Slices: Provisioning Heterogeneous HPC Systems

Alexander Merritt, Naila Farooqui,
Magdalena Slawinska,
Ada Gavrilovska, Karsten Schwan
College of Computing
Georgia Institute of Technology, Atlanta, GA
{merritt.alex,magg}@gatech.edu
{naila,ada,schwan}@cc.gatech.edu

Vishakha Gupta
Intel Labs
Hillsboro, OR
vishakha.s.gupta@intel.com

ABSTRACT

High-end computing systems are becoming increasingly heterogeneous, with nodes comprised of multiple CPUs and accelerators, like GPGPUs, and with potential additional heterogeneity in memory configurations and network connectivities. Further, as we move to exascale systems, the view of their future use is one in which simulations co-run with online analytics or visualization methods, or where a high fidelity simulation may co-run with lower order methods and/or with programs performing uncertainty quantification. To explore and understand the challenges when multiple applications are mapped to heterogeneous machine resources, our research has developed methods that make it easy to construct 'virtual hardware platforms' comprised of sets of CPUs and GPGPUs custom-configured for applications when and as required. Specifically, the 'slicing' runtime presented in this paper manages for each application a set of resources, and at any one time, multiple such slices operate on shared underlying hardware. This paper describes the slicing abstraction and its ability to configure cluster hardware resources. It experiments with application scale-out, focusing on their computationally intensive GPGPU-based computations, and it evaluates cluster-level resource sharing across multiple slices on the Keeneland machine, an XSEDE resource.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Distributed architectures*

Keywords

GPGPU virtualization, resource slice, vgpu, assembly

1. INTRODUCTION

High-end machines are becoming increasingly heterogeneous, incorporating combinations of multi-socket many-core processors, non-uniform memory, and accelerators like GPGPUs, all to satisfy increasing demands on computing and memory resources. Software support for GPGPUs, e.g., OpenCL [23] and NVIDIA CUDA [32], has greatly contributed to their adoption in modern high-end systems in both (1) the HPC domain, e.g., the Tianhe-1A supercomputer, ORNL's Titan, and the XSEDE Keeneland NSF Track II machine used in our work, and (2) in the enterprise domain, e.g., Amazon's HPC cloud offering and Hewlett-Packard's 'Moonshot' server system [18].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
XSEDE'14, July 13–18, 2014, Atlanta, GA, USA
Copyright 2014 ACM 978-1-4503-2893-7/14/07...$15.00.
http://dx.doi.org/10.1145/2616498.2616531

Variations in heterogeneity among high performance systems are diverse. Nodes in the Titan machine, for example, are each comprised of a single-socket 16-core CPU, 32 GiB of main memory and one NVIDIA K20x 'Kepler' GPGPU, all interconnected via a Cray Gemini network fabric. In Keeneland, each node contains 16 CPU cores across two sockets, each with their own I/O hub to which three NVIDIA M2090 'Fermi' GPGPUs and an InfiniBand card are connected. In the latter system, applications experience varying memory access latencies among not only CPU threads, but also among the GPGPUs and network fabric, requiring host-memory copies to traverse I/O hub boundaries. Pushing towards exascale systems, we can expect to see increased heterogeneity [24] [30] [18] [9] and a greater dependence on software (runtime) systems enabling developers to cope with it [15] [13], including to manage its resources [8].

Management of current and future systems creates complex challenges for both applications and systems software. Not only must applications incorporate a variety of accelerator programming models, but they must also become aware of data partitioning challenges, data movement costs, and invariably, the physical configuration of the underlying high-performance machine. Modern management software for high-performance systems has begun to incorporate aspects of heterogeneity into its algorithms, albeit limited, e.g., Torque enables cluster administrators to specify how many GPGPUs are present on each node, with recent research exploring the execution of multiple applications on shared hardware [39]. Yet, to maximize efficiency, the current state of the art remains such that developers must profile their codes and modify them to exploit specific hardware characteristics, and in addition, such modifications are subject to change across deployed computer systems, thus requiring investments in time and development for porting. Finally, beyond efficient resource usage by e.g., single simulation codes, difficulties are compounded for new usage models expected for high-end machines, where simulations run alongside online data analytics or visualization codes used to assess simulation validity [1] [30] [31]. To study heterogenous cluster hardware and its efficient use by applications, we advocate that clusters should be presented to applications as dynamic 'slices' of their hardware, supported by techniques to understand how to best provision slices for high performance and to gain insights into application characteristics that present opportunities for fully exploiting cluster hardware. We present *GPGPU Slices*, each providing an application with exactly the resources it needs, via a portion of the cluster's resources – CPUs, GPGPUs, and memory – where slice allocations can be made at sub-node granularity. GPGPU Slices leverage

profiles that describe an application's GPGPU usage and characterize its sensitivities to changes in the logical platform, including changes affecting GPGPU locality and host-GPGPU interactions, e.g., data movements and dependencies between GPGPU operations. Profiles are extracted automatically, from observations of an application's use of the accelerator API (e.g., its CUDA calls). Profiles and data about current resource availability, then, is used to compose for each application the assembly it requires.

The specific contributions in this paper are to: (1) present our notion of a cluster 'slice' and how its realization in our software runtime can benefit studies of GPGPU applications on high-performance systems; (2) demonstrate at scale the efficiency of our runtime infrastructure, and the use and extraction of accelerator profiles associated with applications which guide the efficient formation of a hardware slice; and (3) discuss insights about GPGPU application characteristics that enable more efficient exploitation of the heterogeneous high end machines. Cluster slicing is implemented and evaluated for Linux-based environments and NVIDIA CUDA GPGPUs. Experimental results obtained with realistic HPC codes such as LAMMPS, an S3D mini-application from the SHOC benchmark suite [10], and NAS-LU [35] on the Keeneland machine demonstrate the use of slicing to provide gains of up to 5x in system throughput for mixed workloads on 72 nodes, and the scaling of a single GPGPU Slice to 10 GPGPUs for LAMMPS with overheads less than 15% and up to 111 GPGPUs for SHOC/S3D with less than 1% overhead.

