# A Pattern Specification and Optimizations Framework for Accelerating Scientific Computations on Heterogeneous Clusters

Linchuan Chen Xin Huo Gagan Agrawal
Department of Computer Science and Engineering,
The Ohio State University
{chenlinc,huox,agrawal}@cse.ohio-state.edu

***Abstract*—Clusters with accelerators at each node have emerged as the dominant high-end architecture in recent years. Such systems can be extremely hard to program because of the underlying heterogeneity and the need for exploiting parallelism at multiple levels. Thus, easing parallel programming today requires not only high-level programming models, but ones from which hybrid parallelism can be extracted. In this paper, we focus on the following question: *"can simple APIs be developed for several classes of popular scientific applications, to ease application development and yet maintain parallel efficiency, on clusters with accelerators?"*.**

**We approach this problem by individually considering popular patterns that arise in scientific computations. By developing APIs for generalized reductions, irregular reductions, and stencil computations, we show that several complex scientific applications can be supported. We enable compact specification of these applications (40% of the code size of MPI), while also enabling parallelization across nodes and devices within a node, and with work distribution across CPU and GPU cores. We enable a number of optimizations that are normally implemented by hand by scientific programmers. We compare well against existing MPI applications while scaling across nodes, and against handwritten CUDA applications for executions on a single GPU, and yet can scale by using all parallelism simultaneously. On a cluster with 64 GPUs, we achieve speedups between 600 and 1800 over sequential (single CPU core) versions.**

## I. Introduction

High-end computing systems have been going through a paradigm shift as far as *intra-node* architectures are concerned. Today, four of the ten fastest supercomputers in the world (based on the latest top 500 list at the time of writing this paper) involve *accelerators* on each node, such as one of the NVIDIA GPUs or the Intel Xeon Phi system. Since consideration of scaling performance in a cost-effective and power-efficient manner will be driving the design of future high-end systems, it is easy to see that heterogeneity and increasing intra-node complexity will be key features of these systems.

The heterogeneity in current systems is creating a major obstacle in programmability and portability of high-performance applications. Particularly, programming clusters of heterogeneous nodes is very challenging, and portability across different existing and emerging heterogeneous architectures is also a very difficult problem. Consider a cluster with one (or more) GPU(s) on each node. Prior to some of the recent efforts in this area [2], [3], [9], [17], [20], [30], developing a scientific application for such an environment would have involved 1) partitioning the work across nodes, and write explicit message passing code (e.g. using MPI or a PGAS language), 2) writing GPU code using CUDA or OpenCL for each GPU, 3) partitioning the work between GPU(s) and CPU cores within each node, and explicitly program data transfers between these devices.

Though several ongoing efforts are addressing this problem, improving programmer productivity for scientific computations continues to be an ongoing challenge. Broadly, creating a high-level, general, and efficient parallel programming system remains the 'holy grail' of parallel computing. As compared to the various more general parallel programming efforts that are now considering CPU-GPU clusters [2], [3], [9], [17], [20], [30], we focus on a different problem, which is of *creating a high-level and efficient framework that can cover a reasonable variety of, but not all, scientific applications*.

More specifically, we focus on the question *"can high-level APIs be developed for several classes of popular scientific applications, to ease application development, while achieving high performance on clusters with accelerators?"*.

This paper addresses this question. We start by individually considering popular *patterns* that arise in scientific computations. For our current work, we have focused on *generalized reductions*, *irregular reductions*, and *stencil computations*, which can cover scientific applications involving structured and unstructured grids and particle simulations. Together, these three patterns are sufficient to cover a reasonable subset of scientific applications (e.g. 16 out of 23 Rodinia [5] benchmark applications), and moreover, our current system can be extended to cover more patterns in the future. We create simple and high-level APIs for expressing these computations, and implement them efficiently.

From a programmability view-point, we were able to express our target scientific applications using the APIs we have developed, and the resulting size of the code was 40% of the length of the code with MPI (versions that did not perform optimizations such as non-blocking communication or tiling and using CPU only) as an average over a set of benchmarks we used. The crucial advantage, however, is that from the same API, we can execute code on CPU-GPU clusters, by not only creating message passing code automatically, but also creating CUDA-based GPU executables and dividing the workload between CPU cores and one or more GPUs on the same node. Our work distribution and communication handling mechanisms exploit the knowledge of underlying communication pattern(s), which is exposed by our API. Furthermore, we are able to perform various optimizations automatically at runtime.

Focusing on the performance aspect, we have evaluated our framework using five different scientific applications, each

involving one or more of the patterns we have supported. For four of these applications, we obtained an MPI implementation from one of the widely distributed benchmark suites, and for two of the applications, we obtained hand-written CUDA implementations. Our evaluation study shows that distributed memory execution obtained by our framework gives comparable (in some cases even better) performance than hand-written MPI applications. The same holds true for GPU execution. But, unlike these handwritten applications, our framework allows us to obtain distributed memory, shared memory, accelerator-based, and inter-device parallelism, scaling well in each case. The optimizations supported in our framework are effective, and contribute to the performance obtained from the framework. On a 32 node CPU-GPU cluster, we obtain speedups between 600x and 1800x over sequential (single CPU core) implementations.

## II. Programming Model

Our programming model is designed with the following considerations. First, we want a high-level API that can capture most of scientific applications. Second, we want an API from which efficient execution can be achieved on accelerators, while also effectively partitioning the work across CPU and GPU cores, as well as across nodes, and moreover, without involving restructuring compiler support. The last consideration is important for us because of the need for creating a robust implementation with a moderate effort.

Achieving the above goals while also providing a very general programming system (comparable to MPI or CUDA) does not appear feasible. To address the challenge of creating high-level APIs for scientific computations, we derive inspiration from summarization of scientific computing kernels as *dwarfs* by Phil Collela (and as further generalized in Berkeley's landscape on parallel computing [13]). Their observation has been that scientific applications follow one of the seven templates or patterns: structured grid or stencil computations, unstructured grids, N-body simulations, generalized reductions, dense linear algebra, sparse linear algebra, and spectral methods (FFTs).

Within this set of templates or patterns, we further observe that standard libraries are available and popularly used for dense and sparse linear algebra and FFTs. Thus, our claim is that a dominant fraction of popular scientific applications developed today using a general purpose parallel model (e.g., MPI or one of the PGAS languages) involves one or more of the following patterns: structured or unstructured grids, generalized reductions, or N-body interactions. Further, N-body problems can be expressed using indirection arrays, just like unstructured grids, which we will show through our work with a mini-application called MiniMD later. Thus, we have focused on three patterns, which can in fact be used for 16 out of 23 Rodinia benchmarks [5]. Moreover, our work can be extended in the future to support additional patterns, and thus a wider class of applications.

### A. Communication Patterns and APIs

In this section, we introduce the supported communication patterns and the corresponding APIs, which are summarized in Table I.

