A Dynamic Accelerator-Cluster Architecture

Sebastian Rinke*†, Daniel Becker‡, Thomas Lippert§, Suraj Prabhakaran*†, Lidia Westphal¶, Felix Wolf*†§
*Laboratory for Parallel Programming, German Research School for Simulation Sciences, 52062 Aachen, Germany
Email: {s.rinke, s.prabhakaran, f.wolf}@grs-sim.de
†Department of Computer Science, RWTH Aachen University, 52062 Aachen, Germany
‡Corporate Technology, Siemens AG, 81739 München, Germany
Email: becker.daniel@siemens.com
§Jülich Supercomputing Centre, 52428 Jülich, Germany
Email: th.lippert@fz-juelich.de
¶MAGMA Giessereitechnologie GmbH, 52072 Aachen, Germany
Email: l.westphal@magmasoft.de

***Abstract*—Accelerators such as graphics processing units (GPUs) provide an inexpensive way of improving the performance of cluster systems. In such an arrangement, the individual nodes of the cluster are directly connected to one or more accelerator devices via PCI Express. This results in a static mapping of accelerators onto compute nodes, where each accelerator can only be accessed from exactly one compute node. While this static mapping enables efficient data transfers between a given accelerator and the compute node it belongs to, differing computational demands across jobs may, however, produce either underutilized accelerators or nodes whose computational demands cannot be satisfied with the number of accelerators available to them. In particular, smaller numbers of GPUs available per node may enforce explicit MPI parallelism across compute nodes where it is not necessary. To address this limitation, we propose a novel accelerator-cluster architecture in which network-attached accelerators are dynamically assigned to compute nodes. This allows not only their optimal utilization but also a more precise match between application requirements and accelerator hardware. We outline the general concept of our dynamic architecture and show that it can offer substantial benefits to certain classes of applications without significantly harming the performance of others.**

I. Introduction

Once designed exclusively for computer graphics and difficult to program, today's *graphics processing units* (GPUs) are extremely flexible parallel processors that are often used to enhance the performance of general-purpose processors. This is why such devices are often called *accelerators*. Being optimized for data-parallel throughput computations using large numbers of compute cores, their low density of control logic makes them also extremely energy efficient. Motivated by growing demand for computational power and the desire to minimize energy consumption, cluster architects take more and more advantage of such flexible graphics-processor designs to improve the performance of their systems. A prominent example is the Tianhe-1A system, which ranks number five among the world's fastest supercomputers (Top500 list from Jun 2012).

Accelerators owe much of their success to powerful programming models such as CUDA and OpenCL. Both programming models enable the user to define highly parallel compute kernels for execution on accelerator devices. However, since most accelerators are currently PCI Express (PCIe) devices with their own dedicated memory, data have to be transferred between the main memory of the host and the memory of the device. Thus, a given graphics device in a cluster is directly attached and, therefore, exclusively dedicated to a distinct general-purpose compute node—typically a shared-memory multiprocessor with several general-purpose cores. Since a cluster may provide more than one accelerator per compute node, we can generally speak of a $N-\text{to}-1$ mapping between accelerators and compute nodes. The critical point is the static nature of this mapping, which requires modifications to the hardware to be changed.

While facilitating efficient data transfers between hosts and their accelerators, the static mapping ignores varying requirements across and potentially even within the jobs of a given workload. Some applications may require more than N accelerators per node and some less, others may require none at all. Load imbalance within an application may even create varying accelerator needs among the nodes assigned to the same application. In particular, if the maximum number of GPUs available per node is too small, single-node applications may be required to prematurely embrace parallelism across compute nodes via MPI at the expense of higher development effort and lower programmer productivity. Simply analyzing the job mix and providing nodes with different numbers of accelerators would ignore that requirements of the workload as a whole are also subject to change. For example, after learning how to apply double buffering, a key user may all of a sudden be able to efficiently exploit twice the number of accelerators per node. On the other hand, reconfiguring the cluster hardware too frequently is not only expensive but also jeopardizes system stability. Finally, the static mapping complicates the replacement of broken accelerators, which always affects the

host nodes they are attached to. Overall, the static mapping is neither able to ensure proper utilization of the available accelerator devices nor is it guaranteed to equally satisfy the requirements of all applications in a potentially unstable workload mix.