2. GPGPU ASSEMBLIES

To accomodate new usage models where exascale systems are expected to host multiple applications on the same set of physical nodes, we propose that the provisioning and management of hetergeneous cluster resources be done using *slices* of the cluster hardware. Slices offer a more flexible granularity of provisioning that can adapt to specific applications' needs by exposing components internal to the cluster nodes for use. For example, GPGPU-'heavy' application codes may need only enough CPU resources (not necessarily unique CPUs) to maintain access to and drive the use of available GPGPUs in the cluster, leaving all remaining CPU resources across selected nodes available for provisioning to other slices. A more hybrid code which may, for example, leverage multiple computational libraries, some of which can or cannot take advantage of GPGPUs underneath, can be given a more diverse slice on the cluster, with a variety of CPUs and GPGPUs selected across some number of nodes in the cluster. There may also be codes which do not take advantage of GPGPUs or other modern accelerators, such as from legacy implementations, which may be best adapted to cluster slices containing only CPUs and some sufficient amount of host memory. Lastly, codes well-tuned for the underlying physical cluster and node configuration may use a slice entirely comprised of whole nodes.

The presentation of a slice to an application is in the form of a *GPGPU Assembly*, a processing-centric, software-abstracted view of the resources belonging to a slice. Providing an abstracted view of resources within a slice enables increased flexibility in the provisioning of such resources. For example, in current systems, codes that aggressively 'scale out' in the number of GPGPUs used must explicitly incorporate distributed programming models to enable partitioning of their work across multiple nodes. With a GPGPU Assembly, there is no need to re-program a code when it wishes to use more accelerators than those present on a single node of the underlying high performance machine. Instead, using an assembly, it simply scales to some greater numbers of GPGPUs available in the cluster, by 'borrowing' GPGPUs from other physical nodes and accessing them remotely. The slicing runtime, i.e., the GPGPU

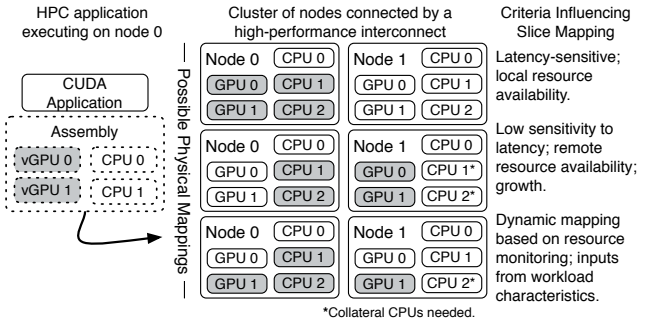


Figure 1: Example assembly mappings to slices provided by our cluster slicing runtime, considering various constraints.

Assembly, assists the application in transparently and efficiently accessing such remote accelerator resources.

The ability of applications to run on the resulting alternative logical machine configurations is enabled by the well-defined APIs with which today's applications access GPGPUs (i.e., with the CUDA or OpenCL APIs in our current implementation). Our slicing runtime, then, composes slices and constructs suitable assemblies, interposing and redirecting API calls to the physical GPGPUs according to the mapping instantiated between the application and underlying physical GPGPUs. Applications simply link with the interposing assembly library, and the runtime ensures efficient GPGPU access and use. The GPGPU assembly abstraction leverages the ideas of device virtualization [2] [6] [27] [29] [17] – through device remoting not visible to guest virtual machines [25] – and of performance isolation [6]. Its implementation, however, does not require hypervisor-based hardware virtualization, but instead, implements these concepts at user-level, to be able to run on today's large-scale cluster machines hosted by standard Linux operating systems.

2.1 Assembly Mappings

Logical representations of physical GPGPUs within a cluster slice are accomplished using the notion of a *virtual GPGPU* or *vGPU*. Together with an enumeration of CPU resources and a reservation of host memory they collectively form the *logical* hardware platform. The slicing runtime's current implementation classifies vGPUs as either *local* or *remote* based on whether a single node hosts both the physical GPGPU and the host application thread. Applications are not made aware of this classification as the use of an assembly is transparent, enabling applications to program vGPUs in the same manner as without an assembly. An assembly's composition is determined at runtime, immediately followed by the first instantiations of each vGPU's mapping to their physical counterparts in the cluster. Applications may 'release' a GPGPU by cleaning up resources it allocated, enabling an assembly to dynamically reattach vGPUs to others within the cluster.[1]

Figure 1 illustrates three alternative mappings for an assembly configured with two vGPUs and two general purpose cores. The first mapping is implemented using local GPGPUs for a latency-sensitive application; the second one uses non-local GPGPUs to accommodate throughput-sensitive application requiring more than the two GPGPUs that a node could offer, or where local resources were not available; a third includes the use of more than one node as influenced by active monitoring to better utilize node resources. The benefits derived from such flexibility are that (1) applications can scale beyond the statically configured nodes' physical limitations, (2) there is added opportunity for efficiency in resource allocation when mapping parallel workloads to cluster hardware, and

[1]HPC codes typically do not release GPGPU resources until their computation has completed.

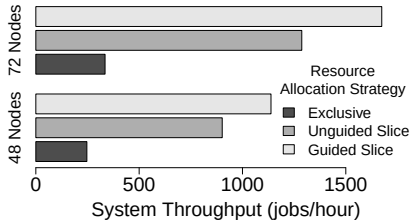


Figure 2: System throughput measurements comparing a standard backfill technique with two scenarios for forming GPGPU Assemblies, one considering application behaviors and sensitivities, and the other not. The experiment assumes a continuous arrival of three HPC applications on a shared cluster (LAMMPS, NAS-LU and SHOC/S3D) across two separate cluster sizes on the ORNL Keeneland Initial Delivery System (KIDS).

(3) applications become more portable, as GPGPU Assemblies can be used to adapt the hardware mapping to match what applications expect. We show some concrete configurations in Section 5.

Applications using GPGPU Assemblies may be single- or multi-threaded, composed of multiple processes, and they can span multiple nodes, as is the case with MPI-based cluster codes [38]. Given their typically diverse needs, criteria for assembly formation can be varied to match different application expectations, workload characteristics, and desired levels of cluster utilization. A contribution of our research is the runtime support needed to enable flexible assembly formation and use, described next.

