A scheduling and runtime framework for a cluster of heterogeneous machines with multiple accelerators

Tarun Beri, Sorav Bansal and Subodh Kumar
Indian Institute of Technology Delhi
New Delhi, India
{*tarun,sbansal,subodh*}*@cse.iitd.ac.in*

***Abstract*—We present a runtime system for simple and efficient programming of CPU+GPU clusters. The programmer focuses on core logic, while the system undertakes task allocation, load balancing, scheduling, data transfer, etc. Our programming model is based on a shared global address space, made efficient by transaction style bulk-synchronous semantics. This model broadly targets coarse-grained data-parallel computation particularly suited to multi-GPU heterogeneous clusters. We describe our computation and communication scheduling system and report its performance on a few prototype applications. For example, parallelization of matrix multiplication or 2D FFT using our system requires the regular CPU/GPU implementations and about 30 lines of additional C code to set up the runtime. Our runtime system achieves a performance of 5.10 TFlop/s while multiplying two square matrices of 1.56 billion elements each over a 10-node cluster with 20 GPUs. This performance is possible due to a number of critical optimizations working in concert. These include prefetching, pipelining, maximizing overlap between computation and communication, and scheduling efficiently across heterogeneous devices of vastly different capacities.**

I. Introduction

Modern computing environments consist of multiple nodes connected by a network. Each node may comprise multi-core processors and possibly many-core accelerators like graphics processing units (GPUs). Due to their attractive performance per-$ and per-watt, such accelerators have become mainstream in scientific computation and other domains. Nonetheless, writing efficient programs employing both CPUs and GPUs spread across a network remains challenging.

GPU programming is often based on bulk synchronous SPMD tasks delegated by CPU and run in a separate memory. Data marshalling to GPU memory for use by its kernels is an intricate exercise. Moreover, GPU clusters particularly complicate issues like synchronization, scalability, scheduling, load balancing and efficient data movement. Our system hides much of this complexity and presents a simple and unified interface to program and reason in this heterogeneous environment. It belongs to the genre of programming models like Map-reduce [1], that allow the programmer to focus on core logic and the runtime takes care of parallelism management, load distribution and communication.

Shared memory programming [2] is considered intuitive and familiar, but it also is inefficient if the memory is distributed across a network. That is why the partitioned global address space [3] model has gained popularity, in which applications directly manage the location of data. However, this gets especially cumbersome with GPUs, where a simpler programming style is even more valuable. This paper explores the feasibility of a unified shared memory style programming framework for GPU clusters and concludes that efficiency can indeed be achieved. In particular, we demonstrate that the traditional inefficiency of the shared memory approach can be offset by hiding communication latency behind coarse-grained computation and batching communication using ideas from transactional memory: the application operates on a local *views* of the global shared memory and inter-view conflict is resolved lazily.

To achieve this, we model CPUs and accelerators loosely as bulk synchronous computing units (BSP [4]) with logical phases of local computation and communication. This matches the performance case of GPUs. We map the appliation provided task-graph to a set of CPUs and GPUs, balancing load, abstracting and reducing data exchange, and hiding latency by overlapping computation and communication.

In our framework, applications provide tasks that uniformly see a global address space and can be thought of logically as BSP super-steps. Tasks are coarse-grained and may be further subdivided into any number of independent concurrent *subtasks*. Thus a subtask is a data-parallel work-sharing construct of a task, and is individually scheduled by our runtime on any available CPU or GPU in the cluster. Each subtask executes an application-provided kernel function. This kernel may is written in OpenCL [5] and executed on the assigned CPU or GPU. For generality, we also support multiple implementations of a subtask's Kernel, separately optimized for different device architecture in the cluster (e.g., C++ for CPUs and CUDA [6] for GPUs).

The data dependence of subtasks executing on CPUs is directly inferred from the kernel program and the required data is fetched (or pre-fetched) on demand. However, GPUs do not expose native on-demand paging. Hence, these data dependencies must be statically determined, This pre-specified dependency also allows us to aggressively pre-fetch data and hide network latency.

Drawing inspiration from transactional memory, each sub-

task is presented with a private logical view of the entire shared address space. All accesses then are local and writes are not immediately visible to other subtasks. Conflicting writes in different subtasks' views however are not aborted but rather resolved in a reduction (or synchronization) step at the end of the task. The reduced result is available to subsequent tasks. This implies that any operation on data produced by a different subtask of the task may only be performed in a subsequent task, when the writes are finalized.

Private views with deferred synchronization lead to sequential consistency trivially and there are no locks or race conditions among subtasks. This elimination of data hazards allows subtask kernel logic to be simple and local. Also, the concurrent semantics of subtasks enables our scheduler to pipeline them and significantly hide their data transfer latencies (by overlapping computation of one with data transfer of another). Concurrent subtasks can also be executed out of order and grouped in any order to increase the locality of reference. Our examples in section III and V demonstrate the simplicity of programs and exhibit 1600x+ speed-up over single CPU (image convolution) and 11x+ speed-up over single GPU implementation (LU decomposition). Note that the reference implementations do not scale to larger problems due to the limited memory on a single node.

Our framework can realize any task that may be optionally decomposed into a set of concurrently executable subtasks with checkout/checkin memory semantics and a synchronized reduction step to resolve conflicting checkins. The reduction is a generalization of the "reduce" step of Map-reduce and entire views are made available to the step. Tasks having non-deterministic access pattern (like graph traversal) or fine-grained/frequent communication or complex conflict resolution may not perform efficiently in our system. In this paper, we focus on coarse-grained experiments.

CPUs allow threads to be scheduled on a single core, but GPUs do not allow per-core scheduling and kernels occupy the entire GPU. This disparity poses scheduling challenges. Subtasks large enough to effectively use the GPU can be too slow on the CPU. Shorter ones may improve CPU performance, but GPUs remain under-utilized. User designing subtasks differently for different devices would compromise abstraction and simplicity of programming. In our framework, the application can logically partition tasks and remains unaware of where each subtask may be scheduled. Multiple subtasks can be scheduled on the same device. A single subtask can also be split to multiple devices. This supports dynamic adaptation of task size to the device on which it is scheduled.