Inspired by the evolution from node-attached to network-attached storage [1], we propose a novel cluster architecture where the mapping between general-purpose nodes and accelerators is dynamic. Instead of attaching accelerators statically to individual nodes, our architecture maintains a pool of network-attached accelerators that can be assigned to their hosts on demand. As long as there are still accelerators available, every host node can acquire as many or as few accelerators as actually needed. Even among the nodes occupied by the same application, the number of accelerators per node can vary. The primary advantages of our architecture, which can be realized using off-the-shelf components plus a middleware package we have developed, are improved hardware utilization and increased flexibility with respect to the number of accelerators available to individual compute nodes. The primary disadvantage is the bandwidth penalty of the network transfer, which we keep at a minimum using an efficient protocol based on MPI and GPUDirect. In an experiment with linear algebra kernels running on a single host node, we show that the speedup our architecture can offer without any cross-node parallelism is up to 2.2 with reference to what could be achieved using a single accelerator directly attached to this node. On the other hand, the impact of the reduced bandwidth on a hybrid MPI/CUDA molecular-dynamics application with one accelerator per node is almost unnoticeable. In addition, our MPI-based solution can be deployed on almost any cluster, making as little assumptions on the underlying platform as possible.

The remainder of the article is organized as follows: After reviewing related work in Section II, we describe our dynamic architecture in Section III including the required commodity hardware components and the execution model. Section IV is devoted to the middleware needed to assign and access remote accelerators and its prototypical implementation. An emphasis is given to the efficient communication between hosts and accelerators. In Section V, we present experimental results for communication performance, the linear-algebra kernels and the molecular-dynamics code. Finally, we summarize our results and outline future work in Section VI.

II. Related Work

In this section, we review alternative approaches of using network-attached accelerators as well as more flexible ways of connecting accelerators to hosts via PCIe. According to the focus of our paper, an emphasis is given to efficient communication and resource-management aspects.

Designed as a GPU virtualization approach based on a client-server architecture, the rCuda [2] framework enables the execution of CUDA kernels on remote GPUs. Specifically, rCuda allows clients without GPUs to run CUDA kernels on servers equipped with CUDA-enabled hardware. On a more technical level, a client-side wrapper library intercepts CUDA calls and redirects them to a server which executes them on its CUDA-enabled hardware. Once the wrapper library is loaded, it uses the socket API to connect to a set of servers provided in a client environment variable. After establishing the connection, the communication between client and server runs over TCP/IP, which may introduce higher overhead in comparison to our MPI-based solution. In addition, since clients are not aware of each other, the current version (v3.2) of rCUDA does not support coordinated GPU resource management for parallel jobs with multiple processes. vCUDA [3] is a GPU virtualization framework designed for CUDA applications running on virtual machines. Similar to rCUDA, vCUDA is based on a client-server architecture with clients executing on a guest operating system and a server executing on a host operating system with direct access to CUDA hardware. As in rCUDA, clients are granted access to the CUDA hardware by the interception of CUDA API calls and their redirection to the server. Primarily targeting a portable rather than an efficient virtualization mechanism, the communication between clients and server is performed over XML-RPC. As part of its GPU-virtualization approach, vCUDA also provides support for suspending and resuming the client's virtual machine. That is, to resume the client session after it has been suspended, vCUDA automatically restores the latest device state of the virtual GPU. However, the overhead of running the client on a virtual machine together with the XML-based communication protocol makes this solution second choice for HPC cluster environments, where utmost performance is crucial. Originally developed for online games, Zillians Inc. advertize a solution called V-GPU [4] for dynamic GPU provisioning in clouds. Similar to vCUDA, V-GPU intercepts GPU API function calls on clients and redirects them to GPU servers. Like in our architecture, the number of GPUs that can be used by each VM server is configurable. However, whereas our MPI-based protocols offer a maximum of portability, Zillians's implementation based on remote direct memory access (RDMA) on top of Infiniband or 10 Gigabit Ethernet (10GE) is tied to specific network fabrics. At the time of writing, neither the software itself nor any performance results had been published. Moreover, the Many GPUs Package (MGP) [5] allows compute nodes to transparently run OpenCL kernels on both local and remote OpenCL devices in a complete cluster environment. The (remote) devices are selected at runtime or before application start with the help of environment variables. MGP consists of two layers: (i) a MOSIX Virtual OpenCL (VCL) layer and (ii) an API layer. The MOSIX VCL layer provides a runtime environment to the API layer in which all available OpenCL devices are considered local to every