2.2 Performance and Interference Assessment

For flexibility and accuracy in assessing the utility of an assembly to its application and the potential effects of its hardware slicing on other codes, we offer explicit support for assembly profiling and monitoring. (1) Inspection and tracking of an application's CUDA call history is used to construct an *accelerator profile*, which intuitively, describes how an application uses GPGPUs. The analysis is performed offline, with data gathered via our CUDA library interposer observing CUDA calls and marshaled RPC invocations from an application run using a vGPU. As functions and their arguments are marshaled for transport, their properties are recorded in memory and provided in a history file for further processing. As part of future work, we will include (2) additional capture and use of system-level performance data from Linux (e.g., CPU utilization and memory usage) as well as additional user-programmable GPGPU instrumentation [12]. Both will jointly serve to maintain a *dynamic assembly profile*. This paper focuses on offline analysis of the CUDA API used by an application to define such a profile. This profile may later also include an assembly's use of other shared resources, such as CPU memory, GPGPU memory, and the cluster interconnect. (3) Finally, such profile data is used to assess an application's ability to use remote GPGPUs, which we term a *sensitivity model*, described further in Section 3.

Dynamic tracking and maintenance. Application behavior may change over time, e.g., due to multiple distinct execution phases. Like other high-end codes, LAMMPS [38], a molecular dynamics simulator used in our experiments, for instance, alternates between a compute intensive GPGPU phase and an IO intensive check-pointing phase. The runtime can adjust to such changes by re-evaluating, at runtime, the mappings of vGPUs to GPGPUs. Potentially large GPGPU-resident application state, however, prompts us to confine such changes to situations in which application processes terminate and/or explicitly release (or re-acquire) GPGPUs.

High-performance execution on well-utilized systems. The principal case for GPGPU Assemblies explored in this paper is to demonstrate the utility and efficiency afforded via slice management of heterogeneous cluster hardware and to identify insights attained from applications using them, via experiments illustrating gains in application scalability and increasing levels of system utilization and throughput guided by an application-specific accelerator sensitivity model. As shown in Figure 2, GPGPU Assemblies enable concurrent execution of three high-performance codes, with measurements demonstrating a sustained increase of up to 5x in system throughput on a cluster of 72 nodes – utilizing a total of 828 CPUs and 144 GPGPUs – when mapping hardware slices assigned to multiple CPU- and GPGPU-configured codes onto overlapping cluster nodes. This high utilization of cluster resources is obtained without compromising the performances of each code being run as application profiles guide the composition of hardware to meet their needs for scale and for (low) levels of perturbation. Additional results in Section 5 show improved performance for codes like SHOC/S3D and demonstrate the efficiacy of GPGPU Assemblies to enable at-scale studies of such shared application environments, providing insights into application behaviors which increase the exploitation of heterogeneous clusters.

GPGPU Assemblies use sophisticated GPGPU-resident monitoring to assess potential differences in how well GPGPU resources are used for locally vs. remotely accessed GPGPUs and in addition, to inform programmers about the efficiencies of GPGPU and CPU resource usage in their applications. Such insights can be used by application developers to assist the runtime by classifying their application kernels into categories indicating the utility of GPGPU remoting, e.g., by disabling remoting for certain application kernels due to their latency sensitive nature. The following sections discuss principles governing the effective use of GPGPU Assemblies, both at the application and hardware layer, and metrics enabling the runtime to characterize workloads according to these guidelines.

3. APPLICATION PROFILES

In composing an assembly for an application, it is important to understand the application's behavior and expectations, such as those pertaining to its GPGPU performance, so that it can be determined whether it's suitable for mapping to an assembly consisting not of tightly coupled, on-node GPU accelerators, but of remote ones instead. Accomplishing these goals requires understanding the impact of a heterogeneous environment on the various elements comprising high-performance GPGPU codes. We apply the aforementioned profiles and sensitivity model for these purposes.

Accelerator Applications. Typical accelerator applications are written with host and accelerator portions intended for execution on two different classes of processors resident on nodes: general-purpose host CPUs and GPGPUs. The host portion is responsible for making the supported GPGPU runtime calls for programming the GPGPU: initiating data transfers between the two disjoint memory subsystems, synchronization, launching computational kernels, and to execute any other required host-only computations. Thus, a GPGPU alone is insufficient to initiate compute kernels. Additional resources required to configure the state of the GPGPU include available CPU, memory, and interconnect bandwidth (e.g. PCIe and/or some internode fabric). Such constraints must be taken into account when forming a GPGPU Assembly and determining how assemblies can share the underlying hardware resources.

Accelerator Profiles. As GPGPUs are not (yet) first-class citizens, they do not have arbitrary access to system memory, the CPU cache, or the network. Application codes, therefore, must explicity program the movement of data and control between the host and GPGPU environments. The frequency at which the network is accessed, for one, is directly correlated to a code's remoting sensitivity, further influenced by both network latency and bandwidth. Additionally, the degree to which an application utilizes a network's capacity suggests a greater influence from perturbation effects when that resource is shared. Given the ability of our runtime to transparently map vGPUs to GPGPUs on any node within

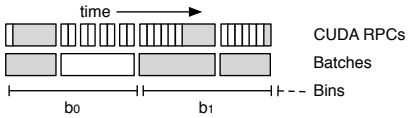


Figure 3: CUDA RPCs, their containing batches (messages) and an example binning interval are shown. Different bin sizes allow for finer or coarser sampling of messages sent by applications. Gray boxes indicate synchronous operation, white boxes asynchronous, and box widths the relative data size.

the cluster, we represent this sensitivity via an *accelerator profile*, or α-profile for short:

$$\alpha = \{S, n\}$$

where n represents the total count of GPGPUs assigned to an application's slice, and S the value of our sensitivity metric, defined below, describing the application's relationship with GPGPU resources.

Sensitivity Model. When accessing GPGPUs, application performance depends not only on physical network properties, but also on how it accesses GPGPUs, including the frequency of access, inherent "access" costs, the amount of data moved with each access, whether data transfers can be overlapped with application computation (i.e., whether communication is blocking), and finally, currently available network bandwidth—specifically considering the network is a shared resource. An application's use of a GPGPU Assembly enables us to extract these characteristics. Each use of the network is represented by a message, characterized with a size in megabits, transmit latency in seconds, and time stamp indicating when it was made available for transmission on the network. Across an application's entire execution we *bin* these messages, e.g., into intervals of one second (Figure 3). For each bin b, we compute S_b consisting of two properties: (i) an estimation of bandwidth utilization and (ii) a sum of latencies contributed by each message

$$S_b = \frac{size_b}{len_b} \cdot \frac{1}{W} + delay_b \tag{1}$$

where W is the maximum bandwidth (mbps) afforded by the network fabric, *size* the sum of message sizes, *len* the interval length, e.g., half a second, and *delay* the total time spent blocking on each message; asynchronous messages do not contribute to *delay* as we assume perfect overlap of transmission with progress. An application's consequent 'sensitivity' to using remote accelerators, S, is then calculated via the mean of all S_b

$$S = \frac{1}{N} \sum_{b=1}^{N} S_b \tag{2}$$

where N is the total number of bins.