In our tests, a BLAS [7] based sequential implementation of multiplication of two dense square matrices with 8K elements each (on Intel Xeon X5650) was outperformed by a CUBLAS [8] based implementation (on Tesla M2070) by a factor of 33x+. Similarly, a BLAS based LU-Decomposition on the GPU reported 10x+ speedup over the sequential implementation. This more than an order of magnitude disparity is further aggravated in a cluster environment if the faster GPUs are placed near the data while the slower CPUs are on a remote node away from the data. Dynamic load balancing is usually sufficient in CPU only environments, but with GPU friendly subtasks, even a single CPU subtask may slow the entire task down. To counter this performance disparity, our runtime performs multi-scheduling, allowing an idle GPU to start another instance of a subtask already assigned to a slow CPU. This may reduce utilization, however, and we evaluate its effectiveness in section V.

The primary contributions of this paper are:

1) We present the first shared-memory based programming framework that transparently maps and autonomously schedules an application's functions on any cluster of CPUs and GPUs in a load-balanced fashion.
2) We investigate the optimizations necessary for such a framework to be practical. Our runtime, (*Unicorn*) batches and combines data transfer for efficient communication. It performs prefetching and pipelining allowing compute-communication overlap. It also supports multi-scheduling in response to dynamically changing load.
3) Using benchmark applications implemented atop our runtime, we present a concrete study of scheduling and data transfer optimizations in a CPU/GPU cluster.

II. Related Work

Parallel programming frameworks have been an active area of research. The holy grail is to let the programmer focus on program logic and have the framework infer parallelism and execute parallel code efficiently on heterogeneous clusters. This remains challenging.

OpenMP [9] (shared memory) and MPI [10] (message passing) are two of the most widely used parallel programming frameworks. Shared memory models are often extended to multiple machines resulting in distributed shared memory (DSM) systems [2], [11], [12], [13]. DSM systems are easier to program than message passing ones, but generally employ complex memory consistency protocols (often requiring application specific knowledge) resulting in high coherence overheads. Message passing alternatives generally require transfer of not only data but also some control and program state information. With a large number of small data transfers, the latency quickly becomes a bottleneck. Our system uses the best of these two worlds by exposing a DSM style model to the programmer but behind the scenes employs shared memory (within a machine) and message passing (across machines) for data transfers. Deferred data exchanges in bulk amortize message passing overheads.

CUDA [6] is C/C++ like programming language for highly data-parallel processing on NVIDIA GPUs. These

GPUs implement hardware threads that can be created and scheduled quickly (a few cycles compared to a few thousand cycles in CPUs). CUDA programs often use thousands of threads to perform a data-parallel task. The memory hierarchy (private and shared) on the GPU is fully exposed to the programmer for program-specific data placement and optimization. The device memory is independent of the host memory and user must often explicitly copy the data from the host CPU. Once data is copied to the device, the GPU works independently and after finishing the computation, the user synchronizes the results back into the host memory. This bulk-synchronous nature of GPUs is a fundamental design parameter in our framework. Extending this bulk-synchronizability allows application programs to transparently generalize from one GPU to multiple and from one machine to a cluster of GPUs.

Many single node CPU-GPU programming frameworks like [14], [15], [16] have been proposed. But these lack scheduling and load balancing capabilities and division of work between CPU and GPU is largely left to the programmer. StarPU [17] is a broader single node framework that supports CPU/GPU co-scheduling, but for optimal results it often requires calibration runs and scheduling hints from the programmer. It employs fine-grained schedulable units called codelets. Manual sizing of codelets in this framework is a challenge as the same size generally does not suit CPUs and GPUs well. GPUSs [18], a derivative of StarSs, also requires programmers to annotate specific code blocks with constructs (that identify tasks and target devices) and directionality clauses (that determine data movement). These hints are used to build a task dependency graph, which determines the scheduling of individual tasks. XKaapi [19] is another system that employs a work stealing scheduler to distribute the load on CPUs and GPUs. In all these systems, scheduling decisions once made are not re-assessed and the granularity of work division between various accelerators and CPU cores is left to the programmer. These systems also do not target multiple machines.

Existing cluster programming systems can be broadly classified into two categories. First are language based approaches like [3], [20], [21], [22], [23], [24], [25], usually extending a sequential language like C or Fortran. The second category are library based approaches like [26], [27], [28], [29], [30]. The former ones focus variously on functional, loop or data parallelism and generally use shared address spaces (built on top of DSM or more specifically PGAS [31]) with fine grained synchronization. Their focus is to mainly allow the user to express parallelism at a high level and most do not support GPUs. In contrast, library based approaches employ some MPI-like communication where machine specific details are not completely abstracted from the programmer. Instead of focusing on program logic, the programmer has to directly handle issues like synchronization, scalability and latency. Hence, usual problems like race conditions and deadlocks remain.

Our framework *Unicorn* is also a library based approach. It is built on top of pthreads, MPI and CUDA. Its novelty is in the interface, which is general and intuitive, yet efficient. It unifies computation on local and remote computing units (CPUs and GPUs) using a bulk synchronous scheme. Its runtime environment automatically handles data distribution, scheduling and re-scheduling, load balancing, and synchronization. The closest existing work targeting GPU clusters are StarPU-MPI [32] and Phalanx [33]. The former is an extension of StarPU but it does not fully abstract the existence of multiple machines: the programmer must either explicitly manage communication with an MPI-like interface or explicitly submit independent tasks to each node of the cluster. Rather than a unified cluster programming framework, it is an MPI based aggregation of independent StarPU instances running on each node. It also lacks a holistic cluster wide scheduler. Rather, it uses multiple independent schedulers (one per node) which leaves load balancing mainly in the hands of the programmer. Phalanx is C++ template library with powerful mechanisms to create hierarchy of threads which are mapped onto a hierarchy of processors and memories. But it also lacks scheduling and load balancing capabilities. It also does not use CPUs and GPUs collectively for computations.

III. Programming Model

We detail our programming model with the help of an example. Figure 1 presents the pseudo-code skeleton (with execution logic removed) of a program to compute the one dimensional Fast Fourier Transform (1D-FFT) using our model. The model is implemented through a library of functions, some of which can register user-provided callbacks. These callbacks determine the division and distribution of input data, the computation logic and the reduction of the output. Our model supports allocation of PGAS like shared address spaces, where both the program input and the program output are stored. While an address space is logically shared, it is usually distributed across multiple machines and devices by our library. We have chosen to omit any 'flush' or global read/write primitive, and the nature of programming changes significantly because of that choice. We find this style of programming simpler, yet powerful enough for many coarse-grained scientific applications.