**Generalized Reductions:** these arise in a variety of data-intensive and scientific applications. They have been supported by existing HPC programming systems, e.g., OpenMP [10] provides a reduction clause for combination of computation results from different threads, although the reduction is only applicable to individual scalar variables (in the case of most compilers).

| **User-defined functions for generalized reductions** |
|---|
| `void (*gr_emit_fp)(Object *obj, void *input, size_t index, void *parameter)`<br>generates (a) key-value pair(s) based on one input unit (starting at `index`), and inserts the key-value pair(s) to the reduction object `obj` |
| `void (*gr_reduce_fp)(VALUE *dst, VALUE *src)`<br>reduces `src` to `dst`, the pointer to a value in the reduction object |
| **User-defined functions for irregular reductions** |
| `void (*ir_edge_compute_fp)(Object *obj, EDGE edge, void *edge_data, void *node_data, void *parameter)`<br>processes an edge, generates (a) key-value pair(s), and inserts the key-value pair(s) to the reduction object `obj` |
| `void (*ir_node_reduce_fp)(VALUE *dst, VALUE *src)`<br>reduces `src` to `dst`, the pointer to a value in the reduction object |
| **User-defined functions for stencil applications** |
| `void (*stencil_fp)(void *input, void *output, int *offset, int *size, void *parameter)`<br>processes a single element in the input grid and writes the result to output grid |

TABLE I: User-defined Functions for Communication Patterns

The first row of Table I shows the signatures of user-defined functions for generalized reductions - while the API is similar to MapReduce [11], [6], [7], there are some differences also for achieving higher efficiency. The API involves two types of operations: *emit* and *reduce*. The *gr_emit_fp* function is invoked to process the smallest input unit, and generates one or more *key-value* pairs. The first parameter of the *gr_emit_fp* function is a system-defined data structure, referred to as the *reduction object*, which is implemented as a hash table with support for parallel key-value insertion, and is used to accumulate the reduction result. Parameter `index` is generated by the runtime system, and captures the start location of the particular input unit. Users could pass extra information (e.g. cluster centers for Kmeans) through `parameter`. The *gr_reduce_fp* function indicates the operation conducted (typically commutative and associative) while inserting one key-value pair to the reduction object. This function is also invoked while global combination is being conducted.

```
Real X(numNodes), Y(numEdges);   ! node/edge data arrays
Integer IA(numEdges,2);          ! indirection array

for (i = 0; i < numEdges; i++) {
    X(IA(i,1)) = X(IA(i,1)) op Y(i);
    X(IA(i,2)) = X(IA(i,2)) op Y(i);
}
```

Fig. 1: An Irregular Reduction Loop

**Irregular Reductions:** Irregular reductions arise in the context of unstructured grids, and in some of the implementations of N-body problems. Nodes in unstructured grids are connected by edges explicitly, and one can think of the computations in terms of *nodes* and *edges*. For example, the input data comprises the node data, the values associated with the edges, and array(s), referred to as the *indirection* array(s) (edges), which stores the end points (nodes) that are connected by each edge.

The code listed in Figure 1 shows a typical irregular reduction loop, denoted computations performed by iterating over all edges. Each iteration updates two elements in array $X$ using commutative and associative operations *op*, with accesses

through the indirection array $IA$. Because each node may be connected by multiple edges, the update of the elements in $X$ involves reductions. Overall, the key distinctive property of irregular reductions are the accesses of the node data using indirection arrays.

We now discuss how a high-level API can be used for irregular reductions. The *ir_edge_compute_fp* function is invoked to process an edge, the lowest level of processing granularity. Like generalized reductions, a system-defined reduction object is provided to store the reduction result. Among the parameters of this function, `edge` is the single edge being processed, and includes the edge ID, as well as identifiers of the nodes it connects. `edge_data` stores attributes associated with `edge`. To update a node, the user needs to pass the node ID as the key to the `insert` function of reduction object. Function *ir_node_reduce_fp*, a *node reduce function*, works in the same way as the *reduce* function in generalized reductions, i.e., it is invoked to update the reduction result of a node in the reduction object.

The API we have developed is simplified (and operates correctly) only with a certain types of partitioning of the unstructured grid [18]. We refer to the edges as the *computation space* and the nodes (being updated) as the *reduction space*. The *inter-process workload partitioning* strategy we applied is based on the reduction space, which involves the following steps: 1) Divide the nodes into equal partitions, 2) Group the edges: if both nodes of an edge belong to the same partition, this edge is exclusively assigned to this partition; if an edge is connecting nodes belonging to different partitions (say partitions 0 and 1), then this edge is assigned to both processes. To avoid race conditions, when an edge is being processed, only the node(s) belonging to the current partition is updated. This partitioning method allows each process to update its own set of nodes, disjoint from other nodes, which also improves the node access locality. Note that the above partitioning scheme can be (and is) applied at multiple levels, across nodes, across devices, and even within a device to improve locality.

**Stencil Computations:** Stencil applications involve computations of updating each element in the input based on the values of its neighbor elements. The input for stencil applications is usually a structured grid or matrix with two or higher dimensions.

The standard function we support is *stencil_fp*. It defines a stencil operation that is applied to a single element in the grid. The API also provides *get functions* like `GET_FLOAT3` and `GET_INT2`, which are pre-defined macro functions for referencing elements from input or output with a certain number of dimensions: the arguments to these functions include the data buffer to be accessed and the coordinate (offset) of the element to be accessed. The return value is the reference to that element. Parameter `size` is implicitly passed to the *get functions* for the runtime to compute the absolute position of an element in the input/output buffer.

### *B. Case Study: A Comprehensive Example Involving Multiple Communication Patterns*

We now use an example to show how our programming model is used to parallelize an application that involves different communication patterns. The application we use here is Molecular Dynamics (*Moldyn*) [19], which simulates the interaction and motion of molecules in a period of time. It involves an irregular reduction kernel for computing the forces among molecules (CF) based on the interactions (edges) among the nodes. In addition, two kernels computing the kinetic energy (KE) and average velocity (AV), respectively, are generalized reductions.

Listing 1 shows the user-defined functions. `force_cmpt` and `force_reduce` are the functions for the irregular reduction kernel CF. Function `force_cmpt` is an *edge compute function*. It is invoked by the irregular reduction runtime for CF kernel to compute the distance and force between the two nodes connected by an edge. If the distance of the two nodes is within a threshold, it uses the node ID as the key, and the force as the value, to update the reduction result of each node. *node reduce function* `force_reduce` indicates that the reduction operation applied to the reduction result is simply accumulation. Similarly, functions `ke_emit` and `ke_reduce` are the user-defined functions for the kernel KE, and `av_emit` and `av_reduce` are for the kernel AV. Details of these four functions are not shown to keep the code listing short.