compute node. Based on this assumption, the API layer enables applications to control these OpenCL devices. During program execution, communication between compute nodes and remote OpenCL devices is performed over TCP/IP. As with rCUDA, MGP currently cannot ensure an exclusive assignment of accelerator devices to compute nodes to avoid resource sharing. Finally, the additional TCP/IP transport overhead may again be prohibitive for HPC applications. Designed as an extension of the CUDA programming model, CUDASA [6] can be used to distribute computations over multiple accelerator-equipped compute nodes of a given cluster. Basically, CUDASA augments the CUDA execution model by introducing *jobs* and *tasks* in addition to well-known *kernels*. These higher-level abstractions offer control mechanisms for a distributed environment with multiple network-attached GPUs. The extended execution model is based on a head-node process, which runs the main program, and compute node processes, which are responsible for running the actual compute jobs created by the main program. Within jobs, a data-locality aware scheduler is responsible for assigning work to GPU-equipped compute nodes, trying to reduce communication overhead between compute nodes as much as possible. Data exchange between different nodes of the same job is accomplished via a distributed shared-memory model. However, the focus of CUDASA seems to be directed rather on introducing a programming model for distributed multi-GPU programming than on enabling the flexible assignment of GPUs to compute nodes in current HPC cluster environments.

Given that current accelerators are PCIe devices, solutions for I/O consolidation represent another avenue towards a more flexible accelerator-to-compute node assignment. For example, Dolphin Express [7] describes a PCIe-based switch interconnect featuring both I/O and clustering capabilities. In such an ensemble, every compute node uses its PCIe link for communication with other nodes—just like it uses it to communicate with local PCIe devices. Another PCIe-based solution for I/O consolidation was proposed by NextIO [8], where each compute node in a given rack is connected through a dedicated PCIe link to the same NextIO switch, which provides a common pool of I/O resources. Here, every I/O device can be exclusively assigned to one compute node on demand. In this way, both NextIO and Dolphin Express enable a flexible assignment of PCIe devices such as GPUs to compute nodes. In comparison to a pure software solution, the I/O consolidation approach induces less communication overhead. However, I/O consolidation requires new hardware (e.g., cabling and switches), whereas a software solution like ours makes use of existing network resources.

III. Dynamic Accelerator-Cluster Architecture

In this section, we introduce our dynamic accelerator-cluster architecture, which maintains a pool of network-attached accelerators that can be made appear as locally attached to any compute node in the network. We start with an overview, continue with a description of the individual components, and finally discuss the execution model.

A. Overview

In contrast to the static $N-\text{to}-1$ mapping between accelerators and compute nodes in current accelerator-enhanced clusters, our dynamic architecture couples accelerators with compute nodes more loosely. Instead of being directly connected to a compute node (e.g., via PCIe), an accelerator is attached to a high-speed interconnection network which is shared among all compute nodes and accelerators. This approach enables compute nodes to share a common set of accelerators, and avoids an implicit correlation between compute nodes and accelerators. The primary benefit is a more precise match between application demand and accelerators available in the system plus improved hardware utilization. As a positive side effect, broken accelerators or compute nodes no longer affect the availability of operational compute nodes or accelerators, respectively.

Before it can be used by an application, an individual accelerator is assigned to exactly one compute node. This assignment can be done either statically before application start or dynamically while the application is running. For this purpose, an accelerator resource manager (ARM), which maintains information on which accelerators are available or in use by which compute nodes, handles allocation and deallocation requests. Thus, compute nodes can dynamically allocate accelerators based on application needs. Given that the number of compute nodes and accelerators can now scale independently, every compute node can benefit from additional accelerators attached to the cluster, for instance, during a cluster environment upgrade.