We propose a two level task hierarchy. The programmer decomposes the application into interdependent *tasks*. Address spaces are created at the beginning of a task, or passed from a task to the next. A task is itself a parallel entity: the programmer decomposes it into concurrent *subtasks*. A task can be spawned by another and any spawned task is ready for execution at the completion of all tasks it depends on. Subtasks of a ready task are distributed among all available devices, and a subtask is expected to run to completion. The task hierarchy is abstract and a program time decision. It is

```
struct complex { float real, imag; };
struct fft_conf { size_t rows, cols; };

fft_1d(matrix_rows, matrix_cols)
{
 key = "FFT";
 register_callback(key, SUBSCRIPTION, fft_subscription);
 register_callback(key, OPENCL, "fft_ocl", "prog.ocl");

 if(get_host() == 0)  // Submit task from single host
 {
  // create address spaces
  size = matrix_rows * matrix_cols * sizeof(complex);
  input = malloc_shared(size);
  output = malloc_shared(size);

  initialize_input(input); // application code

  // create task with one subtask per row
  subtasks = matrix_rows;
  task = create_task(key, subtasks, fft_conf(matrix_rows,
       matrix_cols));

  bind_address_space(task, input, READ_ONLY);
  bind_address_space(task, output, WRITE_ONLY);

  submit_task(task);
  wait_for_task_completion(task);
 }
}
```

```
fft_subscription(task, device, subtask)
{
 fft_conf* conf = (fft_conf*)(task.conf);
 row_size = conf->cols * sizeof(complex);

 // subscription offset and length (one row per subtask)
 subscription_info sinfo(subtask.id * row_size, row_size);

 // subscribe to input and output memory by index
 subscribe(task.id, device.id, subtask.id, 0, sinfo);
 subscribe(task.id, device.id, subtask.id, 1, sinfo);

 // OpenCL kernel launch configuration
 set_launch_conf(task.id, device.id, subtask.id, ...);
}

// OpenCL Kernel (should go into prog.ocl file)
fft_ocl(task, device, subtask)
{
 fft_conf* conf = (fft_conf*)(task.conf);
 complex* input = (complex*)subtask.subscription[0];
 complex* output = (complex*)subtask.subscription[1];

 ... OpenCL 1D FFT Code ...
}
```

Figure 1: Unicorn program and callbacks for 1D FFT

not tied to a cluster topology and is dynamically mapped to and executed on any cluster by our runtime. There is an implied barrier at the end of subtasks that indicates reduction of conflicting memory writes and completion of the task. The 1D FFT task in figure 1 creates one subtask per matrix row.

Our model degenerates to the traditional task graph model if each task comprises a single subtask and reduces to a sequential entity. However, our focus is on applications that can take advantage of large tasks that may itself be decomposed into parallel work units.

A subtask is specified by three callback functions:

1) A *subscription* function that may explicitly specify the regions of one or more address spaces the subtask accesses. Subtask subscriptions may overlap and a subtask may subscribe to an entire shared space.
2) A *kernel execution* function that specifies the execution logic of subtasks, in SPMD fashion.
3) A *data synchronization* function that manages the output of subtasks (i.e., reduction or redistribution).

Figure 2 shows these three stages of the subtask and the pseudo-code of the *Subscription* and *Execution* callbacks (for 1D FFT task) is listed in figure 1. The subtasks of this task do not have conflicting writes, hence an explicit *Data Synchronization* callback is not required.

Please note that explicitly specifying subtask subscriptions is not a requirement of our model. However, due to a lack of virtual paging on modern GPUs one cannot automatically deduce this information at runtime without instrumenting the user code. Static code analysis is also not a viable option as its scope is limited and it imposes restrictions on how kernels should be written. On CPUs, however, automatic subscription inference is possible by using POSIX's [34] *mprotect* feature to protect shared address spaces and upon access by a subtask, that virtual page along with few nearby pages are prefetched (if necessary). With a pre-fetch of 5 pages, we measured this scheme to be only 10% slower than explicit subscriptions while multiplying two dense square matrices with 4K elements each on an 8-node cluster with 12 CPU cores per node.

Our model has a notion of subscription views. A subtask may subscribe to multiple shared address spaces and multiple discontiguous regions within each address space. In the subtask kernel the user may use the original global address space ("natural view") or a packed remapping of addresses so the subscribed regions appear contiguous ("compact view"). The compact view can save space and also leads to better memory hierarchy usage on GPUs, where memory is scarce (and virtual paging is not available). For both views, a private copy of the subscribed regions of shared address space is created for each potential writer, where it accumulates writes locally. On completion of kernel execution conflicting spaces are combined. Read-only regions are also fetched and cached locally on nodes but are shared among subscribing subtasks scheduled on that node.

Subtask kernels are normally written in OpenCL. However, for best performance one may write specialized kernels for each architecture type in the cluster (e.g. C/C++ for CPUs and CUDA for NVIDIA GPUs). When writing specialized kernels, the programmer need not reason about the memory hierarchy of various devices beyond the point it is normally done while writing sequential CPU or CUDA kernel implementations. This means that most existing kernel implementations can be used as is and can be debugged in sequential or single GPU setup before being used.

We allow two special operators for synchronizing the output computed by different subtasks: `reduce` and `redistribute`. The `reduce` operator requires a reduce callback function from the user, that must be commutative and associative, i.e., the order of reduction should not matter. This allows programs to be written in the popular map-reduce style. The reduce callback reduces the outputs of two subtasks into one. Our runtime system schedules these

callbacks to reduce all subtasks in a hierarchical manner (Figure 3), greedily performing reductions as subtasks complete. The runtime first reduces all subtasks local to a node and all nodes do this in parallel. Once a node is done with all local reductions, it either sends its reduced data to another node or receives it from another node. (i.e., the nodes form logical pairs and do inter-node subtask reductions in parallel). Reductions continue in a binary-tree fashion across nodes. This improves parallelism and reduces data transfer.

