Supporting Multiple Accelerators in High-Level Programming Models

Yonghong Yan[†], Pei-Hung Lin[‡], Chunhua Liao[‡],
Bronis R. de Supinski[‡], and Daniel J. Quinlan[‡]
[†]Department of Computer Science and Engineering, Oakland University
[‡]Center of Applied Scientific Computing, Lawrence Livermore National Laboratory
Email: [†]yan@oakland.edu,[‡]{lin32,liao6,bronis,dquinlan}@llnl.gov

ABSTRACT

Computational accelerators, such as manycore NVIDIA GPUs, Intel Xeon Phi and FPGAs, are becoming common in workstations, servers and supercomputers for scientific and engineering applications. Efficiently exploiting the massive parallelism these accelerators provide requires the designs and implementations of productive programming models.

In this paper, we explore support of multiple accelerators in high-level programming models. We design novel language extensions to OpenMP to support offloading data and computation regions to multiple accelerators (devices). These extensions allow for distributing data and computation among a list of devices via easy-to-use interfaces, including specifying the distribution of multi-dimensional arrays and declaring shared data regions among accelerators. Computation distribution is realized by partitioning a loop iteration space among accelerators. We implement mechanisms to marshal/unmarshal and to move data of non-contiguous array subregions and shared regions between accelerators without involving CPUs. We design reduction techniques that work across multiple accelerators. Combined compiler and runtime support is designed to manage multiple GPUs using asynchronous operations and threading mechanisms. We implement our solutions for NVIDIA GPUs and demonstrate through example OpenMP codes the effectiveness of our solutions for performance improvement.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Accelerators*

General Terms

Design, Implementation, Measurement

Keywords

OpenMP, Directives, Accelerators, Data Distribution

PMAM'15 February 7-8, 2015, San Francisco, USA

http://dx.doi.org/10.1145/2712386.2712405

1. INTRODUCTION

Heterogeneous computer architectures that combine general-purpose multicore CPUs with accelerators, e.g., GPUs, high-density Multiple Integrated Cores (MICs) and FPGA have been widely adopted in both supercomputers and enterprise computational servers. These architectures supply massively parallel computing capabilities provided by accelerators while preserving the flexibility of CPU accommodating computation of different workloads. Very commonly and frequently, computer nodes employ multiple accelerators, of the same or different types. Multiple accelerators, along with their additional memory, allow programmers to speedup their computation further and to solve larger problems than what a single accelerator could tackle.

High-level programming models, such as OpenACC [2] and OpenMP [13], have aimed to reduce the effort required to port an existing sequential application for parallel executions. These programming models provide language directives (pragmas) that programmers can insert into their CPU codes to indicate a region of code and its data which can be offloaded onto an accelerator for computation. However, these models focus on using single accelerator a time for offloading. Significant user effort is still needed to program multiple accelerators. For example, to use OpenMP 4.0 accelerator model for multiple devices[1], multiple offloading regions in a program, each with associated data environment need to be created. This requires programming, mostly done manually, for decomposing data and computation among accelerators, taking caring of boundary condition and data exchanges across accelerators, and synchronizations between them. While this approach is flexible, simpler approach to expressing semantics of work sharing among multiple devices would be desirable and helpful to improve the productivity of using multiple accelerators.

In this paper, we propose language extensions and compiler and runtime support for portable and productive programming for multiple accelerators. Our extensions use OpenMP as a baseline interface, and the experiments are performed using NVDIA GPUs. But we believe the functionality of these extensions can be easily adapted for other base languages, and the solutions presented in this paper are applicable to other types of accelerators. We demonstrate effectiveness of our solutions for the improvement of both performance and scalability using three programs. Our paper has the following contributions:

[1]We use the OpenMP term "device" interchangeably with "accelerator".

- Language extensions for specifying multiple devices as offloading targets for OpenMP; for specifying the distribution of multi-dimensional arrays onto a device virtual topology; for specifying shared data region (halo region) between array subregions between devices; and for specifying the distribution of loop iterations among multiple devices.
- Techniques of compiler and runtime support for implementing these extensions, including mechanisms of memory management and data marshal/unmarshal for mapping array between noncontiguous memory space and contiguous memory buffer of array subregions, and for halo regions residing on multiple accelerators; the support for reduction operations and kernel code generation; and the support for multiple device management.
- The development of the two approaches to interacting with multiple GPUs using one or multiple user threads with asynchronous or synchronous operations. Detailed performance analysis are provided for these two approaches.

The remainder of this paper is organized as follows: The motivation is given in Section 2. Section 3 presents our multi-accelerator language extensions to OpenMP. Section 4 describes our prototype implementation. We present our evaluation results in Section 5. Section 6 presents related work. Section 7 concludes the paper.

2. MULTIPLE ACCELERATOR SUPPORT USING EXISTING MODELS

High-level programming models for accelerators such as OpenMP and OpenACC provide users language directive (pragma) to annotate regions of code and data environment to be copied to an accelerator for execution. These models assume a CPU node as a host connected with one or more identical accelerators as target devices. The execution model is host-centric: a host device "offloads" data and code regions to accelerators for execution, but the accelerators do not initiate communication with hosts. For example, OpenMP provides the target construct for specifying an offloading code region. The map clause of the target directive defines the data and the mapping directions between a host and a device. By default, mapped data live from the start of the target region to the end of the region. Data mapping often involves data movement as host and device are in different memory space in most accelerator architectures.

In Figure 1, we include an AXPY example (vector addition) from line 1 to 7 that conforms to the OpenMP 4.0 standard. The target directive at line 2 has a device(0) clause to indicate that the immediately following code region, the for loop, should be offloaded to the first accelerator. The map clauses specify that the current contents of array y and x should be available on the device when offloaded execution begins, prescribed by the to and tofrom modifier. The content of y must be available on the host when the offloading completes, dictated by the from part of the tofrom modifier for array y. The parallel for directive at line 4 simply means the loop can be executed in parallel using OpenMP worksharing.

The current standards for programming on accelerators in OpenMP or OpenACC, only support for offloading code and data region to one device a time. To leverage multiple accelerators in a single program, programmers will need to manually decompose work and partition data to exploit multiple devices if using those approaches. The AXPY function from line 10 to 35 in Figure 1 demonstrate one way to achieve this. This function creates a parallel region with the number of threads equal to the number of devices (lines 12 and 13). For each device, a host thread manages its target region (lines 28 to 31). The codes for partitioning data and distributing it to each device (lines 16 to 27) must be provided by users. Obviously, the code for this simple data decomposition for multiple devices could be generated by compilers. For larger applications, such code, if written by hand, could become much more complex when programmers have to deal with issues in addition to partitioning arrays, e.g. to handle inter-device operations such as reductions and data exchanges.