*Listing 1: Moldyn: User-defined Functions*

```
1  //user-defined functions for CF kernel
2  DEVICE void force_cmpt(Object *obj, EDGE edge,
3    void *edge_data, void *node_data, void *parameter){
4    /*{compute the distance between nodes...}*/
5    if(dist < cutoff){
6      double f = compute_force(
7                 (double*)node_data[edge[0]],
8                 (double*)node_data[edge[1]]);
9      obj->insert(&edge[0],&f);
10     f = -f;
11     obj->insert(&edge[1],&f);
12   }
13 }
14 DEVICE void force_reduce(VALUE *dst, VALUE *src){
15   *dst += *src;
16 }
17 //user-defined functions for KE kernel
18 DEVICE void ke_emit(...){...}
19 DEVICE void ke_reduce(...){...}
20 //user-defined functions for AV kernel
21 DEVICE void av_emit(...){...}
22 DEVICE void av_reduce(...){...}
```

*Listing 2: Moldyn: Invoking System API*

```
1  MPI_Init(...);
2  Runtime_env env;
3  env.init();
4  //runtime for irregular reduction CF
5  IReduction_runtime *ir = env.get_IR();
6  //runtime for generalized reductions KE & AV
7  GReduction_runtime *gr = env.get_GR();
8  //Compute Force (CF) kernel
9  ir->set_edge_comp_func(force_cmpt); //use force_cmpt
10 ir->set_node_reduc_func(force_reduce); //use force_reduce
11 /*{set edge and node data filenames ...}*/
12 for(int i = 0; i < n_tsteps; i++){
13   ir->start();
14   // get local reduction result
15   result = ir->get_local_reduction();
16   // update local node data
17   ir->update_nodedata(result);
18 }
19 /*{set input filename}*/
20 //Kinetic Energy (KE) kernel
21 gr->set_emit_func(ke_emit); //ke_emit as emit func
22 gr->set_reduc_func(ke_reduce); //ke_reduce as reduce func
23 ...
24 gr->start();
25 double ke_output = (gr->get_global_reduction());
26 //Average Velocity (AV) kernel
27 gr->set_emit_func(av_emit); //av_emit as emit func
28 gr->set_reduc_func(av_reduce); //av_reduce as reduce func
29 ...
30 env.finalize();
31 MPI_Finalize();
```

Listing 2 shows how the system API should be invoked to

create a runtime environment and runtime instances for the kernels. Line 2 creates a *runtime environment*, which is then used to create runtime instances for different communication patterns. In this example, Line 5 and Line 7 create two runtime instances, for irregular reductions and generalized reductions, respectively. Line 9 to Line 18 is the code for the irregular reduction kernel CF, and Line 19 to Line 29 is the code for the two generalized reduction kernels KE and AV. If multiple kernels follow the same communication pattern in an application, the same runtime can be reused, by resetting configuration parameters (e.g., the user-defined function names and input data). In this example, generalized reduction runtime `gr` is used by both KE kernel and AV kernel. Alternatively, users could also define separate runtime instances for KE and AV.

### C. Discussion

Our framework has certain limitations, both in terms of the patterns it can handle and how these patterns are specified. The key thing to note, however, is that use of our framework does not prevent invocation of other libraries since our framework is simply based on C/C++. Thus, for applications that have certain other patterns in addition to the ones we support, we can still accelerate code development while allowing users to manually program other sections or use other libraries.

Some of the other limitations are as follows. Although multiple kernels of a same communication pattern in an application can be supported by our framework, only a single target object can be processed every time a runtime instance is launched. For example, only one grid can be processed in a particular stencil runtime execution, and similarly, only one set of edges can be processed in every irregular runtime execution. One requirement we impose is that function headers should begin with a *system-defined macro*, which is converted to device specific qualifiers at compile time. Our future work will address these limitations.

## III. Framework Implementation and Pattern Specific Optimizations

Our framework is designed for clusters with heterogeneous CPU-GPU nodes. We also allow each node to have multiple GPUs.

### A. Code Organization and Compilation

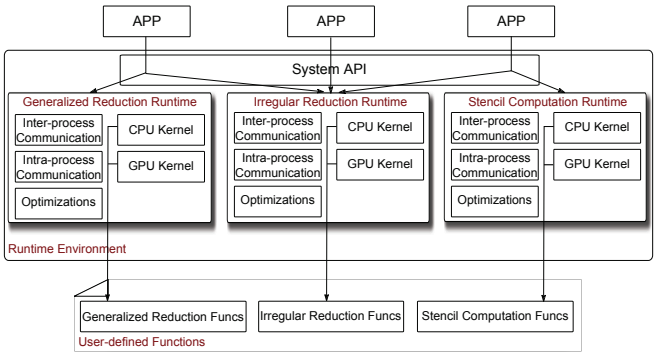



Fig. 2: Code-level Structure of the Framework

We describe the organization of our framework using Figure 2. The application logic itself is specified by writing implementation of functions. Although we can execute an application utilizing both CPU and GPU cores, the user needs to write only one set of sequential functions processing a smallest input unit. The source code of the user-defined functions is invoked by both CPU and GPU kernels. Because function pointers on one device could not be used by other devices, the user code is copied in both CPU GPU namespaces. Besides the two kernels, what are compiled together also include both inter-node and intra-node communication libraries, along with the optimizations written specifically for each communication pattern. The rest of this section describes the implementation of these.

### B. Runtime Execution Flow

We create one MPI process for each node, and intra-node parallelism is exploited through CPU multi-threading (pthread) and GPU *co-processing*. Inter-process communication and GPU kernel launches are handled by the CPU threads. Compared with one process per core and one process per GPU model, our approach improves the inter-core data access locality, reduces memory requirements, and avoids unnecessary costs of inter-process (but intra-node) communication or memory copies.

Inter-process workload partitioning is conducted in a distributed manner: each process fetches only the partition that belongs to itself. The input loading might involve extra processing. In irregular reductions, besides loading the local node partition, each process also inspects all the input edges and pick the ones that connect local nodes, to form the local computation space. Stencil computations need to allocate extra space for each sub-grid, as *halo* regions are needed for border processing. At the end of the processing, a *global reduction* is conducted for generalized reductions. The global reduction is performed among the processes in a parallel binary tree order, so that up to $log(n)$ parallel reduction steps are needed to form a final result, where $n$ is the total number of processes. Whereas for irregular reductions, since the nodes from the input are partitioned into independent spaces, there is no need to do a global reduction. Reduction results from different processes are written back to disk as separate parts. Similarly, for stencil computations, results of the sub-grids are also directly written back to disk in a distributed manner.

### C. Distributed Memory Execution

Executing an application across nodes involves challenges of inter-process data partitioning and performing data exchange (communication) during the execution.

**Workload Partitioning and Inter-process Communication for Generalized Reductions:** Because there is no dependence among loops of generalized reductions, the input data is evenly partitioned among processes. The only inter-process data exchange is the final combination of the reduction objects from different processes.