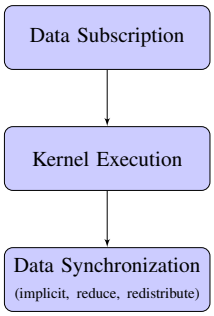


Figure 2: Execution Stages of a Subtask

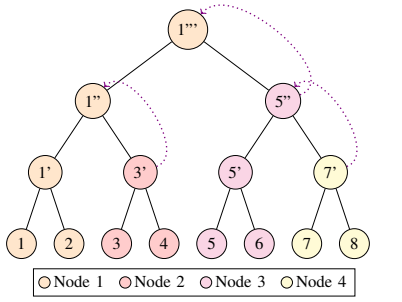


Figure 3: Hierarchical Reduction - leaf nodes are subtasks, others are reduced subtasks; dotted lines are inter-node data transfer

The `redistribute` operator is useful in situations where computation proceeds in a series of steps, where each step is specified as a task. Often, in such programs, data needs to be reordered for the subtasks in the following task based dynamically on the computation of the previous step. With the redistribute operator, the programmer can associate a *rank* with different regions of an output address space. Our runtime then ensures that all memory regions with the same rank, are placed consecutively in the address space, before the next task begins. Thus the regions can be ordered in shared address space by rank and within a rank by the writing subtask id. This operator can also be used to demand all-to-all broadcast or scatter-gather, more efficiently than through individual subscription.

IV. Runtime

We next discuss our runtime system, *Unicorn*, and its implementation for a cluster of nodes across a network, each with multiple CPUs and GPUs.

A. Unifying Threads, MPI, and CUDA

At initialization, our runtime system starts independent MPI processes, one per node in the computational cluster. Each process has two major sub-systems - one for "network" and other for "scheduling". Both are managed by a dedicated thread and take care of cluster wide operations like subtask scheduling and data exchange between nodes. The scheduling sub-system is a two-level hierarchy of threads with "scheduler thread" employing a "compute thread" for every computing unit (e.g., CPU core, GPU device) on the node. All threads serve a private priority queue that contains the commands enqueued for it. These priority queues are revocable and commands once enqueued can be removed before execution. The commands are carefully designed to minimize the load on these queues. For example, one of our commands allows execution of a contiguous set of subtasks (defined by a starting and ending subtask number) in one go. Our scheduler chooses the computing unit and the set of subtasks that should run on it. If the computing unit is remote, the scheduler requests the network sub-system to deliver the command to the concerned queue.

For inter-node communications, the commands passed to the network sub-system undergo a series of optimizations minimizing the number of MPI requests and the volume of data transferred. These include buffering and grouping disjoint requests into larger chunks when possible, filtering out duplicate requests made by computing units, and combining multiple outgoing messages to the same node into one. The compute threads may also issue low priority prefetch requests to the network sub-system (for subtasks expected to run in the future).

All network communications are asynchronous, which compute threads overlap with any pending subtask execution or synchronization. To further minimize the idle time, the compute threads share read-only address spaces among CPU cores and use simultaneous DMA transfers to and from GPUs. In addition to on-CPU caches of memory fetched from remote nodes, our runtime system maintains an on-GPU software LRU cache for portions of the address spaces currently loaded on the GPU device. This greatly helps eliminate unnecessary data transfers when the same read-only data is requested by multiple subtasks scheduled on that GPU, or in cases where data written by a subtask is later read by another subtask of a subsequent task.

The observed overhead of these Unicorn threads is small. For example, the image convolution experiment (section V) with 336 subtasks has a measured overhead of 0.2% of the total execution time.

B. Shared Address Spaces

Our runtime implements distributed address spaces logically shared among all cluster nodes. These are specifically designed for efficient transactional semantics and minimize explicit cache coherence messages (for this reason, we do not employ the usual MSI coherence protocol). These semantics mean that a subtask sees the data from the beginning of the task, and then only its own updates to that data. After the task finishes, the writes of the subtasks become visible to subsequent tasks.

For every address space, each node maintains an initial directory mapping address space regions to its master unmutable copy in the cluster (and all subtasks executing on this node refer to it). As data migrates or replicates, requests get fulfilled and new information is received, a secondary directory maintains updates to the original directory. Further requests for the data are served locally using this secondary directory. If a subtask writes to a copy of the address space,

the secondary directory does not include that location (so that the other subtasks do not see its writes). At the end of the task, all writes are assimilated into the initial directory and the secondary directory is discarded. In case there are no writes to the address space, the secondary directory is retained for potential reuse by subsequent tasks. This design ensures that no explicit coherence messages are required during the task execution. However, at the end of the task, a few coherence messages may be exchanged in case the location of the master copy of any region changes.

C. Scheduling and Avoiding Stragglers

As stated in section I, the GPUs outperform CPU cores by more than an order of magnitude for several compute intensive experiments. Scheduling subtasks across devices of widely different capabilities presents its own challenges. StarPU addresses this performance disparity by grouping multiple CPU cores together into a single device. However, this compromises the simplicity of the programming model as CPU kernels now have to be multi-threaded. This also takes away the liberty to plug in existing sequential CPU kernels in the system. Another alternative is to envision a subtask as a set of work-items and schedule a single work-item per CPU core. This is analogous to OpenCL's work-group and work-item. This feature is implemented in our system but a task may opt not to use it in the interest of simplicity. In that case all CPU cores and GPUs execute the subtasks of the same size. We use a two level dynamic scheduling scheme to automatically size subtasks:

1) A coarse-grained cluster-level scheduler to schedule subtasks among all devices in the cluster.
2) A fine-grained device-level scheduler to create work-items from subtasks assigned to slow CPU cores.

The benefit of dynamic scheduling is that it responds not only to varying subtask load and compute unit capacity but also to varying network throughput. Our coarse-grained scheduler is based on work stealing [35], [36]. It assigns an equal number of subtasks to each computing unit initially. However, after a computing unit finishes, it *steals* subtasks from one of the other computing units. Specifically, we use a two-level random steal where a computing unit first randomly chooses a node and then the node randomly chooses one of its computing unit as the victim. Subtasks are stolen from the tail of the queue to try to preserve the locality of reference. Each time a computing unit completes its queue, it attempts to steal work from others, and this process terminates only when all work queues are empty. We also experimented with one level random stealing where a device directly steals from another device, but found that scheme to be sub-optimal as it generates too many steal requests and is not as scalable.