To avoid, however, that the network traffic between compute nodes and accelerators becomes a serious competitor of the traffic between compute nodes for bandwidth, we recommend to keep the number of accelerators smaller than the number of compute nodes. Our architecture is ideally suited for workloads where the number of accelerators required per node varies greatly and where the ratio of accelerators to compute nodes is low. This is typically the case when some but not all applications need accelerators. Otherwise, traditional static accelerator-cluster architectures might be more appropriate. Conceivable is also a mix of both worlds, where a cluster with, for example, one accelerator attached to each node maintains a pool of network-attached accelerators as excess capacity for those codes that need more than one per node. Especially in view of possible incremental installation, our architecture is therefore most attractive for sites that started with a traditional cluster and now want to experiment with accelerators. Below, we summarize the advantages of our concept:

- Economy—a more precise match between application demand and available accelerators allows for improved

hardware utilization.

- Flexibility—the number of accelerators assigned to a given general-purpose node can be modified any time.
- Transparency—access to remote accelerators occurs via familiar programming models and requires only little source-code changes. The details of the communication mechanism between compute nodes and accelerators are hidden from the programmer.
- Fault tolerance—broken accelerators do not affect the availability of compute nodes and vice versa.
- Extensibility—the ratio of accelerators to general-purpose nodes in the cluster as a whole can be easily increased just by attaching extra accelerator nodes to the cluster-internal network. This gives system providers the ability to quickly respond to higher demand.
- Performance—efficient communication protocols minimize the overhead of using remote accelerators in terms of latency and bandwidth.

B. Components

Our dynamic accelerator-cluster architecture can be realized using commodity components found in typical cluster systems, no specialized hardware is required. As depicted in Figure 1, the components of our architecture include compute nodes, accelerators, an accelerator resource manager, and a network. The compute nodes and the network are identical to their counterparts in current cluster environments. Nevertheless, given that a single compute node may have to feed larger numbers of accelerators, we recommend to equip compute nodes with enough memory. Furthermore, when choosing a network fabric, it has to be taken into account that host-device traffic and traffic between compute nodes share the same network bandwidth. Again, the ratio between accelerators and compute nodes is an important parameter to consider in this context.

1) Accelerator: As shown in Figure 2, we consider an accelerator as being composed of an energy-efficient CPU plus main memory (RAM), a network adapter (NIC), and a typical accelerator such as an NVIDIA Fermi GPU. In contrast to current accelerator types (e.g., GPUs), we define accelerators as components that can communicate with other components over an interconnection network. Hence, the network adapter connects the accelerator to the network for communication with compute nodes, the accelerator resource manager, and other accelerators. However, since current accelerator types are not able to initiate network communications, an energy-efficient general purpose CPU runs an operating system which instructs the network adapter to perform network communications. As can be seen in Figure 2, the device that is responsible for accelerating computation is, in this case, a GPU. GPUs are currently some of the most promising devices to speed up highly parallel and computationally intensive operations. However, in principle any other device suitable for computationally intensive tasks could be used as an alternative.

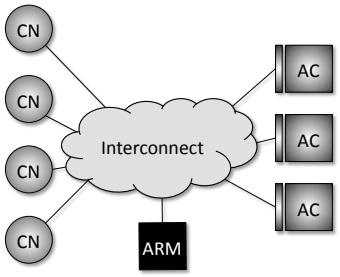


Figure 1. Dynamic accelerator-cluster architecture with compute nodes (CN), accelerators (AC), and accelerator resource manager (ARM).

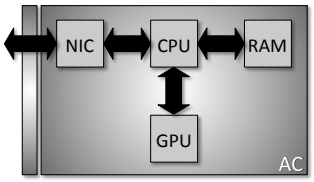
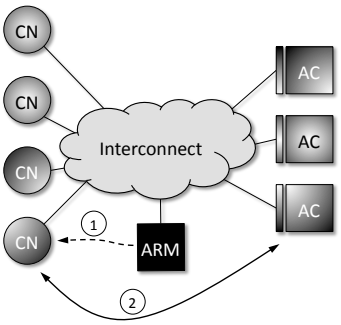


Figure 2. Schematic view of an accelerator (AC) with CPU, main memory (RAM), network adapter (NIC), and GPU.