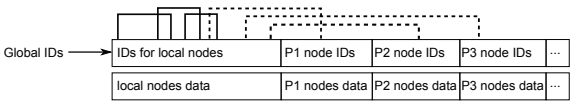



Fig. 3: Arrangement of Nodes for Irregular Reductions. Solid lines: local edges, Dashed lines: cross edges

**Inter-process Data Exchange for Irregular Reductions:** The inter-process data partitioning method we use has already been introduced. Data exchange is only needed before

execution of each reduction loop. It is handled as follows. At the time of loading input, the nodes are equally divided and each process fetches its own partition. Each process also inspects the edge input, and associate corresponding edges to its own partition, as is stated in Section II-A. Cross edges in one partition need to access nodes from remote processes. We call these nodes *remote nodes*. After receiving its edge partition, each process inspects the cross edges, and groups the remote node IDs. Figure 3 shows the arrangement of local nodes and remote nodes. The local nodes are stored continuously in the front. Remote nodes from the same process are placed continuously. Initially, the two nodes of each edge have IDs with global node rank. The runtime on each process converts these IDs into the local rank. A global ID array is maintained for node data exchange. Every time node data is updated (reset) by the user program, data exchange should be conducted. The following steps show what a process does during the node data exchange: 1) Send a request to each remote process - The request message contains the number of nodes it is requesting. 2) Receive the requested number of nodes from each remote process and allocate receive buffers for requested node IDs. Allocate send buffers for node data requested by each remote process. 3) Send global remote node IDs (shown in Figure 3) to corresponding processes. 4) Receive requested node IDs from each remote process to the corresponding receive buffers. 5) Copy corresponding node data into the node data send buffer for each remote process and send each node data buffer to remote process, and 6) Receive remote node data from each remote process and store them in the corresponding node data slot as shown in Figure 3.

Steps 1 through 4 are only performed at the start of the kernel and after the edges (connections) are reset. Otherwise, the remote node ID information is not changed and only data should be exchanged (Steps 5 and 6). One of the key features of our implementation is support for *overlapped execution*. We compute local edges and cross edges separately. The former only depends on local nodes and we let its execution run concurrently with the exchange of remote node data (Steps 5 and 6).

**Grid Partitioning and Data Exchange for Stencil Computations:** Our runtime system expects users to provide certain important parameters, such as the dimensionality and size of the global grid and the virtual processor Cartesian topology on which the grid is going to be decomposed. The stencil runtime system divides the global grid equally according to these parameters, and distributes the sub-grids to corresponding processes.

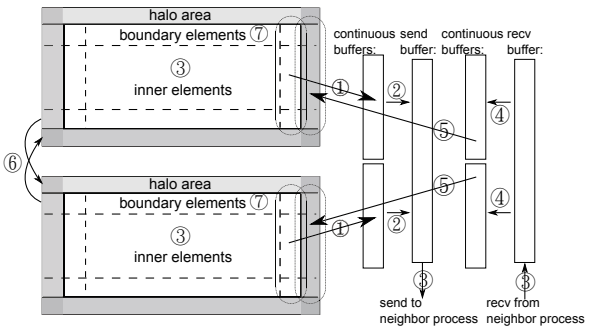



Fig. 4: Time Sequence of the Stencil Runtime for Heterogeneous Executions

Another important issue to be handled by the stencil runtime system is the data exchange. For convenience, data exchange at both process and device levels are discussed together here. Figure 4 shows two device grids inside a process. For simplicity, we use 2D grids to show how data exchanges and computations are conducted. Data exchanges among neighbor processes are conducted on each dimension, but here we only show the dimension with non-contiguous boundary, which is the most complex case.

***Steps for Inter-process Data Exchange:*** Step 1 copies non-contiguous borders to contiguous buffers, which are in turn copied to the process's send buffer. For the CPU, the non-contiguous boundary elements can be copied to the send buffer in strips using `memcpy`. However, for a GPU, non-contiguous device memory is not able to be copied out directly using `cudaMemcpy`. We allocate a host mapped (zero-copy) memory buffer, and launch a kernel to copy such a boundary into the buffer. The buffer is then directly copied into the process's send buffer (Step 2). Similar copy procedures are conducted on other non-unit-stride access boundaries of each grid. When the send buffers for all boundaries are ready, the runtime launches asynchronous MPI send and receive for the boundary exchange. At the same time, the computation of inner elements is launched, which runs *concurrently* with the inter-process data exchange. When the data exchange and inner computation finish, contents in the receive buffer are copied to each sub-grid's *halo* area. For GPU, data is first copied to host mapped buffer (Step 4), after which, a GPU kernel is launched to copy the buffer into the halo area on the device (Step 5).

***Inter-device Data Exchange:*** After the inter-process data exchange is finished, boundaries between neighbor devices are exchanged (Step 6). Exchanges between CPU and GPU use `cudaMemcpy`. For GPU-GPU exchanges, an efficient device-to-device asynchronous memory copy, `cudaMemcpyPeerAsync` is used, which supports concurrent bi-directional transfer on the PCE-e bus. Finally, the boundary elements are processed (Step 7).

### *D. CPU-GPU Workload Partitioning*

Due to the heterogeneity of cores within each node, we cannot evenly partition the workload across them. The processing speed of each device not only depends on its own hardware configurations, but also the properties of the application. We apply different intra-process partitioning strategies according to the communication patterns, to minimize load imbalance while keeping the overhead of scheduling small.

**Dynamic Scheduling and Overlapped Execution for Generalized Reductions:** Because generalized reductions typically have only one or a small number of time steps, input data copied to the GPU is typically used only once. The data copy, if not overlapped with the computation, could significantly influence the overall performance. Also, to avoid load imbalance, dynamic workload scheduling is necessary. Thus, we schedule tasks on devices in chunks, and devices obtain chunks by pthread locking. Each CPU core continuously receives chunks to process. The task retrieval and kernel launches of GPUs is controlled by a CPU thread and two streams are created for each GPU. Each time, the controlling CPU thread retrieves a task chunk for each GPU, and splits the chunk into two smaller blocks allocated in pinned memory. Sequentially for each stream, asynchronous memory copy of input from pinned memory to device memory is then launched, followed by the kernel launch. When all the GPUs finish processing the task blocks in both the streams, the controlling thread gets another task chunk.

**Adaptive Workload Partitioning for Irregular Reductions and Stencil Computations:** Unlike generalized reductions, irregular reductions and stencil computations usually involve many time steps. Thus, we can profile the relative processing speed of each device (specific to the particular application) and utilize the profiling information for static workload partitioning.

***Partitioning for Irregular Reductions:*** After edge and node partitions are organized on each process, the runtime conducts *inter-device partitioning* among the devices. Irregular reductions involve a large number of iterations, and edges (connectivity) can be unchanged for many time steps. Thus, if dynamic scheduling is used in each time step, data chunks have to be reloaded from the host by GPUs repetitively, leading to high I/O overhead. In view of this, we utilize an *adaptive partitioning*. In the first time step, the reduction space is evenly partitioned. This creates nearly equal-sized edge partitions for each device. We profile the processing speed of each device, and adjust the splitting ratio: let $S_i$ be the speed of device $i$, and $N$ be the total number of nodes, then this device will be assigned $N \times \frac{S_i}{\sum_{k=1}^{n_{dev}} S_k}$ number of nodes. The workload is re-partitioned in the second time step according to the new ratio. The newly grouped edges are copied to device memory on each GPU, which would be kept unchanged for a number of iterations until the edges are rebuilt. In addition, the node data has a full copy on each device. Although every time nodes are updated, they need to be updated to each device, the size of node data is much smaller compared with edges. Maintaining a full copy of node data also avoids inter-device data exchange.