Once the coarse-grained scheduler has scheduled a subtask onto a slow device (typically a CPU core), fine-grained scheduler creates multiple work-items from the subtask. This scheduler employs binary search to determine the optimal number of work-items required to execute a subtask. The maximum number of such work-items is equal to the number of CPU cores on a node. For the first subtask, the scheduler creates a single work-group comprising of all CPU cores on the node. For the next subtask, the size of work-group is reduced by half. Depending upon the relative performance of the two, the next work-group created has one-fourth or three-fourth the total number of CPU cores. This dynamic adjustment continues until the end of the task.

It is possible for a subtask to take a long time to finish, while other computing units have become idle. To avoid this, the scheduler may *multi-assign*, i.e., assign the same subtask to multiple computing units, and commit results from whichever finishes earlier. We do not actually migrate away the subtask from the slow device, allowing the multi-assigned units to compete. When the faster one finishes, it subsequently aborts the unused subtask or its result is discarded (if the abort command did not reach in time, and the subtask also completed on the second unit).

Multi-assign is implemented as a generalization of stealing which means that stealing is not limited to pending subtasks but subtasks may be stolen for concurrent execution as well. The steal request includes a hint for multi-assignment. Generally, if the stealer has a low steal success ratio (for the last few steal requests sent by it) or is an aggressive device like GPUs, it requests multi-assignment.

Please note that stealing and multi-assignment only happen towards the end of a task when only few subtasks are left (and most have been executed). At this time, the devices already know their relative power and it is taken into account while deciding how many subtasks should be stolen or to determine if a subtask is straggling and the stealer has more probability to bring it to a faster completion. Further, a subtask is either multi-asssigned to a different node (from the one to which it was originally assigned) or to a different device type on the same node (CPU subtasks are multi-assigned to GPU and vice versa). This reduces the possibility of slow execution even on the re-assigned computing unit. We have found that this capability of assigning the same subtask to multiple units simultaneously can significantly reduce task completion times, especially in networked environments, where the effective data transfer bandwidths may be highly variable and a computing unit on a node with slow link may not receive its input fast enough, delaying the completion of the entire task.

D. Prefetching and Pipelining

An important characteristic of our runtime system is its ability to overlap communication with computation. In particular, if a set of contiguous subtasks is assigned to a computing unit, we start execution of the first subtask in the set, while simultaneously transferring the data for the second. Similarly, we overlap the execution of the second

subtask with the communication of the third, and so on.

This mechanism is especially useful for GPUs capable of compute-communication overlap and multiple kernel launches, where we create a pipeline of subtasks. At any given time, one subtask may be transferring its data to the GPU, one or more subtasks may be executing and one subtask may be copying its data out of the GPU. In our experiments, these techniques have resulted in significant performance improvements — more than 100% gain is observed in image convolution experiment (section V).

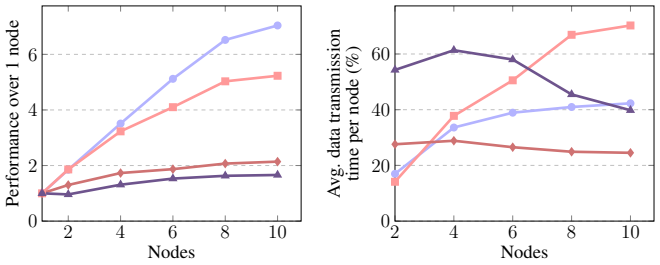
V. Experiments

We have implemented a few coarse-grained core scientific computation benchmarks over Unicorn. These include image convolution, matrix multiplication, LU matrix decomposition, and two-dimensional fast fourier transform (2D-FFT). We have chosen these benchmarks as they have well known parallelizations and each involves a different kind of complexity. Image convolution though embarrassingly parallel has a unique requirement of fringe around the image, matrix multiplication is computationally intensive with heavy data transfer requirements, LU decomposition has a nested task hierarchy and finally 2D-FFT involves a parallelization-unfriendly matrix transpose operation. The goal of these experiments is to assess the class of applications to which our model responds well.

Our experiments were performed on a cluster of ten nodes, each equipped with two 6-core Intel Xeon X5650 2.67 GHz processors and two Fermi generation CUDA cards (Tesla M2070). The machines run CentOS 6.2 with CUDA 5.5. For communication, we use Open MPI [37] 1.4.5 (over SSH) over an Infiniband [38] network with 32Gbps theoretical bandwidth. Unless stated otherwise, the input address spaces are randomly distributed (as $2048 * 2048$ blocks) over the cluster nodes and our runtime transparently moves data on-demand to other nodes, as the program executes. All measurements are based on at least three trials.

We first discuss the implementation of these benchmarks over Unicorn and present how these implementations scale with an increasing number of nodes. Then we compare the performance of these benchmarks to the reference cases when only CPU cores and when only GPUs are used in the cluster, respectively. These results highlight the performance disparity between CPUs and GPUs and also show when and to what extent our scheduler bridges that gap. Finally, we evaluate our runtime's effectiveness for scheduling, multi-assign, load balancing, prefetching and pipelining. We also study the runtime's response to change in subtask size and initial data placement strategy.

In our image convolution experiment all color channels of a 24-bit RGB image of size $43008 * 32768$ are convolved with a $31 * 31$ filter. The input image is stored in a read-only address space (initially distributed randomly across the cluster nodes), logically divided into 336 blocks of size $2048 * 2048$. Each block is convolved using a separate subtask. However, because convolution at boundaries requires data from adjoining blocks, the input memory subscription of a subtask overlaps with other subtasks', potentially at all four boundaries. The output image is generated in a write-only address space. Results in figure 4 show the implementation's scaling. The drop towards the end is because the increase in the number of nodes reduces the number of subtasks available per node which in turn reduces the gains that were otherwise attained through subtask pipelining and compute/communication overlap.