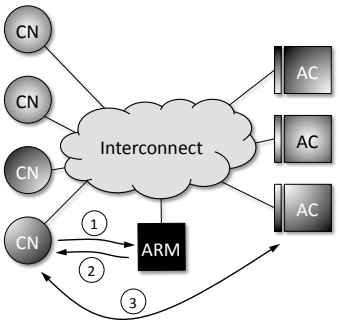
2) Accelerator Resource Manager: The accelerator resource manager is similar to the resource manager in a cluster environment in that it maintains information on which accelerators are available or in use and assigns them to compute nodes upon request. In particular, after assigning an accelerator to a compute node, the corresponding compute node can transparently access its accelerator through a handle as if it was locally attached. The accelerator resource manager supports both a static as well as a dynamic accelerator assignment strategy. In the static case, accelerators are assigned to compute nodes at job start and the assignment (handle) remains active for the duration of the job. With the dynamic assignment strategy, in contrast, accelerators are assigned to compute nodes at runtime. This is done in two steps. First, a compute node requests accelerators from the accelerator resource manager. In response, the accelerator resource manager assigns the corresponding number of accelerators to the compute node, providing one handle per accelerator. Similar to the static assignment, the dynamically assigned accelerators are released once the compute job is finished.

C. Execution Model

The execution model consists of three basic steps: (i) accelerator allocation, (ii) accelerator usage, and (iii) accelerator deallocation. Lacking private accelerators, compute nodes have to allocate accelerators before computations can be offloaded onto them. As mentioned above, the allocation and thus the assignment of accelerators to compute nodes can be done statically (before job start) or dynamically (at runtime). In both cases, the accelerator resource manager provides each process running on a compute node with



(a) Static assignment.



(b) Dynamic assignment.

Figure 3. Static (a) and dynamic (b) accelerator assignment. Different shadings denote different jobs. Dashed lines denote communication before job start, whereas solid lines denote communication at runtime.

distinct handles, with each handle representing exactly one exclusive accelerator. This guarantees that different processes do not interfere with each other. After the allocation, each compute-node process can use an API for transparently offloading computations onto its accelerators over the cluster interconnection network. This API extends familiar APIs such as CUDA or OpenCL. For example, direct accelerator-to-accelerator data transfers over the network are currently not supported with CUDA (CUDA Toolkit 4.2) or OpenCL (OpenCL 1.2), however, in our scheme accelerators can efficiently exchange data without involving their associated compute nodes. As soon as the compute job is completed, the accelerators are automatically released and made available to other jobs by informing the accelerator resource manager. In the dynamic assignment scenario, compute nodes may also release their accelerators by contacting the accelerator resource manager before the compute job is finished. Note that for the dynamic (de)allocation, compute-node processes use an extra resource management API complementing the computation API.

Using the allocation and subsequent usage of one accelerator as an example, Figure 3 illustrates the static and the dynamic assignment strategy. Note that compute nodes and accelerators from the same job are shaded in a uniform way. Furthermore, dashed lines represent communication before job start, whereas solid lines represent communication at runtime. In addition, circled values indicate the sequence of

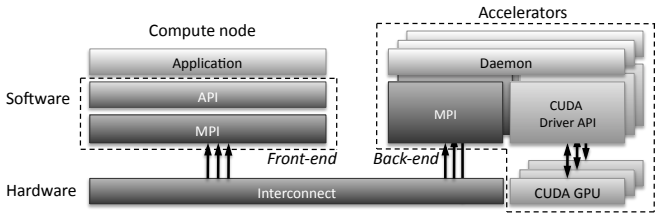


Figure 4. Dynamic accelerator-cluster software architecture with CUDA back-end.

individual communication steps. For the static accelerator-to-compute node assignment shown in Figure 3(a), the accelerator resource manager assigns one accelerator to the compute node before job start (step 1). The result of this assignment is a handle delivered to the compute node, which is eventually used to access the accelerator by means of the computation API (step 2). For the dynamic accelerator-to-compute node assignment shown in Figure 3(b), the compute node uses the resource management API to request accelerators from the accelerator resource manager at runtime (step 1). The accelerator resource manager responds to this request by assigning one accelerator to the compute node (step 2). Finally, the compute node controls the accelerator with the help of the computation API, using the corresponding accelerator handle as a parameter (step 3).