Both components in S_b represent different and important characteristics of the application with respect to its use of the GPGPU. Incorporating a component representing time spent waiting by an application, $delay_b$, gives a direct indication of an application's sensitivity to changes in the physical network, e.g., using PCIe vs Ethernet—larger delays mean larger sensitivity, and thus greater overhead. Inclusion of a measure of bandwidth utilization gives an indication of an application's sensitivity to *shared* use of a network, e.g., with utilization of 90% an application would not only have a higher *chance* of experiencing slowdowns from the shared use of that network, but also potentially also experience greater perturbation effects, resulting in further overheads.

Providing one value for S in an application's α-profile (e.g. as opposed to n values of S, one per GPGPU) assumes meaningful differences in access characteristics among all GPGPUs used by the application are negligible. Furthermore, defining S as an average makes the assumption that the access characteristics of an application with respect to a given GPGPU does not exhibit significant changes over time. We provide the definitions as above based on our observations that, for applications we examined in this study, variances in such characteristics were negligible.

Furthermore, in this work, we do not yet consider the different types of traffic seen on the interconnect, including application-level communications vs. those due to remote GPGPU access, and the control communication used by assemblies.

Application Model. We define the resources needed by an application as a *compute profile*

$$\rho = \{cpu.n, cpu.act, M, \alpha\}$$

where $cpu.n$ represents a count of CPUs required by the application, $cpu.act$ the CPU 'activity' coarsely defined as an average level of utilization of those specific CPUs with a value between zero and one, M represents the amount of host memory required per CPU and α the accelerator profile. Defining an application based on this set of metrics forms a basis for constructing an assembly for the application by identifying specific resources in the heterogeneous cluster at the time of the request. The use of these models will provide future cluster schedulers or assembly specific runtime managers with sufficient information to make decisions regarding GPGPU Assembly formation, and static as well as dynamic changes to mappings. Our current work uses the application profiles and sensitivity metric to determine suitable assembly configurations a-priori, offline.

4. IMPLEMENTATION

Cluster hardware slicing is implemented with a distributed runtime developed for heterogeneous GPGPU-based high-performance clusters, such as the XSEDE Keeneland (KIDS) supercomputer used in our experimental evaluations. The runtime's primary component is a distributed collection of stateful, persistent dæmon processes with each node hosting one instance. Collectively, the dæmon processes maintain monitoring information and service requests for assemblies from applications, with one process designated as the master for responding to GPGPU Assembly requests.

GPGPU Assemblies. The runtime's interposer library implements a subset of the CUDA API to enable transparent access and use of local or remote assembly GPGPUs. Upon the first interposed CUDA function invocation, the library communicates with the local dæmon to request an assembly, carrying with it information about the application such as that captured by the application profile, which in turn forwards the request to the master. The master uses information about unmapped CPUs and GPGPUs together with application profiles to locate available resources to compose an assembly, which is sent back to the originating node. Once received, the library immediately *maps* the assembly onto the cluster: network links are instantiated for any remote vGPUs provided; CPUs required to host the remote paths are reserved on the destination nodes. Application execution resumes and subsequent function invocations are intercepted and either passed through to the CUDA library (for local vGPUs) or are marshaled and enqueued into a memory region associated with a specific vGPU, and are finally sent over the network to the vGPU's physical counterpart for unmarshaling and execution. Dæmons spawn a thread for each incoming remote vGPU connection to, which are then responsible for umarshaling and executing each RPC, passing it through to the CUDA library. CUDA applications configure themselves for a host machine by first querying for the number of GPGPUs available. When a GPGPU Assembly is assigned the application, the response to this query is the number of vGPUs the runtime has created in the assembly, where zero or more vGPUs can be assigned to the same physical GPGPU. To maintain compatibility, vGPUs are assigned numbers in the contiguous range 0-N.

Network Optimizations. Two optimizations for the data-path of remote vGPU mappings are RPC batching and batch pipelining,

and rely on identifying each function in the CUDA API as either synchronous or asynchronous. The latter are queued into a batch to reduce accumulation of setup costs associated with transmissions. Flushes are invoked either when batch queues are full or a synchronous call is encountered. These optimizations are examined further in Section 5.

Profiles. Once a GPGPU Assembly is mapped, the batching subsystem can be configured to record a history of information pertaining to all batches, headers, and return state that are sent across a vGPU data path. For each, the following information is recorded: a time-stamp, size in bytes, and for blocking batches a total latency (time spent executing on the remote GPGPU is subtracted from the sender's measurements). This data is fed into a script which performs message binning and sensitivity calculations. CPU activity is measured with standard tools such as `top`.

5. EVALUATION

We present a study on the efficiacy of slices for provisioning resources on hetergeneous clusters in environments of one or more regularly behaving high-performance codes. Scenarios are constructed for demonstrating the abilities of GPGPU Assemblies to: (1) efficiently provide resources for applications with increasing computational demands, designated as 'scale-out' experiments; (2) identify specific application characteristics which enable adaptation to the flexibility afforded by GPGPU Assemblies, leveraging accelerator profiles and our sensitivity models described earlier in Section 3; and (3) enable applications with slices mapped to overlaping computational nodes at large scales up to 72 nodes to experience reduced perturbation and increased cluster throughput, assisted by insights from (2).

Platform. All applications are run on the 120-node XSEDE Keeneland Initial Delivery System ('KIDS') [22] [45]. Every node is configured with two six-core Intel Xeon (Westmere) X5660 2.8 GHz CPUs, 24 GiB of main memory, three NVIDIA Tesla (Fermi) M2090 GPGPUs, and a Mellanox QDR InfiniBand card. CentOS 6.2 hosts the environment, providing CUDA SDK v3.2 through v5.0.

