller than needed for computation because it allocates the entire array when HArrays are created and uses the entire partitioned work item region for a kernel. To fit in the device memory, we can allocate the device buffer during the kernel computation and use a host buffer as a backing store to split the partitioned region to work with smaller data. This optimization is feasible by enhancing `HArrayApplyKernel`.

3) *Automatic Utilizations of Computational Resources:* The HArray runtime does not support automatic load balancing. For *repetitive* programs, in which the communication pattern is the same in all iterations of the serial loop enclosing kernel calls, the runtime can balance the work by analyzing the performance of devices at each iteration and dynamically adjust work and data distributions. Another utilization such as overlapping computation and communication is not also

supported because of the SPMD execution model we use. The design of providing the overlap is ongoing future work.

## REFERENCES

- [1] U. Consortium *et al.*, “UPC Language Specifications V1.2,” *Lawrence Berkeley National Laboratory*, 2005.
- [2] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT press, 1999, vol. 1.
- [3] J. Kim *et al.*, “SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters,” in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS ’12.
- [4] J. Kim, G. Jo *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*, ser. PLDI ’16.
- [5] T. Gysi, J. Bär, and T. Hoefler, “dCUDA: Hardware Supported Overlap of Computation and Communication,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’16.
- [6] J. Kim *et al.*, “IMPACC: A Tightly Integrated MPI+ OpenACC Framework Exploiting Shared Memory Parallelism,” in *International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’16.
- [7] Y. Chen, X. Cui, and H. Mei, “PARRAY: A Unifying Array Representation for Heterogeneous Parallelism,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’12.
- [8] M. Viñas *et al.*, “Towards a High Level Approach for the Programming of Heterogeneous Clusters,” in *Parallel Processing Workshops (ICPPW), 2016 45th International Conference on*. IEEE, 2016, pp. 106–114.
- [9] J. Lee *et al.*, “An Extension of XcalableMP PGAS Language for Multi-node GPU Clusters,” in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. EuroPar’11. Springer-Verlag, 2012, pp. 429–439.
- [10] S. Potluri *et al.*, “Extending openSHMEM for GPU Computing,” in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium*, pp. 1001–1012.
- [11] M. Nakao *et al.*, “XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters,” in *Workshop on Accelerator Programming using Directives (WACCPD)*, 2014.
- [12] P. Sakdhnagool, A. Sabne, and R. Eigenmann, “HYDRA: Extending Shared Address Programming for Accelerator Clusters,” in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2015.
- [13] O. Kwon *et al.*, “A Hybrid Approach of OpenMP for Clusters,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’12, 2012, pp. 75–84.
- [14] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *Computing in Science Engineering*, vol. 12, no. 3, May 2010.
- [15] F. Darema *et al.*, “A single-program-multiple-data computational model for EPEX/FORTRAN,” *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988.
- [16] B. Catanzaro. OpenCL Optimization Case Study: Simple Reductions. [Online]. Available: <http://developer.amd.com>
- [17] R. L. Moore *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*, ser. XSEDE ’14. ACM, 2014.
- [18] S. Grauer-Gray *et al.*, “Auto-tuning a High-level Language Targeted to GPU Codes,” in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.
- [19] R. W. Numrich and J. Reid, “Co-Array Fortran for Parallel Programming,” in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM, 1998, pp. 1–31.
- [20] P. Charles *et al.*, “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing,” in *Acm Sigplan Notices*, vol. 40, no. 10. ACM, 2005, pp. 519–538.
- [21] C. Rice University, “High Performance Fortran Language Specification,” *SIGPLAN Fortran Forum*, Dec. 1993.
- [22] M. Gupta *et al.*, “An HPF Compiler for the IBM SP2,” in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’95. ACM, 1995.
- [23] The OpenACC Application Programming Interface Version 2.5, 2015. [Online]. Available: [http://www.openacc.org/sites/default/files/OpenACC\\_2pt5.pdf](http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf)
- [24] J. Bueno *et al.*, “Productive Programming of GPU Clusters with OmpSs,” in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 557–568.
- [25] Y. Yan *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*. IEEE, 2017, pp. 788–798.
- [26] G. Bikshandi *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*, ser. PPOPP ’06.
- [27] M. Viñas, Z. Bozkus, and B. B. Fraguera, “Exploiting Heterogeneous Parallelism with the Heterogeneous Programming Library,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1627–1638, 2013.
- [28] X. Cui, X. Li, and Y. Chen, “Programming Heterogeneous Systems with Array Types,” in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*.
- [29] S. Ernsting and H. Kuchen, “Data Parallel Algorithmic Skeletons with Accelerator Support,” *International Journal of Parallel Programming*, vol. 45, no. 2, pp. 283–299, 2017.
- [30] M. Majeed *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.
- [31] S. K. Lam. NumbaPro: High-Level GPU Programming in Python for Rapid Development. [Online]. Available: <http://on-demand-gtc.gputechconf.com/>
- [32] ArrayFire. [Online]. Available: <https://arrayfire.com/>
- [33] A. Klöckner *et al.*, “PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation,” *Parallel Comput.*, vol. 38, no. 3, pp. 157–174, Mar. 2012.
- [34] B. Catanzaro, M. Garland, and K. Keutzer, “Copperhead: Compiling an Embedded Data Parallel Language,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’11, New York, NY, USA.