Nodes	Data transferred (GB) 2	4	6	8	10	Data transfer events 2	4	6	8	10
Image Convolution	1.98	2.98	2.77	3.22	3.40	255	507	643	595	685
Matrix Multiplication	6.13	15.75	25.47	33.69	43.81	67	244	362	476	590
LU Decomposition	15.41	23.25	30.83	35.88	39.92	986	1488	1975	2306	2562
2D FFT	8.00	11.00	13.92	14.69	15.86	528	788	883	981	1038

Figure 4: Performance analysis of various benchmarks

In the matrix multiplication experiment two dense square matrices of size $2^{15} * 2^{15}$ each are multiplied to produce the result matrix. Each input matrix is stored in a read-only address space and the result matrix is stored in a write-only address space of the task. The output matrix is logically divided into $2048 * 2048$ sized blocks and computation of each block is assigned to a different subtask (which subscribes to all blocks in the corresponding row of the first input matrix and all blocks in the corresponding column of the second input matrix). The CPU subtask callback is implemented using a single-precision BLAS [7] function and the GPU callback uses the corresponding CUBLAS [8] function. Results in figure 4 show how the implementation scales. Results also show that the experiment is network intensive with nodes spending about 70% time in data transmission (in 10 node case). Although the input and output matrices are 4 GB each, every input block is required by all subtasks in its row and all in its column. Thus about 44 GB data is eventually transferred in 590 pipelined events.

Using GPUs only, when multiplying of two square matrices of size $40960 * 40960$, XKaapi reports a peak performance of 2.43 TFlop/s (on one node with 8 Tesla C2050 GPUs) whereas *Unicorn* achieves a peak performance of 2.65 TFlop/s on a comparable cluster (four nodes with 2 Tesla M2070 GPUs each) with subtasks of size $4096 * 4096$.

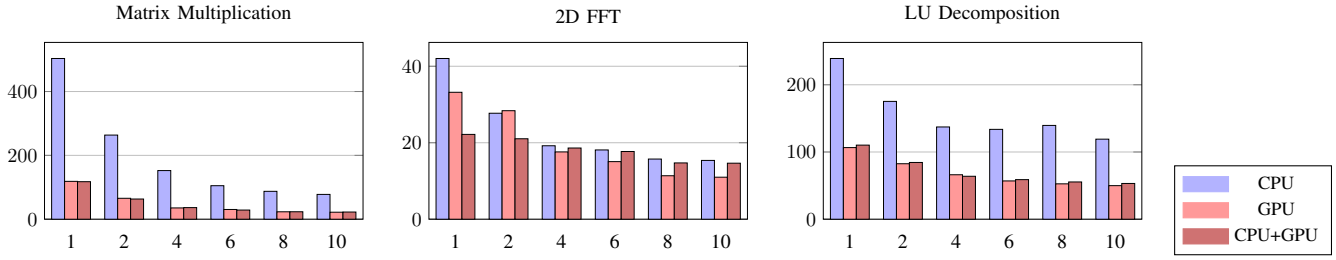


Figure 5: Execution time (sec) versus nodes for various experiments over a matrix of 1 billion elements

The same computation when run over 20 GPUs of our 10-node cluster delivers a performance of 5.1 TFlop/s whereas CUPLAPACK [39], a specialized linear algebra package, reports 6.2 TFlop/s over 32 Quadro FX 5800 GPUs of a 16 node cluster connected via QDR Infiband network.

We next discuss the in-place block LU Decomposition [40] experiment. The input matrix ($2^{15} * 2^{15}$) is kept in a read-write address space and is logically divided into $2048 * 2048$ sized blocks. The matrix is solved top-down for each diagonal block. For a matrix divided into $n * n$ blocks, solving for each diagonal block (i, j) involves three tasks – LU decomposition of the diagonal block (i, j), propagation of its results to other blocks in its row $(i, j + 1...n)$ and column $(i + 1...n, j)$ and propagation of these results to other blocks underneath $(i + 1...n, j + 1...n)$. The first of these three tasks is executed sequentially while the other two are executed in parallel. One task is spawned per diagonal block which, in turn, executes 3 tasks within, making a total of $3n$ tasks (where n is the number of diagonal blocks). The parallelism in tasks (i.e. the number of subtasks) reduces as we move down the matrix because the number of blocks to be solved in parallel decreases. The CPU subtask implementation uses single-precision BLAS functions while the GPU implementation employs the corresponding CUBLAS routines. Figure 4 shows the results.

The 2D-FFT experiment performs two single-precision one dimensional complex-to-complex FFTs (one along matrix rows and the other along matrix columns) over a matrix with $2^{15} * 2^{15}$ elements. The input matrix is initially randomly distributed over the cluster nodes in blocks of 1024 consecutive matrix rows. We use two *Unicorn* tasks for the experiment (each with 32 subtasks). The first task performs 1D-FFT along matrix rows while the second performs 1D-FFT along matrix columns. Note that we do not need to perform an explicit transpose in between the two and instead rely on our memory management subsystem to efficiently serve data. For the first 1D-FFT, the subtask size is 1024 rows while it is 1024 columns for the second. The CPU subtask callback uses calls to the FFTW [41] library, whereas the GPU subtask uses calls to the CUFFT [42] library functions. Figure 4 shows the results.

The results in figure 4 indicate a similar amount of data transfer for matrix multiplication and LU decomposition. However, the former is implemented as a single task with time complexity $O(n^3)$ and the latter has many iterations with three tasks using BLAS calls of $O(n)$, $O(n^2)$ and $O(n^3)$ time complexities. This increases communication latency, which is evident from five times more data transfer events (2562 versus 590). This coupled with the fact that the first of these three tasks is sequential, leads to lower scalability for the experiment. In contrast, both image convolution and 2D FFT have lower time complexities – $O(nm)$ for the former (m being the filter size) and $O(n \log n)$ for the latter but the latter has around 5 times more data transfer, resulting in its lower scalability. These results collectively indicate that experiments with high compute to communication ratio are expected to perform better in our system.

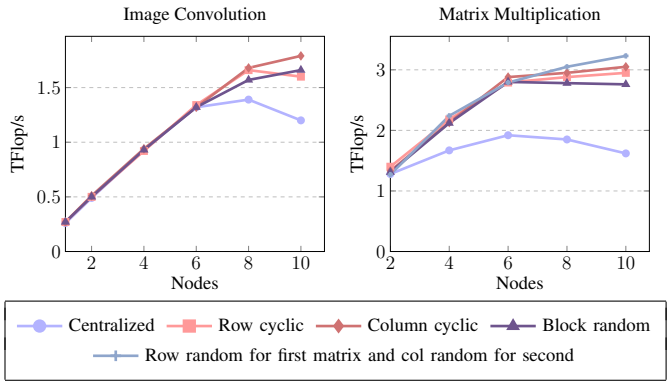


Figure 6: Analysis of various data distribution strategies

We also study the effectiveness of pipelining in hiding remote data access latency. If all required data were locally available on each of the 10 nodes, image convolution gets only 1% faster and matrix multiplication gets 31% faster. On the other hand disabling pipelining makes them 2.29x and 1.19x slower, respectively.