IV. Implementation

As it is the most essential question for the viability of our approach, this article focuses on its performance characteristics in comparison to classic static architectures. In the remainder, we therefore concentrate on the implementation of core communication mechanisms and their evaluation, rather than on advanced features such as the dynamic accelerator assignment strategy (Figure 3(b)). A detailed treatment of the latter will be left to future publications.

Our implementation allows computations to be offloaded onto accelerators over the network. In the current version, computations are CUDA kernels, requiring that each accelerator is equipped with a CUDA-enabled GPU. The computation API of our prototype provides basic functionality for (i) allocating memory on accelerators, (ii) copying data to or from accelerators, and (iii) launching compute kernels on accelerators.

The software stack (Figure 4) of our dynamic accelerator-cluster architecture consists of a front-end on every compute node and a back-end on every accelerator. The front-end translates API calls into requests which are redirected to the back-end, where a daemon receives those requests and executes them on the CUDA-enabled GPU using the CUDA driver API. The front-end talks to the back-end using a handle that uniquely identifies an assigned accelerator, enabling transparent communication between the compute node and the accelerator. Figure 4 illustrates this software stack, which is extensible to any accelerator programming interface and

Listing 1. Example CUDA program.

```
void main(int argc, char **argv) {
 ...
 /* Allocate memory on device */
 cudaMalloc(cudaMalloc_args);

 /* Transfer memory to device */
 cudaMemcpy(cudaMemcpy_args);

 /* Execute kernel */
 k_name<<<dimGrid,dimBlock>>>(k_args);

 /* Transfer memory to host */
 cudaMemcpy(cudaMemcpy_args);

 /* Free memory on device */
 cudaFree(cudaFree_args);
 ...
}
```

Listing 2. Example program on our dynamic architecture.

```
void main(int argc, char **argv) {
 ...
 /* Allocate memory on device */
 acMemAlloc(cudaMalloc_args,ac_handle);

 /* Transfer memory to device */
 acMemCpy(cudaMemcpy_args,ac_handle);

 /* Execute kernel */
 acKernelCreate(k_name,ac_handle);
 acKernelSetArgs(k_args);
 acKernelRun(k_name,dimGrid,dimBlock);

 /* Transfer memory to host */
 acMemCpy(cudaMemcpy_args,ac_handle);

 /* Free memory on device */
 acMemFree(cudaFree_args,ac_handle);
 ...
}
```

therefore not restricted to CUDA by design.

Different from rCUDA [2], the actual communication between compute nodes and accelerators is accomplished through a distinct communication protocol based on MPI wherein each request involves two MPI messages. First, the front-end sends a request message to the back-end. Second, the back-end sends the results (e.g., error code or data) back to the front-end. As an example, we consider a copy operation from host to device. Here, an MPI message carrying the data is sent to the main memory of the accelerator node. This data is then copied from main memory to GPU memory. Finally, the error code of the CUDA copy operation is sent back to the front-end enveloped in an MPI message. Using MPI as the underlying communication substrate offers the following advantages:

- *Availability:* Given that MPI is the de facto standard for communication in cluster systems, MPI can be considered available in those environments our dynamic accelerator-cluster architecture has been designed for.
- *Performance:* MPI allows for efficient inter-process communication over cluster interconnects. In addition, any application using MPI can immediately, that is without modifications, benefit from a new high-speed interconnect as soon as an MPI library supports this interconnect.
- *Portability:* MPI specifies an API which hides the hardware details of cluster systems.

However, to enable MPI communication between a compute node and its accelerator, the process on the compute node and the daemon running on the accelerator have to reside in the same MPI communicator. The creation of this communicator involves the accelerator resource manager.