***Partitioning for Stencil Computations:*** The sub-grid partitioned to a process is further partitioned along the highest dimension, and a sub grid is assigned to a multi-core CPU or a many-core GPU. Similar with irregular reductions, the partitioning is conducted adaptively: the first time step utilizes even partitioning, and determines the relative speed of each device. The grid is re-partitioned in the second iteration based on the speeds of devices.

### *E. GPU and CPU Execution*

We now focus on the problem of efficient execution on each device. The performance of an application on a GPU is typically bottle-necked by the memory accesses. The workload partitioning and execution strategy for each communication pattern in our system extensively utilizes the GPU's memory hierarchy, with the knowledge of specific properties of each pattern. Some of the same challenges also arise for execution on CPU cores. We now describe the key optimizations performed in our system.

**Reduction Localization for Generalized Reductions and Irregular Reductions:** For both generalized reductions and irregular reductions, to avoid race conditions, parallel updating of the reduction objects is ensured by locking (implemented as *atomic* operations). If the number of distinct keys is small, the contention among the threads for updating the values associated with one key can be large, leading to significant waiting times.

For generalized reductions, the runtime system allows for users to configure the size of the reduction objects. If reduction objects are small enough, the runtime system stores them in the shared memory on each SM, which supports much faster synchronization and I/O. Local reductions are first performed in shared memory on each SM, and the reduction results from all SMs are merged to the reduction object in the device memory to form the final GPU reduction result. The runtime system creates multiple reduction objects in the shared memory if the size of each object is small enough, and each reduction object is updated by a subset of the threads in one thread block, which further reduces the contention overhead. This technique is also applied at the CPU side: each CPU core has a private reduction object, which is used for local reduction. A final combination is performed when the computation ends.

For irregular reductions, the *reduction space based* input partitioning scheme we discussed earlier enables the utilization of the shared memory on GPUs. Edges and nodes assigned to each device are further partitioned. The partitioning module determines the number of partitions based on the size of GPU's shared memory and the size of each reduction element, so that each partition of the reduction space can be put in the shared memory: $num\_parts = num\_nodes / \left(\frac{shared\_memory\_size}{reduction\_element\_size}\right)$. At CPU side, workload is partitioned in a similar way, and CPU cores process the workload in partitions. Such a partitioning greatly improves the data locality. The reduction results from different SMs/CPU cores are simply concatenated to form the global reduction result. There is no combination involved because the reduction space was split into independent partitions.

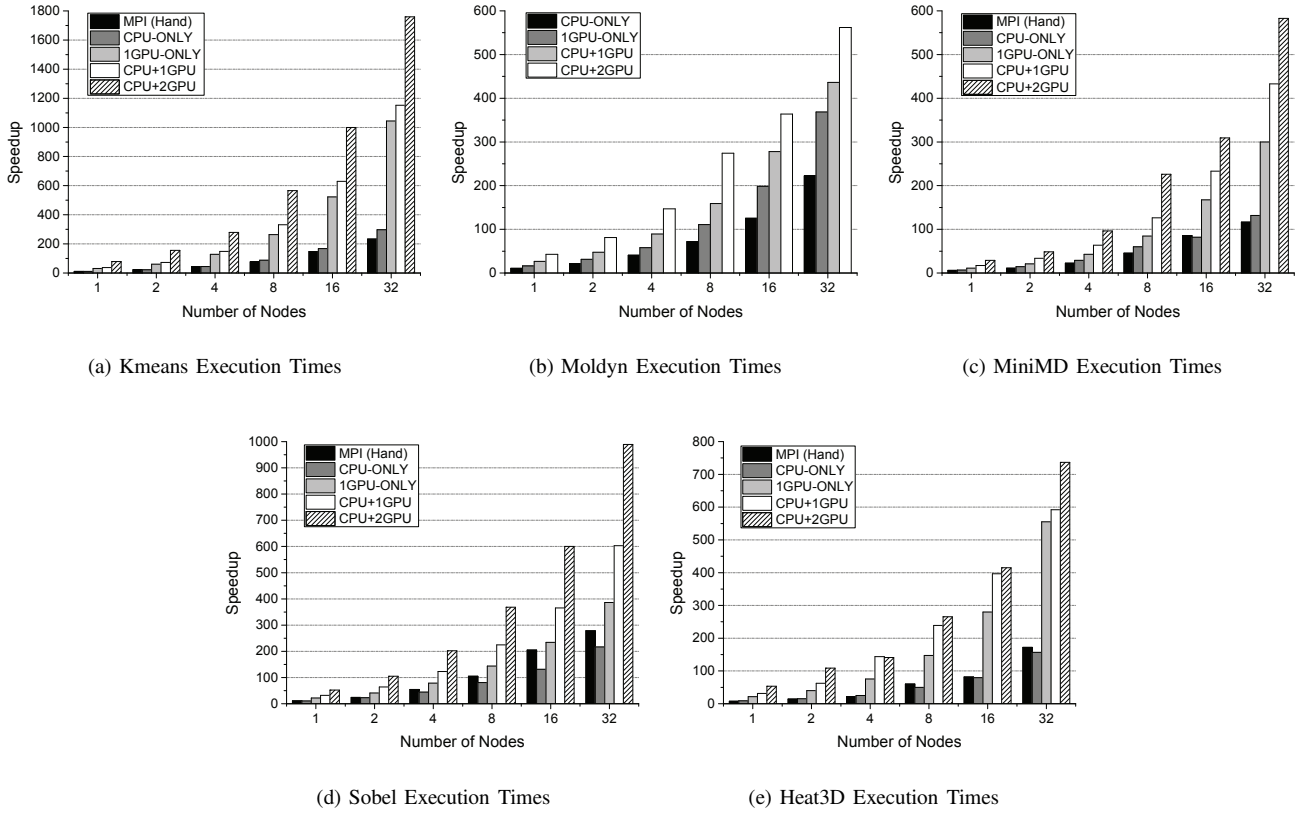
**Grid Tiling for Stencil Computations:** The part of the grid that each device processes is tiled along every dimension. Each CPU core or a GPU SM processes grid elements in tiles. The inner element tiles and boundary element tiles are grouped separately. Because the computation of inner elements only depends on local data, inner tiles are processed *concurrently* with inter-process data exchange. The boundary tiles are processed after the data exchange finishes. Grid tiling is beneficial for two reasons: 1) it improves the cache performance for the neighbor element access. 2) the boundary plane tiles are grouped together, which enables a single kernel launch on GPUs to process all the boundary planes, without launching separate kernels for each plane.

The default configuration of the on-chip memory on Fermi GPUs is configured as 48KB shared memory and 16KB L1 cache, which is used for generalized reductions and irregular reductions. The stencil runtime changes the configuration into 48KB L1 cache with 16KB shared memory by setting `cudaFuncCachePreferL1` flag, which benefits the cache performance for stencil computations. We are currently using fixed tile width, 16 and 32 for CPU and GPU, respectively.