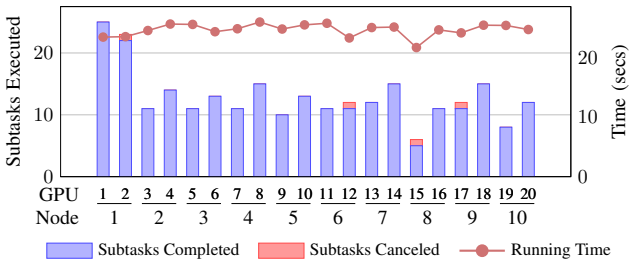


Figure 7: Matrix Multiplication Load Balancing

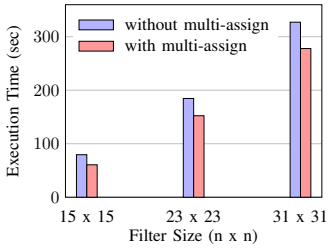


Figure 8: Multi-assign (Image Convolution)

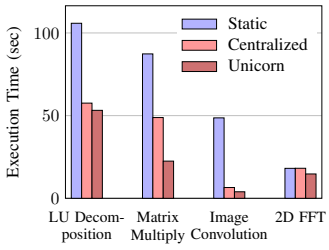


Figure 9: Scheduling models

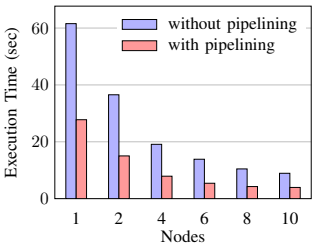


Figure 10: Pipelining (Image Convolution)

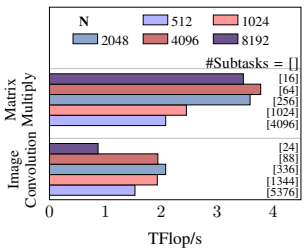


Figure 11: Subtask size ($N * N$)

Figure 5 plots CPUs-only and GPUs-only performances. Results show a wide disparity between CPUs-only and GPUs-only cases for matrix multiplication and LU decomposition. Due to this, employing both together does not provide any additional throughput as all CPU subtasks get multi-assigned to GPUs. Also, running some subtasks on slower CPUs takes away the opportunity to pipeline those in case the GPUs had executed them. However, CPU+GPU performance is close to GPUs-only case. This shows that our scheduler recognizes this and successfully pulls all subtasks to GPUs. The CPU-GPU disparity is narrowest for 2D-FFT, where the best results occur when using both together. That experiment uses only 32 subtasks, however, restricting parallelism to 32 devices only.

We now analyze effectiveness of our runtime system by putting it under stress tests. We perform first test by varying the initial placement of data in address spaces. Figure 6 evaluates image convolution and matrix multiplication for various schemes like *centralized* (entire address space on one cluster node), *row cyclic* (rows of $2048 * 2048$ blocks placed in sequence on all cluster nodes), *column cyclic* (columns of $2048 * 2048$ blocks placed in sequence on all cluster nodes) and *block random* ($2048 * 2048$ blocks placed randomly on any cluster node). Additionally, a fifth scheme is plotted for matrix multiplication where the rows of $2048 * 2048$ blocks for first input matrix and columns of $2048 * 2048$ blocks for second input matrix are placed randomly in the cluster. Results show that our runtime maintains performance despite the changes in data availability pattern. Only the *centralized* scheme behaves poorly as the network interface of the node containing entire data chokes.

In another experiment (matrix multiplication using GPUs only), we study effectiveness of our load balancer in *centralized* scheme (figure 7). Since entire data is located on the first node, our runtime schedules more subtasks on the two GPUs of this node as compared to others. The fact that all 20 GPUs in the cluster finished at almost the same time, means that load was optimally distributed among the devices.

In another stress test performed over four nodes with image convolution benchmark, we artificially overload one of the nodes with a process per core computing trigonometric functions indefinitely. In this case, we expect subtasks assigned to the overloaded node to be moved away from it. Our scheduler does this through stealing and multi-assignment. In the absence of multi-assignment a subtask may start running on a slow device and take a long time to finish, thereby, delaying the entire task. We run this test twice once allowing multi-assignment and once preventing it. Results in figure 8 show that, with multi-assignment, subtasks of the overloaded cores get re-assigned and the task completes faster. Without it, the task remains bottlenecked by the 'slow' cores. Our heuristics generally only multi-assign fewer than 1% of the subtasks, but the later-assigned unit finishes first about 50% of the time. Of course, when one finishes, the other is aborted, leading to a faster overall time. The cancellation protocol itself has non-significant overhead.

Now we compare our distributed and dynamic two-level scheduler with a static scheduler and a dynamic but centralized scheduler. The static scheduler equally divides all subtasks between CPU cores and GPUs in the cluster and does not change the assignments once made. The dynamic centralized scheduler starts with a static allocation of a few subtasks and then incrementally calibrates the allocation based on the observed execution times of each computing unit. Initially, a fixed number of subtasks (say one) are assigned to each unit. Depending on the completion time of the assigned work, the scheduler chooses the number of subtasks to assign to that unit the time around. If a unit finishes the job faster than others, its subtask assignment count is doubled. On the other hand, if a unit completes its subtasks slower than other units, the count is halved in its next allocation. Results in figure 9 show that the *static* scheduler performs poorly for all benchmarks. The *centralized* scheduler performs better than the *static* one but is inferior to *Unicorn's* distributed scheduling.

Figure 10 shows the impact of pipelining using the image convolution experiment. With pipelining, our runtime overlaps computation of one subtask with the communication of the next. Further, on GPUs this makes multiple simultaneous kernel executions possible. Results show that our runtime achieves 2x+ speed-up with pipelining.