The language extensions we proposed in this paper are designed to address these challenges and to improve the

```
1 void axpy_ompacc(REAL* x, REAL* y, int n, REAL a) {
2   #pragma omp target device (0) map(tofrom: y[0:n]) \
3         map(to: x[0:n],a,n)
4   #pragma omp parallel for shared(x, y, n, a)
5   for (int i = 0; i < n; ++i)
6     y[i] += a * x[i];
7 }
8
9 /* use omp parallel, i.e. each host thread
10  * responsible for one dev */
11 void axpy_mdev_v1(REAL* x, REAL* y, int n, REAL a) {
12   int ndev = omp_get_num_devices();
13   #pragma omp parallel num_threads(ndev)
14   {
15     /* chunk it for each device */
16     int devid = omp_get_thread_num();
17     omp_set_active_device(devid);
18     int remaint = n % ndev;
19     int esize = n / ndev;
20     int psize, starti;
21     if (devid < remaint) {
22       psize = esize+1;
23       starti = psize*devid;
24     } else {
25       psize = esize;
26       starti = esize*devid+remaint;
27     }
28     #pragma omp target device (devid) \
29         map(tofrom: y[starti:psize]) \
30         map(to: x[starti:psize],a,psize)
31     #pragma omp parallel for shared(x, y, psize, a)
32         for (int i = 0; i < psize; ++i)
33           y[i] += a * x[i];
34   }
35 }
36
37 void axpy_mdev_v2(REAL* x, REAL* y, int n, REAL a) {
38   #pragma omp target device (*) \
39         map(tofrom: y[0:n] dist_data(BLOCK)) \
40         map(to: x[0:n] dist_data(BLOCK),a,n)
41   #pragma omp parallel for shared(x, y, n, a) \
42         dist_iteration(BLOCK)
43   for (int i = 0; i < n; ++i)
44     y[i] += a * x[i];
45 }
```

Figure 1: AXPY examples; axpy_ompacc: AXPY for a single accelerator conforming to the OpenMP standard; axpy_mdev_v1: AXPY for multiple accelerators using OpenMP parallel and target directives; axpy_mdev_v2: AXPY for multiple accelerators using our language extensions.

productivity of using multiple accelerators. As an example of the design, the function in lines 37-45 of Figure 1 show the multi-accelerator implementation of the AXPY program. Compared to the single device version in lines 1-7, the map(tofrom: y[0:n] dist_data(BLOCK)) and map(to: x[0:n] dist_data(BLOCK), a, n) clauses in line 39 and 40 denote the even partitioning and distribution of array y[0:n] and x[0:n] across multiple target devices, i.e. the BLOCK distribution policy. Scalars a and n each will have a copy in each of the target device, taking the default distribution policy. Similarly, the dist_iteration(BLOCK) clause for the parallel for loop in line 41 indicates that the loop's iteration space is evenly distributed among the devices. Thus, each device processes the subset of the iterations that have the same range as the array subregion of x that is distributed to it because both use the same distribution policy on the same range. A more complicated example, the Jacobi iteration kernel, is shown in Figure 2. For the rest of this section, we will use this example to highlight the language design of our solutions.

3. LANGUAGE EXTENSIONS FOR MULTIPLE ACCELERATORS

High-level programming languages leverage compiler and runtime support to improve the productivity and portability of parallel programming. Directives are high-level language constructs that programmers can use to provide useful hints to compilers to perform certain transformation and optimization on annotated code regions. The use of directives can significantly improve programming productivity. The support for multiple accelerators needs extensions to an existing single-device language to provide at least the following functionalities:

1. Specification of multiple target devices for offloading, which could be an explicit list of devices, or a virtual topology of multiple devices. For example, organizing a list of accelerators in a 2-dimensional Cartesian grid virtual topology could facilitate users to write code for distributing a 2-dimensional array onto them.

2. Specification of data mapping and distribution from host to multiple devices and the data mapping and sharing between devices. This feature can also support data alignment and halo data exchanges.

3. A mechanism to distribute and to coordinate computation work across multiple devices, including: decomposition of data parallel region among devices, reductions; synchronizations; and event-driven computations across accelerators.

Prior work has studied principles of some of those semantics and has provided solutions using different programming interfaces for parallel programming. For example, MPI [6] provides library calls that create a virtual topology from processes of a communicator. High Performance Fortran (HPF) [16] includes data distribution and alignment interfaces for arrays, and approaches to create processor topology. Global Arrays (GA) [1] provides library calls that create arrays that span multiple distributed memory spaces, and calls that specify halo regions to exchange data between nodes. However, for accelerators, because of the offloading natures of accelerator architectures and the inability to initiate communication from accelerators to the host or other devices, we believe these challenges require new solutions to both programming interfaces as well as novel implementations.

Table 1: Examples of device clause extensions to specify multiple targets and to create device virtual topology

device(0:10), device(:10)	Select 10 devices starting with device id 0
device(5:3), device(5,6,7)	Select 3 devices starting with device id 5
device(*)	Select all available devices
device(0:10 & OMP_DEVICE_NVGPU)	Select the OMP_DEVICE_NVGPU type devices from the first 10 devices
device(0:4) topology(top1[2][2])	Select the first 4 devices and create a 2-d Cartesian virtual topology named top1.
device(4:8) topology([*][*])	Select 8 devices to create a system-chosen 2-d Cartesian.

When designing language extensions for these functionalities, one principle we follow is to make the interface as simple as possible for common usage, but also flexible and powerful enough to express rich set of semantics required by complicated and large applications. We use the latest OpenMP as base language for the design.

3.1 Device Types and Virtual Topology

The device clause of the target-family directives in OpenMP standards takes a single integer as a device identification number (id) for the offloading target. We extend the device clause to specify more than one devices, as Table 1 shows through examples. Multiple devices on each node could be of the same or different types. We introduce a name for specifying each type of the accelerators, e.g. OMP_DEVICE_NVGPU for NVIDIA GPUs. Other type of accelerators include Intel MIC architectures, AMD APUs, FPGAs and DSPs. A device is abstracted and identified through an integer identification number (id), and the implementation provides additional APIs to query the device type from its id, and to query its system identification and properties.

The extensions also allow to create a virtual topology of multiple devices, a concept borrowed from MPI and HPF. The examples show how to use the extensions to create a Cartesian virtual topology from a list of devices. A virtual topology could be named so we can simply reuse it later. Virtual topology will help users to specify mapping of a multiple dimensional array to a set of devices in a more intuitive and productive way.