## IV. Experimental Results

Our target platform is a cluster with discrete CPU-GPU nodes. For evaluating our system on such a target architecture, we used a CPU-GPU cluster, which consists of 32 nodes, with each node having a 12 core Intel Xeon 5650 CPU and 2 NVIDIA M2070 GPUs (thus, 64 GPUs in all). Each node has a system memory of 47 GB, and each GPU has a device memory of 6 GB. We used mpicxx for MVAPICH2 version 1.7, and nvcc 5.5.0 to compile all the codes, with *-O3* optimization enabled.

We had the following goals in our experiments. 1) Evaluate the overall effectiveness of the framework, particularly, assessing how it can help scale applications across nodes in a cluster, by use of CPU and GPU cores, and by even using multiple GPUs within a node, 2) Compare the performance of the applications using the framework against existing hand-written benchmarks, including both MPI applications (for distributed memory performance and scalability) and CUDA applications (for GPU performance), and 3) Quantify the benefits from




(a) Kmeans Execution Times

(b) Moldyn Execution Times

(c) MiniMD Execution Times

(d) Sobel Execution Times

(e) Heat3D Execution Times

Fig. 5: Intra-node and Inter-node Scalability and Comparison Against Hand-written MPI Applications

optimizations supported in the system. The execution times we report here include workload distribution time and computation time, i.e., the application setup and initialization times are not included.

### A. Applications Used

Five applications of varying complexity and involving one or more patterns were chosen for our evaluation study.

**Kmeans:** Kmeans is a simple data mining application involving generalized reductions. We used a three-dimensional dataset with 40 centers (200 Million points - 2.3 GB). We report the execution time with execution of a single iteration. For comparison with an existing MPI implementation, a widely distributed kernel was taken[1].

**Moldyn:** This application simulates interactions among molecules in a three-dimensional space. The primary pattern here is irregular reduction, but it involves generalized reductions as well. A dataset with 1 million nodes and 130 million edges was used and the application was executed for 1000 iterations. Because we did not find a comparable hand-written MPI application, we only focus on performance with our framework.

**MiniMD:**. MiniMD is a *mini-app* developed by the Mantevo project at Sandia National Labs. The goal of the Mantevo project is to create representative applications (a few thousand lines of code) of full-scale scientific applications. Particularly, MiniMD is a smaller version of a large-scale software LAMMPS. MiniMD simulates the motion of a large number of atoms (or molecules) by computing the position and velocity of each atom in the system throughout a predefined number of time-steps. Though it follows a different algorithm than the Moldyn application, its major kernel, force computation, is also an irregular reduction kernel. It also includes a number of generalized reduction kernels, such as those for energy computation. The simulation was conducted for 1000 iterations in our experiments and 500,000 atoms (double precision) were simulated. The MPI implementation we used is the one developed by the Mantevo project (the code also uses OpenMP for shared memory parallelization).

**Sobel:** Sobel edge detection is an application used in image processing. Two $3\times3$ masks are convolved to the input matrix with two-dimensional points. Thus, the application is a 9-point stencil application. A $32{,}768\times32{,}768$ floating point matrix was used as the dataset. MPI benchmark used for comparison was obtained from the *UPC Benchmarking Suite*, developed at the George Washington University[2]. Edge detection was conducted for 15 iterations.

**Heat3D:** This code simulates heat transmission in a 3D space, involving a 7-point stencil. An existing MPI implementation is widely distributed and used[3]. A $512\times512\times512$ grid with double precision elements was used, and results reported are from 100 iterations.

### B. Code Sizes

As shown in Figure 6, we compared the code sizes between implementations as part of our framework and available MPI programs. The ratios are 0.53, 0.37, 0.40, and 0.28 for

[1] http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html

[2] http://upc.gwu.edu/download.html

[3] http://dournac.org/info/parallel_heat3d

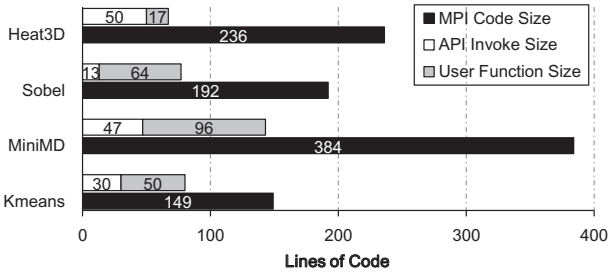



Fig. 6: Comparison of Code Sizes between Handwritten (MPI) Benchmarks and Versions Using our Framework

Kmeans, MiniMD, Sobel, and Heat 3D, respectively. In other words, over these four applications, we have an average of 40% of the code size of MPI implementations. A more detailed examination shows that our framework is reducing the coding effort spent on workload partitioning and communication. The reduction in code size varies as the code ratio for initialization and resetting are different. It is also important to emphasize that unlike MPI implementations, the code written with our framework can also utilize accelerators like GPUs.

### C. Overall Performance from the Framework

Our first set of experiments focuses on how our framework scales while we increase devices on each process and while we increase the number of processing nodes. A comparison of distributed memory scalability against existing MPI implementations was also performed, using four applications.

Figure 5 shows the results of scaling our framework on a heterogeneous cluster with 5 applications. In all cases, CPU-only execution uses all 12 cores on each node, i.e., we use between 12 and 384 cores for our experiments. The speedups shown in the figure are against the executions on a single core of the CPU.

We first focus on the benefit of utilizing GPUs together with the CPU cores. For Kmeans, the GPU is 2.69 times faster than 12-core CPU. After combining the computing power of 12 CPU cores with one GPU, a speedup of 1.20 is achieved over 1-GPU execution, or a speedup of 3.23 over 12-core CPU execution. Combining the second GPU also results in a 1.92x speedup over the 1-GPU execution, or a speedup of 5.16 over 12-core CPU execution. Similar inter-device scalability could be seen from the other four applications, although the CPU-GPU execution time ratios are different for different applications. For Moldyn, the GPU is 1.50x faster than the CPU. CPU + 1GPU outperforms GPU-only by 1.54x, and CPU + 2GPU brings a 1.64x incremental speedup. For MiniMD, GPU is 1.70 times faster than CPU, and the CPU + 2GPU achieves an average speedup of 3.89x over using CPU only. For Heat3D, GPU is 2.4x faster than CPU, and the average speedup of using CPU together with two GPUs over using a single CPU on each node is 5.5x. Similarly, for Sobel, CPU + 2GPU is 4.68x faster than CPU-only. The performance of stencil applications Sobel and Heat3D varies for different decompositions, and we report the best execution results here.

The overall speedups from utilizing 32 nodes and all the cores on each node vary between 562 and 1760 for different applications. Kmeans has the maximum overall speedup, as the GPUs are much faster than the CPU due to the effective usage of shared memory for generalized reductions, and there is a negligible communication overhead. The other four applications achieve a lower speedup of near or above 600, with Sobel near 1000, because of limited speedup from GPUs and various overheads.