Lastly, figure 11 shows the response of our runtime to change in subtask size. Within a reasonable range - (2048-8192) for matrix multiplication and (1024-4096) for image convolution - our system is able to maintain the throughput close to peak performance. For extreme sizes, however, the throughput degrades as on one extreme there are too few subtasks to generate enough parallelism and on the

other there are too many subtasks resulting in data transfers dominating the exploitable parallelism.

The experiments demonstrate that a system like ours can produce significant speedups for many useful applications by effectively utilizing all computational devices available in a cluster. Our primary contribution is a simple and practical programming model for modern set of computing devices, and an efficient runtime implementation that supports it.

VI. Conclusions and Future Work

We present a bulk synchronous programming model that allows distribution of computation across multiple CPU cores within a host, multiple GPUs, and multiple hosts connected over a network, in a unified manner. Our model maps efficiently to modern devices like GPUs, as they are already bulk synchronous in nature. Our runtime undertakes all local and networked data transfers, scheduling, and synchronization in an efficient and robust manner. By design, we eliminate races and deadlocks as all devices operate in a private view of the address space.

Bulk synchronous semantics defer memory commits to task boundaries. This can force some applications to have very short subtask kernels, resulting in an inefficient design. The subtask barrier can be made optional. However, the simplicity of semantics will suffer. Moreover, subtask graph will also have to be used to make scheduling decisions in addition to the task graph. A detailed analysis of network throughput will also help incorporate subtask affinity to nodes where more of their input reside. These improvements and experience with a wider set of applications and a larger cluster can promote better understanding of this model.

References

[1] Dean and Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, Jan. 2008.

[2] Min *et al.*, "Supporting realistic OpenMP applications on a commodity cluster of workstations," in *Intl. Conf. on OpenMP Shared Memory Parallel Programming*, ser. WOMPAT, 2003.

[3] El-Ghazawi and Smith, "UPC: Unified parallel C," in *ACM/IEEE Conf. Supercomputing*, ser. SC, 2006.

[4] Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, 1990.

[5] Stone *et al.*, "OpenCL: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, 2010.

[6] Nickolls *et al.*, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, Mar. 2008.

[7] Dongarra *et al.*, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, no. 1, Mar. 1988.

[8] "The NVIDIA CUDA basic linear algebra subroutines," https://developer.nvidia.com/cuBLAS.

[9] Dagum and Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, Jan. 1998.

[10] Gropp *et al.*, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, Sep. 1996.

[11] Amza *et al.*, "Treadmarks: Shared memory comput. on networks of workstations," *Computer*, vol. 29, no. 2, Feb. 1996.

[12] Nieplocha *et al.*, "Advances, applications and perf. of the global arrays shared memory programming toolkit," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, May 2006.

[13] Parzyszek, "Generalized portable shmem library for high perf. computing," Ph.D. dissertation, 2003.

[14] Wienke *et al.*, "OpenACC: First experiences with real-world applications," in *ICPP*, ser. Euro-Par, 2012.

[15] Lee and Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *SC*, 2010.

[16] "C++ AMP: Language and programming model," http://msdn.microsoft.com/en-us/library/hh265136.aspx, 2012.

[17] Augonnet *et al.*, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.

[18] Ayguadé *et al.*, "An extension of the StarSs programming model for platforms with multiple GPUs," in *Euro-Par '09*.

[19] Gautier *et al.*, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IPDPS '13*.

[20] Krishnamurthy *et al.*, "Parallel programming in Split-C," in *ACM/IEEE Conf. Supercomputing*, ser. SC, 1993.

[21] Blumofe *et al.*, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, Aug. 1995.

[22] Numrich and Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, Aug. 1998.

[23] Nguyen and Kuonen, "Programming the grid with POP-C++," *Future Gener. Comput. Syst.*, vol. 23, no. 1, Jan. 2007.

[24] Hilfinger *et al.*, "Titanium language reference manual," University of California at Berkeley, Tech. Rep., 2001.

[25] Fatahalian *et al.*, "Sequoia: Programming the memory hierarchy," in *ACM/IEEE Conf. Supercomputing*, ser. SC, 2006.

[26] Beguelin *et al.*, "A user's guide to PVM parallel virtual machine," University of Tennessee, Tech. Rep., 1991.

[27] Foster, "Globus toolkit version 4: Software for service-oriented systems," in *IFIP Intl. Conf. Network and Parallel Computing*, ser. NPC, 2005.

[28] Aji *et al.*, "MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems," in *HPCC'12*.

[29] Wang *et al.*, "MVAPICH2-GPU: Optimized GPU to GPU commun. for infiniband clusters," *Comput. Sci.*, vol. 26, no. 3-4, Jun. 2011.

[30] Hill *et al.*, "BSPlib: The BSP programming library," *Parallel Comput.*, vol. 24, no. 14, 1998.

[31] Stitt, "An introduction to the partitioned global address space (pgas) programming model," in *Connexions*, Mar 2010.

[32] Augonnet *et al.*, "StarPU-MPI: Task programming over clusters of machines enhanced with accelerators," in *Euro. Conf. Recent Advances in the MPI*, ser. EuroMPI, 2012.

[33] Garland *et al.*, "Designing a unified programming model for heterogeneous machines," in *Intl. Conf. on High Perf. Computing, Networking, Storage and Analysis*, ser. SC, 2012.

[34] Gallmeister, *POSIX.4: Programming for the Real World*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1995.

[35] Lifflander *et al.*, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Intl. Symp. High-Perf. Parll. and Dist. Comput.*, ser. HPDC, 2012.

[36] Dinan *et al.*, "Scalable work stealing," in *SC*, 2009.

[37] Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Euro. PVM/MPI Users Group Meeting*, 2004.

[38] InfiniBand Trade Association, InfiniBand Architecture Specification, Release 1.1, 2002.

[39] Fogue *et al.*, "Retargeting plapack to clusters with hardware accelerators," in *HPCS*, June 2010.

[40] Demmel *et al.*, "Block LU factorization," http://www.netlib.org/utk/papers/factor/node7.html, 1995.

[41] Frigo and Johnson, "Fftw: an adaptive software architecture for the FFT," in *ICASSP*, 1998.

[42] "CUFFT: CUDA fast fourier transform (FFT)," http://docs.nvidia.com/cuda/cufft/.