3.2 Data Distribution

When offloading a parallel region processing one or multiple multi-dimensional arrays to multiple devices, it often requires partitioning the data and the work among the devices. Our solution for the data distribution extends the map clause to support partitioning of multi-dimensional arrays and to distribute (data movement) each partition onto devices. The language interface allows users to specify how to distribute an array onto a virtual topology of multiple devices and also provide users with options to control which dimension of the array maps to which dimension of the topology.

The extension is the dist_data clause, as also shown in

```
1#pragma omp target data device(*) \
2  map(to:n, m, omega, ax, ay, b,\
3  f[0:n][0:m] dist_data(BLOCK,*)) \
4  map(tofrom:u[0:n][0:m] dist_data(BLOCK,*))\
5  map(alloc:uold[0:n][0:m] \
6        dist_data(BLOCK,*) halo(1,))
7while ((k<=mits)&&(error>tol))
8{
9#pragma omp target device(*)
10#pragma omp parallel for collapse(2) \
11  dist_iteration(BLOCK)
12  for(i=0;i<n;i++)
13    for(j=0;j<m;j++)
14      uold[i][j] = u[i][j];
15
16#pragma omp halo_exchange (uold)
17
18#pragma omp target device(*)
19#pragma omp parallel for private(resid) \
20 collapse (2) reduction(+:error) \
21            dist_iteration(BLOCK)
22  for (i=1;i<(n-1);i++)
23    for (j=1;j<(m-1);j++)
24    {
25      resid = (ax*(uold[i-1][j] + uold[i+1][j])\
26          + ay*(uold[i][j-1] + uold[i][j+1])\
27          + b * uold[i][j] - f[i][j])/b;
28      u[i][j] = uold[i][j] - omega * resid;
29      error = error + resid*resid ;
30    } // the rest code omitted  ...
31}
```

Figure 2: Jacobi kernel using our proposed multi-accelerator directives

Table 2: Distribution policies

DUPLICATE	The full range of this dimension is duplicated into multiple copies, one for each device. This is the default policy if no policy is specified.
BLOCK(n)	Divides the indices in a dimension into contiguous, equal-sized blocks of size N/P (P is the number of devices in the target dimension of the topology) and each device takes one block (n is the number of element in the block; default: n= N/P)
CYCLIC(n)	Maps every *i*th block to number *i* device of the target dimension of the device topology. (default: n=1)

previous AXPY example. The full syntax of this clause is "dist_data(*dist_policy*[,...]) [topology (*dist_target*)]". The *dist_policy* parameters are used to specify the distribution policy for each dimension of the mapped array. We have borrowed the interfaces from HPF for specifying distribution policies, shown in Table 2. The *dist_target* parameters denote the target device of this distribution, either as a virtual topology of devices, or one or multiple device ids. By default, if no topology and targets are specified, the targets are the device targets of the associated target-family directive for the map clause. Table 3 shows examples that demonstrate our design. Figure 2 also includes such usages at lines 2 and 3.

3.3 Halo Regions and Halo Update

The approach we have discussed so far for parallelizing many scientific applications to use multiple accelerators is still the traditional domain decomposition method, i.e. to distribute the work among different processing elements. For a large subset of those applications, e.g. stencil computation on regular grids, decomposed sub-domains logically overlap at the boundaries and the overlapped region are updated regularly with neighbor values during the computation. The overlapping regions are called ghost or halo regions and the operations that perform the update and data exchange are often referred to as halo update.

Table 3: Examples of using dist_data clause extensions to specify data distributions

device (0:4) map(to:x[0:n],a,n)	Implicitly map and distribute the full array x and scalars a and n to each of the four devices
device (0:4) map(to:x[0:n] dist_data(DUPLICATE))	Explicitly map and distribute the full array x to each of the four devices.
device (0,1) map(to: x[0:n/2] dist_data(DUPLICATE) topology (0,1), x[n/2:n] dist_data(DUPLICATE) topology (1),a,n))	Distribute the first half of array x to both device 0 and 1, and the second half to device 1. Map scalars a and n to each of the two devices.
device (0:2) map(to: x[0:n] dist_data(BLOCK))	BLOCK distribution (evenly partition) of array x to the two devices.
device(0:4) map(to:A[100][0:n][64:1024] dist_data (DUPLICATE , BLOCK, DUPLICATE))	Partition array A from its second dimension using BLOCK policy among the 4 devices. Map the full range of the first dimension and the specified range (64:1024) of the first dimension in each partition to each device.
device(0:4) topology([2][2]) map(to:A[100][100][0:64] dist_data (BLOCK, BLOCK, DUPLICATE)	Distribute the first and second dimensions of array A across the first and second dimensions of the device topology, both using BLOCK policy. Map the specified range of the third dimension (0:64) in each partition to each device.
device(0:4) topology([2][2]) map(to:A[100][n/2:n/2][64:64] dist_data (DUPLICATE, DUPLICATE, BLOCK) topology ([][*])	Partition only the third dimension of array A according to the second dimension of the device topology using BLOCK policy. Map the full and specified range of the first and second dimensions of A in each partition. For the first dimension of the device topology, map the corresponding partition to all devices with the same coordinates as of its second dimension.

The extensions we developed include a halo clause to specify halo regions and a halo_update directive for halo update operations. Halo regions are part of data distribution annotation, and it is only applied to BLOCK distribution policy for obvious reasons. When using the halo clause, halo regions are specified using the number of array elements in each dimension. The clause also allows for specifying the halo regions for left (descending index) and right (ascending index) neighbors separately, and for two types of edging policies (periodic and reflecting) for dealing with halo regions at the dimension edges of the original array. For 1-element periodic edging halos of array, e.g. A[n], the right halo to A[n-1] is A[0] and left halo to A[0] is A[n-1]. For reflecting edging halos, the right halo to A[n-1] is A[n-2] and left halo to A[0] is A[1]. The full syntax for the halo clause is "halo (*#halo_elements*[:*edging_policy*]|*#right_halo_elements*[:*edging_policy*] [,...]". The examples in Table 4 show the use of the clause.