To quantify the scheduling (including the possibility of load imbalance), synchronization, and communication overheads among devices, we can calculate the *perfect speedups* of CPU+1GPU and CPU+2GPU executions over the CPU-only execution. This calculation uses the relative performance of each device, and ignores any overheads. In other words, we assume that work can be perfectly distributed with no imbalance, scheduling costs, synchronization, or communication. We compare the actual execution speedups with the perfect speedups in Table II. We can see that in most cases, the actual speedups are close to the perfect speedups. On average, we are achieving 89% of the perfect performance for CPU+1GPU executions and 88% for CPU+2GPU executions, indicating that different computing units are being effectively used by our framework.

| | | Speedup Over CPU-only (x) | | | | |
|---|---|---|---|---|---|---|
| | | Kmeans | Moldyn | MiniMD | Sobel | Heat3D |
| CPU+1GPU | Perfect | 3.69 | 2.5 | 2.7 | 3.24 | 3.4 |
| | Actual | 3.23 | 2.31 | 2.15 | 2.94 | 3.2 |
| CPU+2GPU | Perfect | 6.38 | 4 | 4.4 | 5.48 | 5.8 |
| | Actual | 5.16 | 3.79 | 3.89 | 4.68 | 5.5 |

TABLE II: Comparison of Intra-node Scalability: Perfect (Assuming No Scheduling Overheads) and Actual

We next focus on distributed memory scalability of our implementation and compare it against that of hand-written MPI applications (for four of the five applications). For a fair comparison, the following discuss focuses only on the CPU-only execution results from our framework. For Kmeans, Sobel, and Heat 3D, the hand-written MPI applications involve one MPI process per CPU core. In the case of Heat3D, framework achieves an average speedup of 8% over the corresponding MPI code, whereas for Sobel, there is a 11% slowdown. The reason for the slowdown with Sobel is that our stencil runtime system spends extra cycles on computing the offsets while processing the elements. Optimizations such as tiling and overlapping communication with computation hide these overheads to an extent. For Kmeans, our framework is 1.05x faster, on average, because of the use of light-weight threads instead of separate MPI processes. For MiniMD, we are 1.17x faster, which is because of overlapping communication and computation. Overall, within our framework, the speedup in going from 12 cores (1 node) to 384 cores (32 nodes) ranges between 20 and 26.

It should be noted that some of the optimizations that are helping our framework can certainly be applied on MPI (though the handwritten versions we found did not use them). However, such optimizations within MPI will be at the cost of additional programmer effort, whereas our framework applies them automatically. Again, it is important to emphasize that our framework can also benefit from accelerators.

### D. Benefits from Optimizations

The design of our APIs enables a number of pattern specific optimizations. We now focus on showing the effect of overlapped execution of irregular reductions and stencil computations (for distributed memory execution), as well as the benefit of applying tiling to stencil computations.

Figure 7 shows the effect of tiling (for stencil computations) and overlapped execution (for both stencils and irregular reductions) with different number of nodes (one application for each pattern). The execution on each node uses the 12

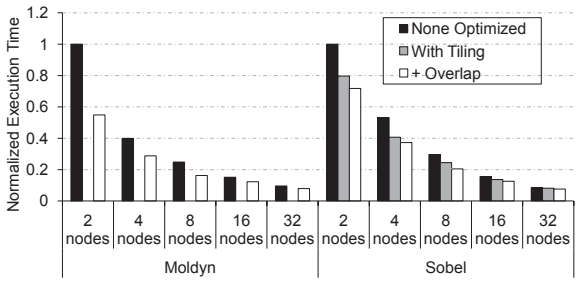



Fig. 7: Effect of Optimizations for Irregular Reductions and Stencil Computations

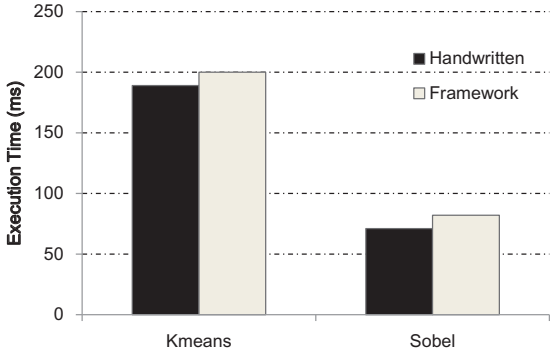



Fig. 8: Comparison with Benchmarks on a Fermi GPU

available CPU cores and the 2 GPUs. The tiling optimization increases the performance of Sobel by up to 20%. Overlapping the asynchronous communication with computation benefits the overall performance significantly. On average, overlapped execution is 37% and 11% faster than non-overlapped execution for Moldyn and Sobel, respectively.

### *E. Comparison with GPU Benchmarks*

To the best of our knowledge, no hand-written codes are available that can use both CPU and GPU cores. Thus, to examine the efficiency of our framework, we compare the code executed by our framework against the hand-written CUDA codes, with a single Fermi GPU on a single node. Among the five applications we have used, we could obtain comparable hand-written versions for Kmeans and Sobel only. The Kmeans benchmark was obtained from the Rodinia benchmark suite [5]. 10 Million points were used for the test. The Sobel code was distributed with the NVIDIA SDK. We use an 8,192$\times$8,192 image as the input.

The results are shown in Figure 8. Kmeans using our framework is only 6% slower than the hand-written benchmark. The CUDA version Sobel from CUDA SDK is optimized to use texture memory for staging the input. Our framework cannot perform such application-specific optimizations, and results in 15% slower execution.

## V. Related Work

The topic of parallel programming for scientific applications has been extensively studied over the last 3 decades. We restrict ourselves to efforts that focus on clusters with accelerators.

One of the set of efforts in this area has been in context of OpenCL. SnuCL is an extension of OpenCL to support explicit message passing [20]. LibWater [17] is a relatively higher-level framework, where users setup communication using *buffers*, and their runtime system invokes MPI communication implicitly. In either of these frameworks, distribution of work across nodes and devices needs to be handled by the application developer. In comparison, our framework handles work distribution and inter-node and/or inter-device communication automatically, but is restricted to specific patterns. OpenACC [1] is accelerated OpenMP for GPUs, based on a community standardization effort, with initial implementations released by Cray, CAPS, NVIDIA, and PGI. To the best of our knowledge, none of the existing OpenACC compilers support work distribution across CPU and GPU cores. One recent effort, however, does target multiple GPUs [31]. Some other efforts are underway that focus on each communication pattern separately. OCD [15] is a work-in-progress project for realizing the Berkeley Dwarfs in OpenCL, and conducting pattern specific optimizations for applications in each dwarf. In other efforts, StarSs [28] provides portability across multiple devices, but cannot currently handle work partitioning across nodes. Yet another recent effort generates code based on description of platforms [25], but cannot provide all functionality we provide.