It is evident that the additional copy operation from the accelerator's main memory to the GPU memory causes more overhead than a direct data copy to a classic PCIe-attached GPU. To address this issue, our memory copy operations—both from host to device and vice versa—leverage NVIDIA's GPUDirect v1. This technology allows network adapters, such as Infiniband that support memory registration including the pinning of memory pages, to share memory-locked pages with NVIDIA GPUs. Now, the same memory buffer can be used for communication with the network on the one hand and with the GPU via direct memory access (DMA) on the other. We exploited this to implement the memory copy in a pipeline fashion where the payload is split into blocks. For example, while some blocks are still being received into the main memory of the accelerator node, others, which are already available there, are already being copied further to GPU memory. Depending on the payload size, this overlap helps minimize if not eliminate the extra time needed to copy the data to GPU memory. As shown in the following section, memory copy operations can now achieve bandwidth results similar to MPI data transfers of the same size.

To illustrate the usage of our API and to compare it with the CUDA API, Listing 1 shows the execution of a kernel using the CUDA API, while Listing 2 shows the same scenario on our dynamic accelerator-cluster architecture. In both cases, the kernel is executed after allocating memory on the device and transferring data from the host to the device. Upon completion of the kernel, results are transferred back from the device to the host, where the device memory is subsequently released. As can be easily seen, our memory allocation/free and transfer operation calls accept the same arguments as their corresponding CUDA API counterparts except for the additional handle used to identify the accelerator. Apparently, the kernel execution through our API includes three steps: (i) the kernel is created before being executed on the accelerator identified by *ac_handle*, (ii) the necessary kernel arguments are set individually, and (iii) the kernel is executed with the specified configuration. Compared to a pure CUDA program, we can therefore argue, that the additional programming complexity of our API is negligible.

V. Experimental Results

In this section, we compare our dynamic architecture with network-attached accelerators quantitatively to a static architecture with node-attached accelerators. In particular, we evaluate the performance impact when a compute node's local GPU is replaced by one or more network-attached GPUs. We start with latency and bandwidth considerations of memory-copy operations between compute nodes and remote accelerators. After that, we show how the availability of multiple network-attached GPUs assigned to a node can provide speedups to two linear-algebra kernels that are impossible to achieve with only a single node-attached GPU. Finally, we demonstrate that the performance of MP2C [9], a molecular-dynamics application designed for a classic GPU cluster with one GPU directly attached to each node, is not significantly harmed. As our architecture is essentially a software architecture, we emulate it using the hardware of a typical static GPU cluster available to our team. The testbed used for this study consists of 4 nodes with 2 Intel Xeon X5670 processors at 2.93 GHz and with 48 GiB RAM each. In addition, all the 4 nodes house one NVIDIA Tesla C1060 GPU (CUDA driver version 270.41.19). The nodes are connected via QDR Infiniband. All nodes run GNU/Linux 2.6.18 (RHEL 5.5). As MPI implementation, we chose Open MPI 1.4.3. Whenever a node was used as compute node in our dynamic architecture, its local GPU was ignored.

We concede that this system setup, in which the "accelerator nodes" are equipped with the same powerful CPU and the same large amount of RAM as the compute nodes, does not exactly correspond to the more cost-efficient solution promoted in Section III. Nevertheless, given the low memory requirements of the pipeline protocol for data transfers and the fact that the CPU is only used to trigger network and GPU operations, we expect that both the performance of the CPU and the amount of RAM collocated with the accelerator could be reduced without noticeable effect on our performance results.