Applications. Applications presented in this evaluation were chosen to represent the types of codes commonly executed in HPC environments to obtain a realistic sense of the capabilities of GPGPU Assemblies: LAMMPS [38], NAS-LU [35], and an S3D chemical reaction simulation kernel provided in the SHOC [10] benchmark suite. LAMMPS is a classical molecular dynamics simulator, dividing a 3-D space of atoms among a grid of MPI ranks. Input scripts define the simulation characteristics, such as melting metals, providing diverse runtime characteristics, e.g., use of the GPGPU (USER-CUDA backend) or CPU, or varying memory footprints. Our observations show LAMMPS, despite various configurations, to behave fairly regularly. NAS-LU is an implementation of NASA's Lower-Upper Gauss-Seidel solver for GPGPUs using a hybrid of MPI and CUDA. It was developed to compute on three different 'classes' of problem sizes (A, B, C in increasing complexity). SHOC/S3D is a weakly scalable code and is configurable with four discrete input sizes[2], and the number of iterations the main kernel is launched. *MatrixMul* from the CUDA SDK is a throughput-sensitive data-parallel processing code, allocating three host- and GPGPU-resident matrices, sending two to the device for multiplication, then receiving the result matrix into host memory. Varying input sizes enables stressing the CPU/GPGPU data-path within the runtime.

[2]SHOC documentation states problem category '4' used in our evaluation is intended to represent "HPC-Focused or Large Memory GPGPUs".

Profiles. An examination of an application's GPGPU access behaviors – its accelerator profile – can shed light onto its resource requirements and behavioral patterns. As discussed in Section 4, a profile is calculated from a recording of messages (batches of CUDA API RPCs) sent between an application using a vGPU and the physical GPGPU counterpart via instrumentation of the batching layer within a GPGPU Assembly. Presenting this sensitivity metric to the runtime upon application launch enables it to guage the extent of the flexibility it can afford for a specific application when composing a GPGPU Assembly. Specific values within each application's profile are shown in Figure 4.

Code	$\rho = (cpu.n, cpu.act, M)$	α (accelerator profile)
LMP_{cpu}	(648, 1, 2.3Gb)	*cpu backend*
NAS-LU	(6, 0.9, 3.7Gb)	(0.14, 6)
SHOC/S3D	(36, 0.05, 3.8Gb)	(0.0486, 36)

Figure 4: Profiles for each application as configured for the experiment in Section 5.2.

NAS-LU. Figures 5(a) and 5(c) illustrate an isolated experiment with NAS-LU using a single vGPU. An increase in the overall problem size corresponded to a decrease in the sensitivity, explained by an overall growth in the size of RPC messages and more time spent waiting for the GPGPU to complete each computation. Larger messages enable more efficient use of the network's bandwidth; more time spent waiting on the GPGPU means a reduced exposure to elements on the network. Its sensitivity showed very little variation across execution (except at initialization). Measurements with UNIX utility `top` shows NAS-LU to exhibit relatively high CPU utilization, suggesting additional host-side computations are performed in addition to GPGPU computations.

SHOC/S3D. Configured with the largest problem size category and 1200 kernel invocations, SHOC/S3D exhibits lower sensitivity to remote GPGPUs than NAS-LU, but similarly with little variation throughout execution (Figure 5(b)). Measurements show SHOC/S3D to have nearly negligible CPU requirements, as the code performs the bulk of its computations on the GPGPU.

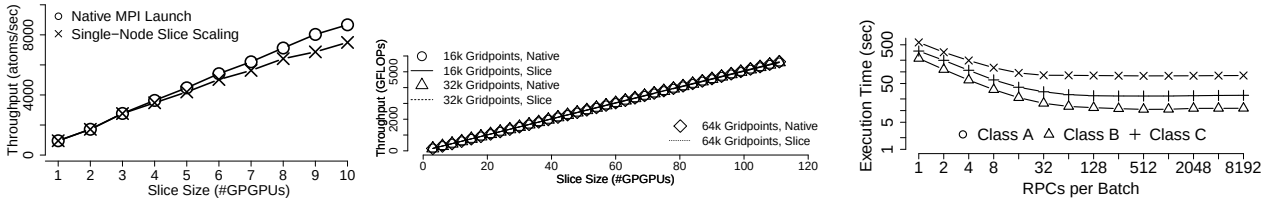
LAMMPS. We evaluated the GPGPU backend of LAMMPS in 'scale-out' experiments, and the CPU backend in our at-scale performance study (Section 5.2). Measurements show the GPGPU backend to have significant CPU utilization and a reduced capability to leverage the flexibilities of a GPGPU Assembly, indicated by our calculations of its GPGPU sensitivity (omitted for brevity). Execution of the CPU backend exhibited full use of available CPU resources, as expected.

5.1 Application Growth

Application growth on clusters can be achieved by spawning additional ranks, spilling to other nodes when local resources are exhausted. GPGPU Assemblies can provide for such growth *dynamically* by searching for and 'borrowing' available GPGPUs found throughout the cluster, establishing an efficient data path over the interconnect, and presenting a different logical view of the cluster.

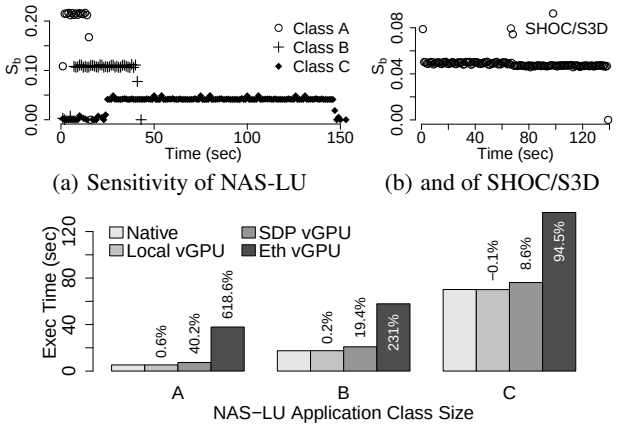
Figure 6(a) compares the throughput of LAMMPS with and without an assembly up to ten GPGPUs. As the problem size grows, more ranks are created. Beyond three GPGPUs, the MPI runtime spreads ranks across the cluster. With an assembly, all ranks remain on the same node, using an efficient data path established by the assembly to access remote GPGPUs. At ten GPGPUs, performance differs by 13.5%, an overhead attributed to increasing pressure from the additional ranks on CPU resources (e.g. the cache), and its sensitivity to use of the network, compounded with each additional GPGPU. In contrast, SHOC/S3D scales near linearly up to 111 GPGPUs – 111 ranks on the same node – due to low CPU resource needs and extremely low GPGPU sensitivity.