Table 4: Examples of using halo clauses and halo update directives

map(to:A[100] [100] [100] dist_data (BLOCK, BLOCK, DUPLICATE) topology([2][4]) halo(2:periodic,4,))	Distribute array A onto 2x4 device topology. BLOCK distribution policy is applied for the first and second dimension. For the first dimension, 2 elements are defined as a halo region with neighbors from both descending index (left) and ascending index (right) directions. Periodic edging policies is selected. For the second dimension, 4 elements for both left and right neighbors and non-edging policy define the halo regions.
map(to:A[100][100][100] dist_data(BLOCK, BLOCK,DUPLICATE) topology([2][4]) halo(2:reflecting| 3:periodic,4,))	Similar to the above example but specifies different halo region sizes and edging policies for left and right neighbors of the first dimension. The | symbol is used to separate left and right halos.
halo_update left(A[*][])	Update the left halo region in the first dimension from neighbors.
halo_update right(A[][*])	Update the right halo region in the second dimension.
halo_update leftright(A[*][*])	Update left halo regions first and then right ones in the first and second dimensions.
halo_update leftright(A, B, C)	Update the left and then right halo regions in all dimensions that have halo region specified for the three arrays.
halo_exchange (A, B, C)	Simplified directive for the above one, i.e. halo_exchange is the same as "halo_update leftright"

The halo_update directive takes parameters of update directions and updating array and dimensions. The full syntax is "halo_update *update_direction*(*array_dims*[,...])". The value for update direction could be one of left, right, leftright and rightleft. It is important to note that halo update operations among multiple devices are collective operations involving a barrier-like synchronization between devices, which may incur high overhead. To allow for updating left and right halo separately gives users fine-grained control of coordinating computations and data movement. The examples in Table 4 show the use of this directive.

The Jacobi kernel example in Figure 2 also uses this syntax to specify a halo region for array uold in line 5 and 6. The uold array is evenly distributed among the devices and the halo(1,) clause specifies one-column as the halo region, in both the left and right directions. The halo exchange directive in line 16 makes calls to perform halo data exchange.

3.4 Loop Iteration Distributions

When distributing a parallel loop among multiple devices, the loop iteration space must be divided among the devices. In OpenMP worksharing construct, the schedule clause is provided for specifying how the loop iteration space should be scheduled among threads of a team. This principle applies to the loop iteration distribution among multiple devices. However, users must be careful because incorrect distribution of loop iterations may lead to accesses to array subregions that are not distributed to that device.

The language extensions we introduced to achieve this is the dist_iteration clause for a parallel loop. This syntax and semantics of this clause is similar to dist_data if a level of loop iterations is treated as a dimension of an array. The full syntax is "dist_iteration(*dist_policy*[,...]) [topology (*dist_target*)]". The values allowed for *dist_policy* includes BLOCK, CYCLIC and DUPLICATE. For example, the BLOCK distribution specifies that the loop iteration space will be evenly distributed across the selected dimension of the device virtual topology. Line 19 of Figure 2 shows an example with this clause. This approach gives users a simple interface to partition and distribute a parallel loop among multiple devices.

The optional topology(*dist_target*) clause for both dist_data and dist_iteration is used to select the distribution dimensions of either a pre-defined topology or a new topology. Be default, if no topology is present, the two dist_* clause will use the topology defined in the device ... topology of the target-family directives.

The extensions we introduced do not break the syntax and semantics of the target directives in the OpenMP standard that support offloading onto single device. We achieve this by carefully designing the default behavior of these extensions. For example, the default distribution policy, The support of using multiple devices using these extensions

4. COMPILER AND RUNTIME SUPPORT

One of our design principles of these language extensions is to make the programming interfaces simple, but expressive enough to enable the creation of portable programs with minimum user efforts. The implementation of these extensions relies on advanced compiler transformation and novel runtime techniques. Essentially, a compiler will automate the generation of codes of runtime calls for working with multiple devices, including functions for preparing devices, handling data movement, distributing data and computation, book-keeping details of managing multiple accelerators, and so on.

4.1 Overview

Our current implementation targets NVIDIA GPUs and generates CUDA code, though the solutions should be generally applicable to other types of accelerators. We use the HOMP [11] (Heterogeneous OpenMP) compiler as a baseline for the implementation.

The implementation takes an incremental, two-stage translation approach: the first stage is the translation from coding using the proposed directives and clauses in this paper into standard OpenMP 4.0 code using single device directives and clauses (similar to the translation from axpy_mdev_v2() to axpy_mdev_v1() function in Figure 1), plus necessary runtime calls for distributing data and loop iterations. The second stage is the C/CUDA code generation from the standard single-device OpenMP code using the HOMP compiler. Compared to a single stage translation which directly generates C/CUDA code from the input codes using multiple device extensions, this incremental translation approach has several benefits: 1) We can reuse the existing compiler and runtime support for single device in HOMP; 2) Debugging the compiler and runtime implementation becomes simpler when being divided in two simpler stages; 3) As a source-to-source compiler, the results of the first stage translation can be compiled with any other standard OpenMP 4.0 compilers to support multiple devices.

4.2 CUDA Kernel Code Generation

Code generation from a parallel loop to a CUDA kernel is performed by the HOMP compiler. The CUDA kernel uses a round-robin scheduler to dispatch loop iterations batch by

batch to all the threads of a CUDA kernel that will exhibit coalesced memory access behavior by the threads for optimal performance. The Figure 3 shows the CUDA kernel for the inner Jacobi iterative algorithm of Figure 2. The double-nested loops in the source code (line 18 to 30 in Figure 2) are collapsed into a single loop as requested by the collapse(2) clause of the input code. A round-robin scheduling is realized by a call to XOMP_static_sched_init that initializes the scheduler, and the following while (one batch per iteration) loop after the initialization. At the end of this kernel, per-block reduction is performed as shown at line 43.

4.3 Multiple Device Management

Managing multiple accelerators from a host includes tasks such as maintaining queues that store offloading request, launching kernels onto devices, and initiating data movement operations between the host and devices, etc. Interactions with GPU accelerators, could often be done through either synchronous operation (blocking the calling CPU thread) or asynchronous operations. Given a multi-threaded runtime environment, there are several approaches for managing multiple devices.

1. Using a single user-level thread to manage multiple devices using synchronous operations. This is apparently a simple approach and commonly used for dealing with one accelerator. However, it will limit the parallel capabilities between multiple devices because the thread has to wait for the completion of tasks of one device before doing anything with another device.

2. Using multiple user-level threads, or OpenMP threads to manage multiple target devices, and each thread uses synchronous operations to interact with each device. This approach has the advantage of parallel interaction with devices to achieve parallelism in term of multiple accelerators. However, the user-level threads are still blocked to wait for the completion of accelerator tasks and cannot do any other work.

3. Using one user-level thread to manage all the target devices, but using asynchronous operations to interact with devices. The asynchronous interactions will allow parallel launching of device tasks to maximize the utilization of all devices. However, since this user-level thread still needs to wait for the completion of the device operations at some point of the execution, this thread will not be able to do any useful work while waiting for the completion of these operations.