Dividing work between CPU and GPU cores has also received attention in recent years, with prominent efforts being StarPU [2], OmpSs [3], XKaapi [16], and the work by Scogland *et al.* [26]. Among these systems, only Scogland *et al.* [26] provides task scheduling from high-level user code. In comparison, our work provides an even higher-level API, and performs processing for multiple nodes also. Qilin system [21] has been developed with an adaptable scheme for mapping the computation between CPU and GPU simultaneously. It uses curve-fitting to train the model.

Though our direct comparison (for distributed memory parallelization) has been against MPI, PGAS models have also become popular for scalable parallel programming. Models like Global Arrays [24] and UPC [8], to the best of our knowledge, do not support execution on GPU clusters, and cannot partition work across CPU and GPU cores. Languages that are based on more advanced compiler support, such as X10 [4] and Chapel [27], are more likely to be able to handle parallelism at multiple levels. For example, X10 has support for GPUs (CUDA), though not completely transparent to the programmer [9]. Overall, PGAS model implementation continues to be an area of very active research [27], [29], [14].

Recent years have seen an increasing number of efforts on domain specific languages, including many of them being implemented on GPUs and GPU clusters. Liszt [12] is a domain specific language for mesh-based PDE solvers, similar to irregular reductions in our system. Their framework is designed for CPU clusters, multi-core architectures, and GPUs, and thus is portable across platforms. However, it cannot accelerate computation using both CPU and GPU cores. Similarly, OP2 [23] is a library designed for accelerating unstructured mesh applications on either the multi-cores or many-cores, but not both together.

Physis [22] is a parallel programming model designed for stencil computations over GPU clusters. Though it runs on heterogeneous clusters, it only utilizes the GPUs for computation. Mint [30] is a source to source compiler for translating annotated 3D stencil C code into optimized CUDA code. It can support execution on a single GPU only. Zhang *et al.* proposed a framework for code generation and performance tuning for 3D stencil applications on GPU clusters [32]. Similar with Physis, although it was designed for distributed execution, only GPU cores, and not the CPU cores, are exploited. Com-

pared with these efforts, our contributions are in exploiting both CPU and GPU cores, and that our programming model can also enable an application that involves multiple distinct communication patterns.

## VI. Conclusions

We have focused on the problem of developing a high-level programming model for scientific applications which can exploit parallelism at multiple levels (across nodes in a cluster, across CPU cores, on GPUs, and across CPU and GPUs). This has been achieved by focusing on specific communication patterns that commonly arise in scientific applications, and developing high-level APIs specific to each. Our extensive runtime system involves pattern specific workload partitioning (at different levels) and optimizations for dealing with communication overheads and CPU and GPU specific features.

We have evaluated the scalability of our framework with five different applications, each involving one or more of the patterns. Our framework achieves high inter-node and inter-device parallelism. At the same time, it does not lose performance compared to handwritten codes that focus on a single level of parallelism (i.e. MPI codes or CUDA). On a 32 node GPU code, speedups between 600 and 1800 are obtained.

Our future work is to extend our framework to cover more communication patterns, and exploiting other architectures such as clusters involving Intel MIC coprocessors.

## References


[1] Openacc standard. http://www.openacc-standard.org.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[3] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Productive programming of gpu clusters with ompss. In *IPDPS*, 2012.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, 2005.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.

[6] Linchuan Chen and Gagan Agrawal. Optimizing MapReduce for GPUs with Effective Shared Memory Usage. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 199–210, New York, NY, USA, 2012. ACM.

[7] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating MapReduce on a Coupled CPU-GPU Architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 25:1–25:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[8] UPC Consortium et al. Upc language specifications. *Lawrence Berkeley National Lab Tech Report LBNL–59208*, 2005.

[9] Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat. GPU programming in a high level language: compiling X10 to CUDA. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, 2011.

[10] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1), January 1998.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, 2004.

[12] Zachary et al. DeVito. Liszt: a Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *SC*, 2011.

[13] Krste Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY, 2006.

[14] Montse Farreras, George Almási, Calin Cascaval, and Toni Cortes. Scalable rdma performance in pgas languages. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–12, 2009.

[15] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. OpenCL and the 13 dwarfs: a Work in Progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, 2012.

[16] Thierry Gautier, Fabien Le Mentec, Vincent Faucher, and Bruno Raffin. X-kaapi: A multi paradigm runtime for multicore architectures. In *ICPP*, 2013.

[17] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. Libwater: heterogeneous distributed computing made easy. In *ICS*, 2013.

[18] Xin Huo, Vignesh Ravi, Wenjing Ma, and Gagan Agrawal. An Execution Strategy and Optimized Runtime Support for Parallelizing Irregular Reductions on Modern GPUs. ICS '11, 2011.

[19] Yuan-Shin Hwang, Raja Das, Joel Saltz, Bernard Brooks, and Milan Hodo Scek. Parallelizing Molecular Dynamics Programs for Distributed Memory Machines: An Application of the Chaos Runtime Support Library, 1994.

[20] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: an OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *ICS*, 2012.

[21] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of MICRO*, 2009.

[22] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: An Implicitly Parallel Programming model for Stencil Computations on Large-scale GPU-accelerated Supercomputers. In *SC*, 2011.

[23] GR Mudalige, MB Giles, I. Reguly, C. Bertolli, and PHJ Kelly. OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures. In *Proceedings of Innovative Parallel Computing*, 2012.

[24] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A portable shared-memory programming model for distributed memory computers. In *Proceedings Supercomputing '94*, pages 340–349. IEEE Computer Society Press, November 1994.

[25] Martin Sandrieser, Siegfried Benkner, and Sabri Pllana. Using explicit platform descriptions to support programming of heterogeneous many-core systems. *Parallel Computing*, 38(1-2):52–65, 2012.

[26] Thomas Scogland, Barry Rountree, Wu chun Feng, and Bronis R. de Supinski. Heterogeneous task scheduling for accelerated openmp. In *IPDPS*, 2012.

[27] Albert Sidelnik, Saeed Maleki, Bradford L. Chamberlain, María Jesús Garzarán, and David A. Padua. Performance portability with the chapel language. In *IPDPS*, 2012.

[28] Vladimir Subotic, Steffen Brinkmann, Vladimir Marjanovic, Rosa M. Badia, José Gracia, Christoph Niethammer, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Programmability and portability for exascale: Top down programming methodology and tools with starss. *J. Comput. Science*, 4(6):450–456, 2013.

[29] Miwako Tsuji, Mitsuhisa Sato, Maxime R. Hugues, and Serge G. Petiton. Multiple-spmd programming environment based on pgas and workflow toward post-petascale computing. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 480–485, 2013.

[30] Didem Unat, Xing Cai, and Scott B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proceedings of the international conference on Supercomputing*, ICS '11, 2011.

[31] Rengan Xu, Sunita Chandrasekaran, and Barbara M. Chapman. Exploring programming multi-gpus using openmp and openacc-based hybrid model. In *IPDPS Workshops*, pages 1169–1176, 2013.

[32] Yongpeng Zhang and Frank Mueller. Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, 2012.