Insights. CPU utilization must not be ignored when altering



(a) LAMMPS scale-out (b) SHOC/S3D scale-out (c) RPC batching effects on NAS-LU

Figure 6: (a) Performance comparison between LAMMPS executing with three ranks per node vs all ranks launched within a single node within a GPGPU Assembly providing increasing numbers of GPGPUs (first three mapped locally, the remainder remote). Each data point between configurations use the same simulation parameters. (b) Similarly with (a) except with SHOC/S3D and across increasing problem sizes, up to 111 GPGPUs. (c) Performance of NAS-LU with a single remote vGPU using SDP, varying the RPC batch size. Each point is the mean of five runs. Note that both axes are logarithmic.



(a) Sensitivity of NAS-LU (b) and of SHOC/S3D

(c) Effect of vGPU choice on NAS-LU performance

Figure 5: (a)-(b) Sensitivities S_b for two applications plotted over time in single-vGPU execution. A bin of 0.5 seconds and $W = 1211$Mbps (SDP) were used. (a) Four outliers are not visible: 0.4, 1.0, 1.4, 1.5 all occurring between 20-30s into execution for problem size C. (b) Configured with problem size 4 and 1200 passes. (c) Execution times for (a).

a GPGPU Assembly to accomodate changes in application needs, as it may diminish the ability of an application to drive 'borrowed' GPGPUs across the cluster, both locally and on remote nodes.

5.2 Multi-Slice Provisioning

Challenges in managing large-scale heterogeneous systems include the ability to (1) transparently express and maintain logical resource provisions for applications and (2) provide flexible environments in which to study them. We next present a scenario with multiple HPC GPGPU codes using cluster slices to provision resources and study the importance of application profiles in their construction.

In this experiment we consider a scenario providing an opportunity to make different decisions for composing GPGPU Assemblies for individual applications in a shared environment. The first trial does not consider our application profiles and the second does. Our goal is to show that both scenarios are made possible using slices, and that we can draw insights using knowledge obtained from our application profiles to explain our measurements. We evaluate the following large-input configurations of the applications on two separate cluster sizes of 48 and 72 nodes each (Figure 9). Figure 8 illustrates the various GPGPU Assemblies created for each application and their slice for both trials considered. Figures 2 and 7 present the measured system throughputs for both trials and cluster sizes, and the measured execution times across assembly configurations.

The first two measurements (left-most tic marks) for each application in Figure 7 illustrate baseline performances: one as we might expect to launch the application – 'filling' each node – and

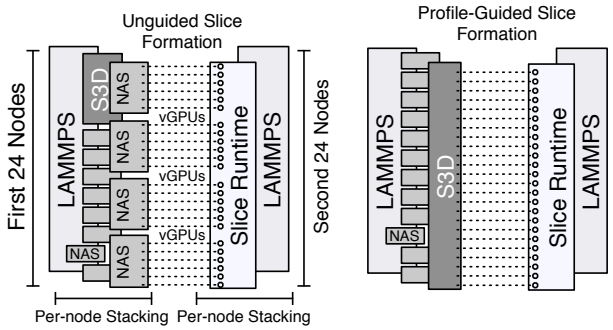


Figure 8: Configuration of experiment for applications described both in the text as well as in Figure 7, for 48 nodes; on 72 nodes additional NAS-LU instances exist, and SHOC/S3D and LAMMPS request more ranks. LAMMPS is given an assembly which alters its rank-to-node placement such that 8 ranks/node exist on the first $N/2$ nodes, and 10/node on the second half. For NAS-LU and SHOC/S3D, assembly vGPUs map to unique remote nodes, always on the latter $N/2$ nodes. 'Per-node stacking' illustrates the overlap between application and runtime-related instances on shared nodes.

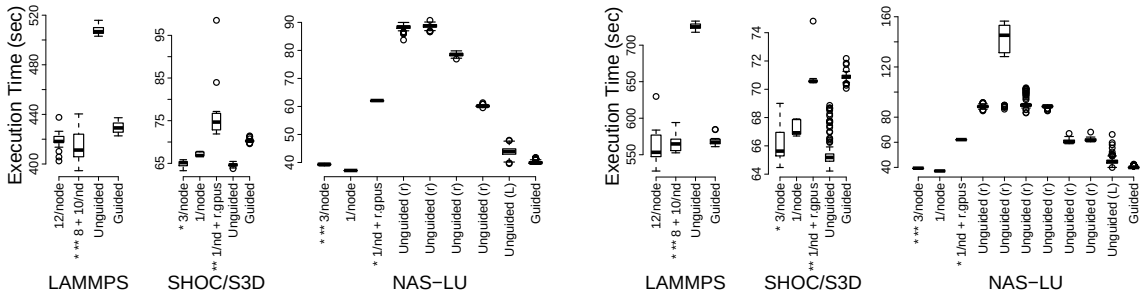
Code	Configuration	Ranks
LMP_{cpu}	'eam' simul. 536M atoms	432, 648
NAS-LU	problem size 'C'	6, 6
SHOC/S3D	problem size '4' 1.2k iters	24, 36

Figure 9: Configuration of apps in shared-slice provisioning scenario. Ranks listed are for 48 and 72 nodes respectively.

another which facilitates better co-placement with other codes on the cluster for the purposes of this experiment. For GPGPU codes, the third measurement illustrates how sensitive a code is to use of remote GPGPUs: 11.7% and 5.5% for SHOC/S3D on 48 and 72 nodes respectively, and 67.2% for NAS-LU. Our measured sensitivity metric matches this behavior (Figure 5).

Naïve mappings of each assembly, such as those in the unguided trial, present pronounced effects on the performances of LAMMPS and the majority of NAS-LU instances launched. With examination of their profiles (Figure 4) we assert that the overheads for the NAS-LU instances, which were assigned remote GPGPUs, experienced perturbation from both (1) exposure to the use of the cluster network, as suggested by the GPGPU sensitivity, and that (2) a significant enough amount of contention on CPU resources existed on nodes where there was substantial overlap of slices chosen for both applications; both LAMMPS and NAS-LU have a high measured CPU utilization.