4. Using one or multiple daemon threads (instead of user threads) to manage one, or multiple, or all of the devices. Apparently, this approach will not block on any user-level thread, theoretically. But in most scenarios, user-level threads often have to wait for the completion of device tasks before proceeding the program execution. Thus, in reality, the advantage of this approach may not be very beneficial.

We choose the second approach in our implementation since it is consistent with our two-stage translation method. Multiple OpenMP threads are used to manage multiple target devices. Each OpenMP thread performs synchronous operations to interact with other devices. The third approach is also chosen as a baseline choice for comparison.

```
1 __global__ void OUT__1__10550__(int start_n, int n,int m,float omega,float ax,\
2                                  float ay,float b,float *_dev_per_block_error,\
3                                  float *_dev_u,float *_dev_f,float *_dev_uold)
4 {
5   // variable declarations, omitted here ...
6   // CUDA 1-D thread block:
7   int _dev_thread_num = gridDim.x * blockDim.x;
8   int _dev_thread_id = blockDim.x * blockIdx.x + threadIdx.x;
9
10  int orig_start =start_n;
11  int orig_end = n*m-1;
12  int orig_step = 1;
13  int orig_chunk_size = 1;
14
15  // use a round-robin scheduler to schedule the loop iterations
16  XOMP_static_sched_init (orig_start, orig_end, orig_step, orig_chunk_size, \
17      _dev_thread_num, _dev_thread_id, \
18      & _dev_loop_chunk_size , & _dev_loop_sched_index, & _dev_loop_stride);
19  while (XOMP_static_sched_next (&_dev_loop_sched_index, orig_end,orig_step, \
20        _dev_loop_stride, _dev_loop_chunk_size, _dev_thread_num, \
21        _dev_thread_id, & _dev_lower, & _dev_upper))
22  {
23    // collapsed two level of loops
24    for (ij = _dev_lower; ij <= _dev_upper; ij ++) {
25      // restoring to original two level loop indices
26      _dev_i = ij/(m-1);
27      _dev_j = ij%(m-1);
28
29      // boundary check
30      if (_dev_i>=start_n && _dev_i< (n) && _dev_j>=1 && _dev_j< (m-1))
31      {
32        _p_resid = (((((ax * (_dev_uold[(_dev_i - 1) * MSIZE + _dev_j] + \
33                       _dev_uold[(_dev_i + 1) * MSIZE + _dev_j])) + (ay * \
34                     (_dev_uold[_dev_i * MSIZE + (_dev_j - 1)] + \
35            /*some terms omitted here */_dev_f[_dev_i * MSIZE + _dev_j]) / b);
36        _dev_u[_dev_i * MSIZE + _dev_j] = (_dev_uold[_dev_i * MSIZE + _dev_j] \
37                     - (omega * _p_resid));
38         _p_error = (_p_error + (_p_resid * _p_resid));
39      }
40    }
41  }
42  // inner thread block reduction
43  xomp_inner_block_reduction_float(_p_error,_dev_per_block_error,6);
44 }
```

Figure 3: Translated Jacobi: GPU side stencil computation

4.4 Distribution of Array and Loop Iteration

For programming languages such as C and Fortran, a multiple dimensional array is stored in contiguous memory space. When distributing an array onto multiple accelerators, only the partition will be moved to and from a device, not the whole array. The runtime is responsible for array partition and creating an array subregion of each partition, referred to in the runtime as a "data map". A data map includes information for the original array, subregion boundary, halo region if existing, and the mapped array region and memory space on the device. An array subregion is in a contiguous rectangular space, but the data of this subregion is not necessarily in a contiguous memory space. When moving data of an array subregion between a host and a device, or between two devices, runtime could make multiple data movement calls, one for each contiguous segment from the source location to its destination. In another approach, the runtime only makes one call for the actual data movement between host and device, or between devices. However, for array subregions in non-contiguous memory space, runtime will need to allocate a temporary contiguous memory space and copy data from noncontiguous memory space to the contiguous temporary memory space (referred to as data marshaling) before the actual movement between devices and host. Conversely, there is a need to restore data from a contiguous memory space to the memory of its original array subregion(referred to as data unmarshalling). Our runtime system takes the second approach and maintains an internal buffer for data marshalling and unmarshalling.

The distribution of array and loop iteration, and the mapping of continuous memory space to multiple memory space on the devices, will change the way we access the subregion

of the map on the device as compared to the original reference to array. Certain offset (positive or negative) and boundary check will be needed to ensure the correctness of the code. The compiler transformations will guarantee array references to its original array index spaces are properly translated to references to the array subregion that is mapped to each device. We also use internal bookkeeping variables and their corresponding runtime functions to keep track of the mapping.

4.5 Halo Regions and Reductions

To handle halo regions, the runtime duplicates boundary elements for each associated array subregion. For a halo region lives in noncontiguous memory space, a buffer will be created in the shared memory space of the GPU and corresponding CUDA threads will be responsible to copy the elements of halo region to the buffer. The halo region exchange is implemented through asynchronous CUDA peer access operations that do not involve the host if the hardware supports this feature. For those hardware without this feature, runtime will perform a two stage of data movement involving a host buffer as relay of data movement between two devices. The runtime will detect the peer access capability of the hardware and decide which exchange option to use.

Reduction using multiple accelerators is implemented through a three-level reduction process for GPUs: 1) the inner CUDA thread block level reductions performed on GPUs 2) the per-device reduction performed as an asynchronous call back operation at host from the results of the inner reduction. and 3) the global level reduction done on the host from the results of per-device reductions performed by a simple loop. The per-device reduction technique is similar to the approach we developed for OpenACC [21]. We are able to use shared memory in the inner block reduction for improved performance.

5. EVALUATION

Three scientific kernels, AXPY, Jacobi and matrix multiplication, are ported to evaluate our initial implementation. The machine used for this study has dual quad-core Intel Xeon 5530 processors (8 cores in total) running at 2.4 GHz with 24GB DDR3 ECC memory. With hyperthreading support, each node can launch a maximum of 16 OpenMP threads. Four NVIDIA Tesla C2050 Fermi GPGPUs are installed, with CUDA version 5.5 driver as its software environment. Each GPU has 448 GPU cores and 3 GB GDDR5 memory.

We reported two sets of results in this paper. First, the performance of using two different multiple device management approaches discussed before is reported, one user-level thread per accelerator and only one user-level threads for all accelerators. Secondly, we reported the total execution time of each offloaded code region with each configuration (a combination of the number of GPU devices and a problem size). This execution time includes costs for memory allocation and data transferring, kernel execution time, and cost for CPU book-keeping and synchronizations for devices. All three benchmarks show good scalability when exploiting multiple GPUs.

5.1 Comparison of the Two Mechanisms of Managing Multiple Accelerators

The two mechanisms include a baseline approach (opt1): one OpenMP thread interacts with all the GPUs using asynchronous operations; and another approach (opt2): multiple OpenMP threads are designated and one thread is only talking with one GPU using synchronous operations. The performance comparison of the two mechanisms are shown as speedup in Figure 4, Figure 5 and Figure 6. The one thread per GPU option (opt2) shows superior performance compared to the baseline option using only one OpenMP thread in interacting multiple GPUs.

However, the performance ratio does not show linear scalability with increment in the number of GPUs. The root cause is in the limitation in hardware. The 4 GPUs in our machine share a single PCIe switch linked with a single QPI link to the chips. The shared datapath will easily become a bottleneck when large amount of memory movements and kernel launching operations flood to the 4 GPUs through the QPI link and the PCIe switch. Our kernels all have well balanced data parallel code regions, and thus the offloading of the kernels to GPUs happens in this flooding fashion. As a result, even those operations of interacting with all the GPUs are fully parallelized by the threading mechanisms, some operations will be serialized by the hardware. Increasing the number of GPUs will add the amount of serialized memory operations, thus increasing the contentions. This will impact the total execution time, thus the scalability. We have performed detailed profiling of the multiple GPU execution using NVIDIA Visual Profiling tools and the profiling confirmed this. The lesson learned from this experiment is that to achieve scalable speedup using multiple GPUs in one system will require a scalable hardware bus system.

With a higher performance delivered, we choose one thread per GPU option (opt2) in the evaluation for the following performance reports.

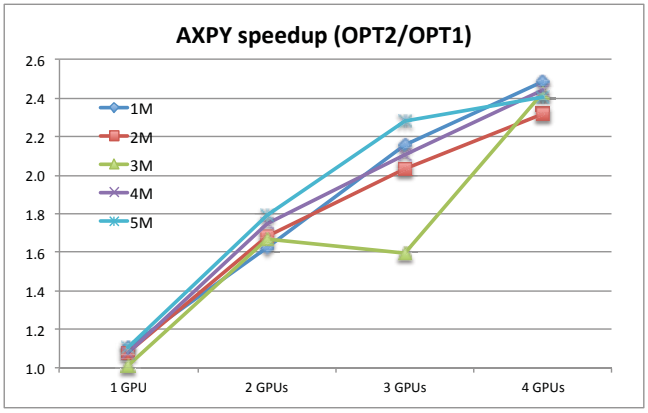

Figure 4: AXPY performance comparison of using two device management mechanisms(OPT1: one thread for all GPUs, OPT2: one thread per GPU

5.2 Performance of using Multiple GPUs

AXPY is a combination of scalar multiplication and vector addition, $Y \leftarrow \alpha * X + Y$, where X and Y are vectors. The computation of AXPY has no data dependence and can easily be parallelized. Figure 7 shows the performance results for AXPY kernel. Overall, it shows good speedup from 1 GPU to 2 GPUs, but the speedup is not impressive

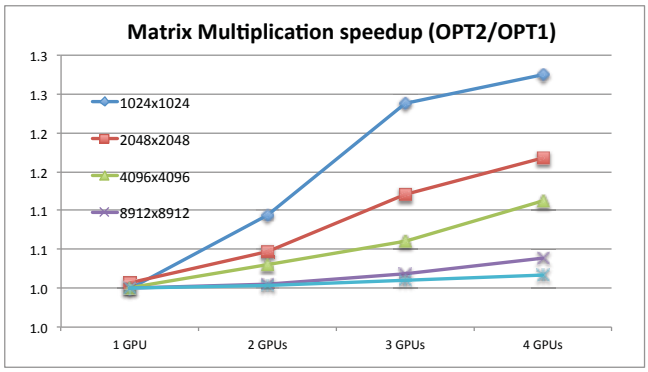

Figure 5: Matrix Multiplication performance comparison of using two device management mechanisms(OPT1: one thread for all GPUs, OPT2: one thread per GPU

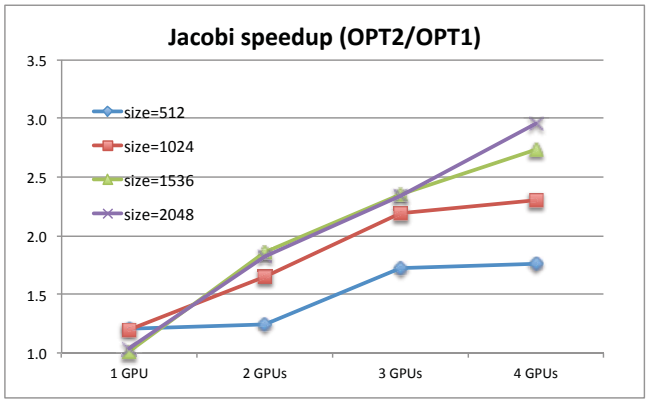

Figure 6: Jacobi performance comparison of using two device management mechanisms(OPT1: one thread for all GPUs, OPT2: one thread per GPU

from 2 to 3 or from 3 to 4 GPUs. Even not reported in the figure, we also collected the results of its OpenMP version using 8 threads, and it outperforms the versions using single and multiple GPUs simply because the amount of computation in AXPY is so trivial that the total execution time will be dominated by data movement overhead, as shown in Table 5. The large gap between data movement cost (map to and from) and computation time make the overlapping of the computation and data movement trivial for improving performance by increasing number of GPUs because of the serialized data movement imposed by the hardware limitation discussed above. Therefore, executions with more than 2 GPUs accumulate higher runtime overhead in the AXPY study, as shown by the marginal performance improvement for 3 and 4 GPUs.