As assembly mappings are transparent to applications, we alter them for the second trial, guided by our application profiles. When the applications are launched, the profiles are provided to our runtime which directly compares the sensitivity values when prioritizing applications chosen for remote GPGPU assignment. SHOC/S3D's low GPGPU sensitivity suggests it is the better candidate which can adapt to the flexibility of an assembly. All NAS-LU instances which were using remote-GPGPU assemblies are mapped locally, and SHOC/S3D is scaled across all nodes. Codes which



(a) Application Execution Times on 48 Nodes.

(b) Application Execution Times on 72 Nodes.

Figure 7: Effects on application execution times on a shared cluster when the formation of GPGPU Assemblies considers application profiles ('Guided') and when not ('Unguided'). Experiment ran separately on 48 and 72 nodes. n/nd signifies n ranks launched per node and $n+m$/nd with n/nd launched on the first half of nodes, m/nd on the remainder. Guided and Unguided signify the configuration and trial in which the application executed; see Figure 8. A single asterisk '*' signifies that configuration was selected for the Unguided trial; a double asterisk '**' signifies the same but for Guided. An 'r' or 'L' suffixed to Unguided signifies the assembly configuration providing **r**emote or **l**ocal GPGPUs for that trial. Figure 2 compares the measured system throughput across both scenarios with a standard backfill approach for applications configured as 12/node and 3/node.

have high measured CPU utilization now have sufficient CPU resources for computation and GPGPU management, and no code is subject to the network which cannot afford it. Measurements show the new mappings to have reduced perturbation for all codes, shown in Figure 7, and increased overall system throughput, shown in Figure 2.

Insights. Our runtime is able to efficiently compose slices at-scale for studying shared application environments on heterogeneous clusters, and an understanding of application characteristics is vital to the composition of slices, e.g. for identifying possible points of resource contention.

5.3 Data Path Efficiency

Diving into the lower layers of the runtime system, this section discusses the efficiency of the data paths provided by vGPUs to applications, using standard CUDA SDK microbenchmarks.

RPC Batching. Hosting more complex and longer-running codes requires analysis of additional costs and optimizations available on the network when implementing remote vGPU mappings. With the sheer number of RPCs generated by LAMMPS and NAS-LU, sending each individually greatly compounds network costs. Two techniques are used in our work, batching and message pipelining. An experiment illustrating their effects is shown in Figure 6(c).

Latency Analysis. We ran MatrixMul on a single remote vGPU to identify which components in our runtime implementation contributed the most to each individual RPC created. Results (omitted for brevity) show the largest contributor to RPC latencies are due to the NVIDIA library, e.g. waiting on GPGPU computations, followed by network transfer operations. Main memory bandwidths were observed to be great enough to provide low overhead within the marshaling routines.

Insights. Network optimizations lead to the greatest initial performance improvements in individual remote GPGPU mappings.

5.4 Summary

Diversity in applications' use of high-performance clusters may be mismatched with the physical configurations present in modern cluster infrastructures. There is much opportunity to investigate the way applications are configured, how they use the hardware given to them, and what assumptions they make to provide for potentially better configurations. The experiments described in this paper show assemblies can be used to study such environments, and be used to construct effcient mappings to cluster slices.

6. RELATED WORK

Early adoption of GPGPUs gave rise to research questions asking how to efficiently virtualize the accelerator to be able to *access* these accelerators, either by punching through virtualization layers or by transporting messages across the network. Initial work was focused in the context of hypervisors efficiently exposing GPGPUs to guest virtual machines, such as VMGL [26], GViM [16], vCUDA [42], gVirtuS [14], and rCUDA [11] all of which utilize API interposition, passing marshaled communications through local sockets or hypervisor-assisted shared memory. This technique was later adopted by user-space middleware systems, such as the work presented in this paper, in addition to VOCL [46], and MOSIX [5]. Our runtime leverages the insights gained from the aforementioned research in its development to construct efficient GPGPU Assemblies for the slices it assigns applications in large-scale heterogeneous clusters, such as consideration of online networking optimization techniques and use of lower-level communication layers (e.g. SDP RDMA operations vs MPI), as well as adopting the ability to permit seamless execution of unmodified applications in an assembly composed of both local and remote GPGPUs.

Additional research in middleware solutions focuses on increasing programmability as well as the development of unified platforms for task scheduling on *single nodes* consisting of CPUs and GPGPUs, leveraging knowledge from the application, e.g. data dependencies gleaned from source annotations. These include libWater [15], StarPU [4], and others [37] [33] [8], as well as research suggesting the operating system—not middleware—should manage the accelerator [34] [40] [21] [20]. Such research efforts may enhance the efficiency of a GPGPU Assembly mapped to a slice, driven by knowledge captured by our runtime and presented to a 'node-level' manager in the form of one of the aforementioned systems, effectively becoming a multi-tier scheduling system, demonstrated also within Rain [41].

Ideas related to sharing compute nodes among applications and enabling software-defined machine configurations to improve overall throughput and hardware utilization are also discussed in [7] [39] [19] and [36]. Becchi et al. [7] consider node (and GPU) sharing among concurrent processes and applications, GPGPU virtualization, and offloading to remote GPGPUs, while Ravi et al. [39] consider the problem of concurrently scheduling applications with different CPU-GPGPU requirements on heterogeneous nodes. Jiménez et al. [19] consider online methods for scheduling applications sharing a *single* heterogeneous node. Our runtime is distinct from all such prior work in taking a step beyond merely demonstrating the feasibility of GPGPU remoting and software-defined machine configurations to instead advocate these existing ideas support a new way of provisioning resources on heterogeneous clusters—using slices—and a new abstraction that enables studying how applications can best take advantage of these provisions. We inves-

tigate in detail: (1) trade-offs and opportunities in disaggregating CPU and GPGPU elements of future HPC machines; (2) the perturbation introduced by mapping multiple *slices* onto shared resources that introduce complex interactions; and (3) proposing a sensitivity analysis to characterize application behavior specifically reflecting the efficiency a GPGPU Assembly will have for a given slice, providing insights on how to best afford the benefits of software-defined mappings onto a shared heterogeneous cluster.

Finally, some efforts explore effective intra- or inter-node GPGPUs usage based on the message-passing paradigm [3] [28] [44] [43]. However, they require usage of specific APIs, and thus code modifications. Our runtime provides a more general approach to resource sharing, and may exploit such libraries to optimize particular aspects of resource sharing.