Table 5: Break down execution time for AXPY with 1 GPU

map_to (X & Y)	computation	map_from (Y)	total GPU time
27.31	1.06	15.77	44.14

Matrix multiplication, $C_{ij} = \sum_{k=0}^{n} A_{ik} * B_{kj}$, has a $O(n^3)$ complexity in the computation. It is known that GPUs can deliver higher performance compared to parallelization using

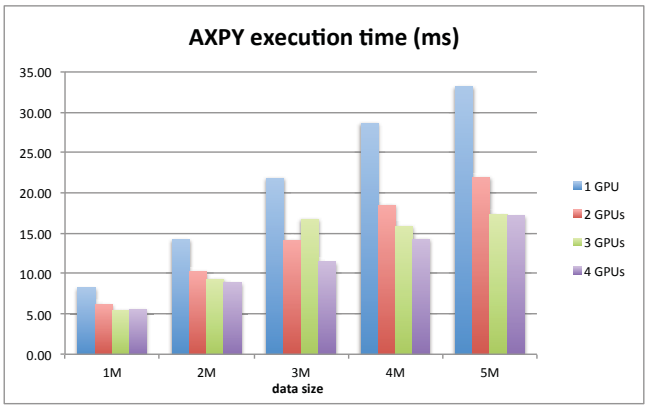

Figure 7: Performance result for AXPY

multi-threading on CPUs. To utilize multiple accelerators, three data distribution strategies of the associated matrices are implemented. 1) This first strategy uses row-based BLOCK distribution policy for the matrix A and C. Each accelerator device owns a block of rows of matrix A and whole matrix B in its memory. The distributed data will be used to compute a subset of matrix C in each individual device. 2) The second strategy is similar to the first one but uses column-based BLOCK policy. Each accelerator device is assigned the whole matrix A and a block of columns of matrix B. A subset (the same amount of columns as that of B) of matrix C can be updated in each accelerator. 3) The last strategy performs BLOCK distribution for both row and column of matrix C. Matrix A is BLOCK partitioned in rows and matrix B is BLOCK partitioned in columns. Each accelerator device can update a row-column subregion BLOCK of matrix C. For the two-dimensional matrices of the three that are stored in memory in row-major, the last two partition strategies will create array subregions that will not reside in a contiguous memory space. Data marshaling and unmarshalling will be required when mapping the array subregions to and from each device.

We observed minor performance differences among the three data distribution strategies, and thus only include the performance results of row-based partitioning strategy. This strategy requires no data marshaling or unmarshalling and the results are shown in Figure 8. Seven different matrix sizes from 1024^2 to 20480^2 are compared. Matrix with large size can no longer fit into memory space in a single GPU. Therefore, computation can be done only with multiple GPU devices assigned. The performance scales decently as the sizes of matrices and the number of GPUs increase. Because the computation dominates the total execution time (about 95% based on our breakdown profiling) in this kernel, the computations among the GPUs overlap with high rate as long as their data are copied in. However, since data copies onto multiple GPUs are mostly serialized because of the architecture limitation discussed before, overlapping data movement with computation are also impacted. Thus the optimal and linear scalability was not observed.

The Jacobi kernel in this study is a 2-D version that updates each element in a 2-dimensional matrix. Each update requires input data from the neighboring elements in four directions (or the surrounding halo regions). Jacobi computation can be distributed freely in a single dimension or

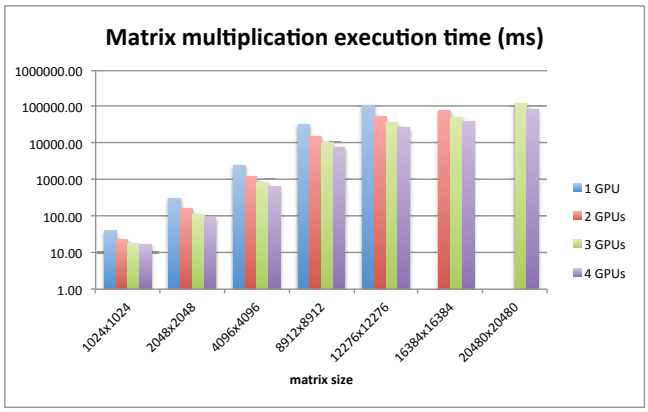

Figure 8: Performance result for matrix multiplication with row distribution

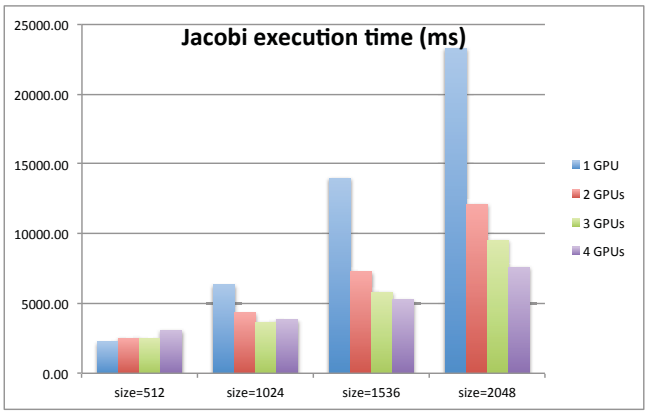

Figure 9: Performance result for 2D Jacobi

multiple dimensions under the condition that halo regions for each data subset are available. Parallelization for multiple accelerators requires data exchanges for the halo regions between devices. In this study, we choose a row-based BLOCK distribution policy to show the multiple accelerator support for Jacobi computation. Figure 9 shows the performance results. Overall, good scalability was observed from the study, especially when the data size is large. In this kernel, as shown in Figure 2, there are three major computational intensive parts; the first one is the data exchanging kernel for moving data from array `u` to `uold`, the second one is the halo region exchange operations, and the third one is the main Jacobi kernel. Our profiling experiments show each of the three kernels takes 24%, 34% and 42% of the total execution time, respectively. As mentioned earlier, the halo region exchange operation is a collective operation, which requires a barrier before the operation, and then another barrier following this operation. These two barriers in each outer iteration introduce large amount of overhead to the execution. Moreover, since halo region exchange involves more than one GPUs to communicate through the PCIe switch. It will introduce more overhead from the hardware with more number of GPUs are involved. In the latest GPU hardware, communication between GPU devices can go through GPUDirect without interfering CPU. Our runtime implementation will check peer-to-peer (P2P) communication support prior to any data movements between GPU devices. The runtime automatically switches between two communication schemes: with GPUDirect to avoid CPU overhead, or relaying data to CPU memory to complete the data transfer.

6. RELATED WORK

There has been extensive research on the single GPU support with high-level directive-based programming models. Tian et al. [19] and Reyes et al. [15] have developed two open source OpenACC compilers using two different compiler frameworks. Liao et al. [11] has included the OpenMP accelerator support in the ROSE compiler. All of this work focuses on the single accelerator support, and even the specification of OpenACC and OpenMP have not defined any support for multiple accelerators.