7. CONCLUSIONS AND FUTURE WORK

With the proven utility of heterogeneous computing, there is a need to provide applications with scalable and seamless solutions for using heterogeneous machine resources. This need is met by GPGPU Assemblies, which present applications with software-defined platforms comprised of exactly the cluster and GPU resources they require. Our runtime implements assemblies by transparently running an application's CUDA requests on local or remote GPUs, driven by metrics that include high application performance and high levels of cluster resource utilization. Experimental evaluations show assemblies to improve levels of cluster throughput while demonstrating an ability to keep perturbation levels minimal, as well as providing an environment to flexibly adjust mappings to cluster slices for observing and studying changes in application behaviors to draw insights into how applications might better exploit future large-scale high-performance systems.

Future extensions to the runtime include supporting dynamic assembly reconfiguration – dynamic node 'discovery', or managing the dynamic growth in GPGPU resource needs for more irregularly behaving codes at large scales, such as image- or graph-processing codes where resource demands are more closely represented through the characteristics of the data, rather than the size of the data.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This project was funded in part by the NSF Track II 'Keeneland' project.

8. REFERENCES

[1] H. Abbasi et al. DataStager: scalable data staging services for petascale applications. In *HPDC*, 2009.
[2] Amazon Inc. High Performance Computing Using Amazon EC2. http://aws.amazon.com/ec2/hpc-applications/.
[3] A. Athalye et al. GPU aware MPI (GAMPI) – a CUDA-based approach. Technical report, University of Texas at Austin, 2010.
[4] C. Augonnet et al. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Euro-Par*, 2011.
[5] A. Barak et al. A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. *PPAAC*, 2010.
[6] P. Barham, B. Dragovic, K. Fraser, et al. Xen and the art of virtualization. In *SOSP*, Bolton Landing, USA, 2003.
[7] M. Becchi et al. A virtual memory based runtime to support multi-tenancy in clusters with gpus. In *HPDC*, 2012.
[8] M. Boyer et al. Load balancing in a changing world: dealing with heterogeneity and performance variability. In *CF*, 2013.
[9] N. Carter et al. Runnemede: An architecture for ubiquitous high-performance computing. In *HPCA*, 2013.
[10] A. Danalis et al. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU-3*, 2010.
[11] J. Duato et al. Enabling CUDA acceleration within virtual machines using rCUDA. In *HiPC*, 2011.
[12] N. Farooqui et al. Lynx: A dynamic instrumentation system for data-parallel applications on gpgpu architectures. In *ISPASS*, 2012.
[13] M. Garland et al. Designing a unified programming model for heterogeneous machines. In *Supercomputing*, 2012.
[14] G. Giunta et al. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Euro-Par*, 2010.
[15] I. Grasso et al. Libwater: Heterogeneous distributed computing made easy. In *ICS*, New York, NY, USA, 2013.
[16] V. Gupta et al. GViM: GPU-accelerated Virtual Machines. In *HPCVirt*, Nuremberg, Germany, 2009.
[17] V. Gupta et al. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *USENIX ATC*, 2011.
[18] Hewlett-Packard. Hp moonshot. http://thedisruption.com/, 2014.
[19] V. J. Jiménez, L. Vilanova, I. Gelado, et al. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *HiPEAC*, 2009.
[20] S. Kato et al. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX ATC*, 2011.
[21] S. Kato et al. Gdev: First-Class GPU Resource Management in the Operating System. In *USENIX ATC*, 2012.
[22] Keeneland web site. http://keeneland.gatech.edu/, 2013.
[23] Khronos Group. The OpenCL Specification. http://tinyurl.com/OpenCL08, 2008.
[24] P. Kogge et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, University of Notre Dame, CSE Dept., 2008.
[25] S. Kumar et al. Netbus: A transparent mechanism for remote device access in virtualized systems. Technical report, CERCS, 2008.
[26] H. A. Lagar-Cavilla et al. VMM-independent graphics acceleration. In *VEE*, San Diego, CA, 2007.
[27] J. Lange et al. Palacios: A New Open Source Virtual Machine Monitor for Scalable High Performance Computing. In *IPDPS*, 2010.
[28] O. S. Lawlor. Message Passing for GPGPU Clusters: cudaMPI. In *IEEE Cluster PPAC Workshop*, 2009.
[29] A. Merritt et al. Shadowfax: Scaling in heterogeneous cluster systems via gpgpu assemblies. In *VTDC*, 2011.
[30] K. Moreland. Oh, &#*@! exascale! the effect of emerging architectures on scientific discovery. SCC'12.
[31] K. Moreland et al. An image compositing solution at scale. In *SC*, 2011.
[32] NVIDIA Corp. NVIDIA CUDA Compute Unified Device Architecture. http://tinyurl.com/cx3tl3, 2007.
[33] S. Pai et al. Improving gpgpu concurrency with elastic kernels. In *ASPLOS*, 2013.
[34] S. Panneerselvam et al. Operating systems should manage accelerators. In *HotPar*, 2012.
[35] S. J. Pennycook et al. Performance analysis of a hybrid MPI/CUDA implementation of the NAS LU benchmark. *SIGMETRICS Perform. Eval. Rev.*, 2011.
[36] R. Phull et al. Interference-driven resource management for gpu-based heterogeneous clusters. HPDC, 2012.
[37] J. Planas et al. Self-adaptive ompss tasks in heterogeneous environments. In *IPDPS*, 2013.
[38] S. J. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comp. Phys.*, 117:1–19, 1995.
[39] V. T. Ravi et al. Scheduling concurrent applications on a cluster of cpu-gpu nodes. In *CCGRID*, Washington, DC, USA, 2012.
[40] C. J. Rossbach et al. Ptask: Operating system abstractions to manage gpus as compute devices. In *SOSP*, 2011.
[41] D. Sengupta et al. Multi-tenancy on gpgpu-based servers. In *VTDC*, 2013.
[42] L. Shi et al. vCUDA: GPU accelerated high performance computing in virtual machines. 2009.
[43] M. Strengert et al. CUDASA: Compute Unified Device and Systems Architecture. In *EGPGV*, 2008.
[44] J. A. Stuart et al. Message passing on data-parallel architectures. In *IPDPS*, 2009.
[45] J. Vetter et al. Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community. *Computing in Science Engineering*, 13(5):90–95, 2011.
[46] S. Xiao et al. VOCL: An Optimized Environment for Transparent Virtualization of Graphics Processing Units. InPar, 2012.