Although these high-level programming models lack such support, users can use the hybrid programming model to utilize multiple accelerators. Xu et al. [20] used OpenMP + OpenACC to utilize multiple GPUs attached in a multi-core system (a single node of a cluster) and explained how to do the inter-GPU communication with OpenMP synchronization and OpenACC data synchronization primitives. Hart et al. [8] used Co-Array Fortran (CAF) + OpenACC and Levesque et al. [10] used MPI + OpenACC to utilize multi-GPU in GPU cluster. The inter-GPU communication in these cases are managed using approach similar to distributed shared memory model. In this hybrid model work, explicitly calls to partition the task and move the necessary data to each GPU are required.

StarPU [3] and XKaapi [7] use a runtime library approach to scheduling tasks in multiple accelerators. StarPU relies on a cost model for scheduling tasks while XKaapi uses work-stealing. Without compiler support, users of these libraries have to manually write the offloaded code to a particular accelerator. SkelCL [18] provides a set of high-level abstractions to express computation: pre-implemented parallel patterns (skeletons) for computation and container data types for vectors and matrices. However, reduction and halo region exchange are not supported. Similarly, SkePU [5] is a C++ template library with generic skeletons and containers for multi-GPU systems. Our work uses a directive-based approach with less intrusive changes to existing codes.

There are many studies focusing on optimizing one particular type of applications for multiple accelerators. FLAME [14] uses a high-level notation to express dense linear algebra operations and a runtime scheduling system for targeting multiple CPUs and GPUs. Song et al. [17] used static data partitioning and dynamic distributed scheduling to solve dense linear algebra problems on heterogeneous GPU-based clusters. Three levels of data split were used to share work among nodes, within nodes, and between CPUs and GPUs. PARTANS [12] is an autotuning framework for stencil computation on multiple GPUs. It can automatically optimizes data distribution depending on problem size and GPU hardware features, including PCIe interconnect configurations. In comparison, our efforts aim to have high-level, general-purpose programming extensions for multiple GPUs.

OmpSs [4] can annotate an application with directives to make the original single GPU program run in parallel on multi-GPUs within a node, or across nodes in a cluster. However, the code and data region to be offloaded still need

to be programmed manually by the user. Komoda et al. [9] extended the OpenACC programming model to support multi-GPUs in a shared memory system. It proposed a new directive to specify the range of indices for one iteration of the loop so that the compiler and runtime do not need to replicate the data in all GPUs, and only move the required amount of data based on certain distribution policy. The inter-GPU communication is transparently managed by the runtime so the user does not need to consider the existence of the underlying GPUs. It also proposed another directive to solve the array reduction problem to overcome the support of only scalar reduction limitation in OpenACC model. Our work enables explicit control over data and loop distribution among multiple accelerators in OpenMP. Compared to OpenACC, OpenMP already has the worksharing support in multiple host threads. Our work demonstrates that the existing worksharing capability in a high-level programming model can be extended to target a multi-core shared memory system connected with multiple accelerators.

7. CONCLUSION AND FUTURE WORK

In this paper, we explored the language extensions required in high-level directive-based programming models to express the additional semantics for leveraging multiple accelerators within a single shared-memory system. Using OpenMP as a baseline, we proposed a set of extensions to the target-family directive to express multiple devices, virtual topology of devices, data distribution, halo region handling, and loop distribution. With these extensions, programmers can easily to exploit multiple accelerators using simple directives and clauses to annotate data and computation distribution for offloading execution. Based on an existing OpenMP research compiler, we further defined the corresponding compiler transformation and runtime support to implement the newly proposed multi-accelerator directives. Preliminary evaluation using a few kernels has demonstrated good scalability of our implementation when using multiple GPUs.

In the future, we plan to improve the performance of using multiple accelerators, such as overlapping computation and data transferring and data restructuring for improved GPU memory performance. We will also extend our work to support multiple accelerators attached to multiple networked computation nodes. Finally, we will explore how heterogeneous, single-node programming models can interact with programming models (e.g. MPI) targeting multiple nodes with distributed physical memory.

Acknowledgment

This work was supported by the National Science Foundations under Award No. CNS-1205708 and SHF-1422961. This work was also performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

8. REFERENCES

[1] Global Arrays Toolkit. http://http://hpc.pnl.gov/globalarrays.

[2] OpenACC: Directives for Accelerators. http://www.openacc-standard.org.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568. IEEE, 2012.

[5] J. Enmyren and C. W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[6] M. P. Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[7] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. *Parallel and Distributed Processing Symposium, International*, 0:1299–1308, 2013.

[8] A. Hart, R. Ansaloni, and A. Gray. Porting and Scaling OpenACC Applications on Massively-parallel, GPU-accelerated Supercomputers. *The European Physical Journal Special Topics*, 210(1):5–16, 2012.

[9] T. Komada, S. Miwa, H. Nakamura, and N. Maruyama. Integrating Multi-GPU Execution in an OpenACC Compiler. In *ICPP '13: Proceedings of the 42nd International Conference on Parallel Processing*, pages 260–269, 2013.

[10] J. M. Levesque, R. Sankaran, and R. Grout. Hybridizing S3D into an Exascale Application using OpenACC: An approach for moving to multi-petaflops and beyond. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 15. IEEE Computer Society Press, 2012.

[11] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman. Early Experiences with the OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators (IWOMP'13)*, pages 84–98. Springer, 2013.

[12] T. Lutz, C. Fensch, and M. Cole. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24, Jan. 2013.

[13] OpenMP Architecture Review Board. The OpenMP API Specification for Parallel Programming. http://www.openmp.org/.

[14] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 121–130, New York, NY, USA, 2009. ACM.

[15] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Euro-Par 2012 Parallel Processing*, pages 871–882. Springer, 2012.

[16] C. Rice University. High performance fortran language

specification. *SIGPLAN Fortran Forum*, 12(4):1–86, Dec. 1993.

[17] F. Song and J. Dongarra. A scalable framework for heterogeneous gpu-based clusters. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 91–100, New York, NY, USA, 2012. ACM.

[18] M. Steuwer and S. Gorlatch. Skelcl: Enhancing opencl for high-level programming of multi-gpu systems. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 7979 of *Lecture Notes in Computer Science*, pages 258–272. Springer Berlin Heidelberg, 2013.

[19] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. Compiling a High-Level Directive-Based Programming Model for GPGPUs, 2013.

[20] R. Xu, S. Chandrasekaran, and B. Chapman. Exploring Programming Multi-GPUs using OpenMP & OpenACC-based Hybrid Model. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1169–1176. IEEE Computer Society, 2013.

[21] R. Xu, X. Tian, Y. Yan, S. Chandrasekaran, and B. Chapman. Reduction operations in parallel loops for gpgpus. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, pages 10:10–10:20, New York, NY, USA, 2007. ACM.