A. Bandwidth

Among the computation API operations described in Section IV, the memory-copy operation between a compute node and the remote accelerator assigned to it is the one that is most sensitive to communication performance. Since those data transfers are usually large in size, typically in the order of several megabytes, the additional MPI over Infiniband latency of roughly two μs is negligible. We therefore focus on the bandwidth of acMemCpy() between a compute node and a remote accelerator. As a benchmark, we selected the bandwidthTest from the CUDA SDK 3.2, which was ported to the dynamic cluster architecture. Since we use MPI as communication layer in our protocol stack, the upper bandwidth limit is set by MPI and in our case by Open MPI over Infiniband. To compare the efficiency of our communication protocol with this upper bound, we measured the pure MPI bandwidth using the PingPong benchmark of the Intel MPI Benchmarks (IMB). The results in Figure 5 show that transmitting a 64 MiB message with MPI on our system reaches a peak bandwidth of about 2660 MiB/s. In addition to this graph, Figure 5 depicts the host-to-device bandwidth measurements for five different implementations of acMemCpy(). Among those, the simplest one is called *naive*. In the naive implementation, all data is first received via a blocking MPI receive call before it is eventually copied to the GPU target buffer. This naive approach requires an MPI receive buffer of the size of the complete data to be copied, which increases the necessary main memory size of the accelerator node. A more efficient approach is the pipeline protocol described in the previous section. Since this protocol does not store the whole message in main memory but only a small fraction of its blocks, the main

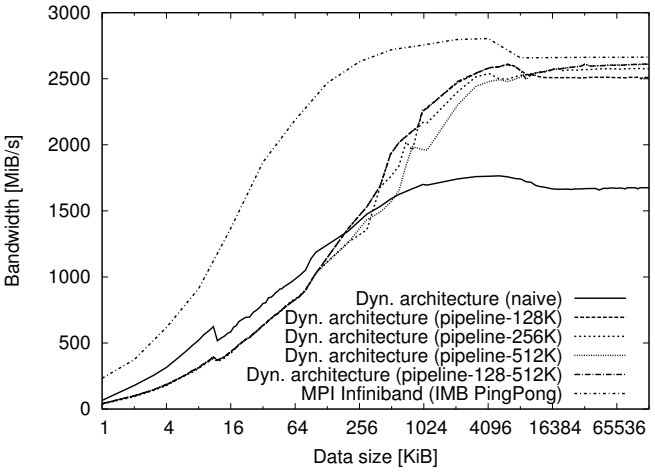


Figure 5. Host-to-device bandwidth for the pipeline protocol with different block sizes and GPUDirect.

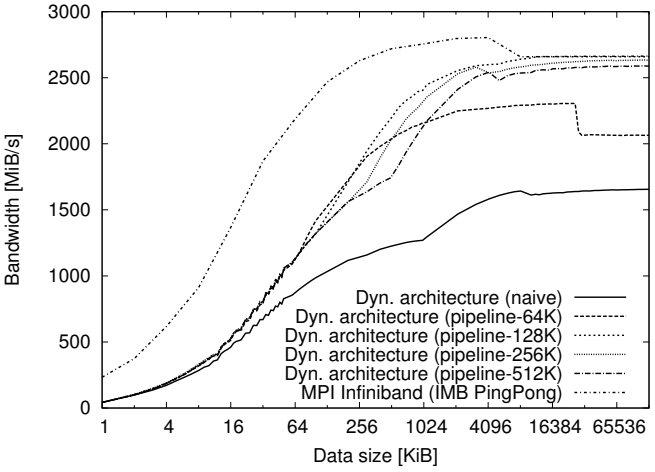


Figure 6. Device-to-host bandwidth for the pipeline protocol with different block sizes and GPUDirect.

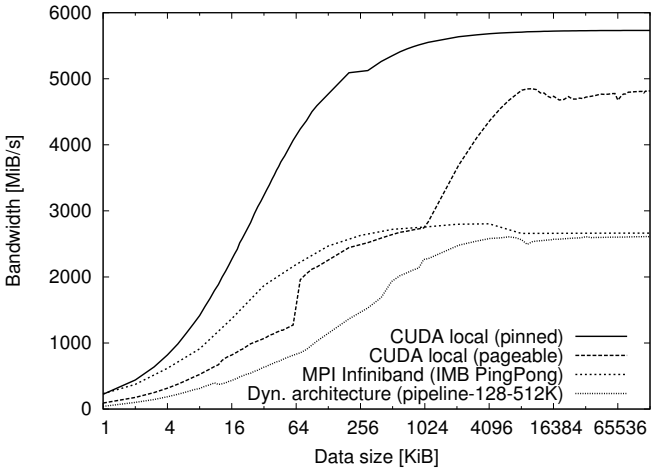


Figure 7. Host-to-device bandwidth comparison between node-attached and network-attached GPU.

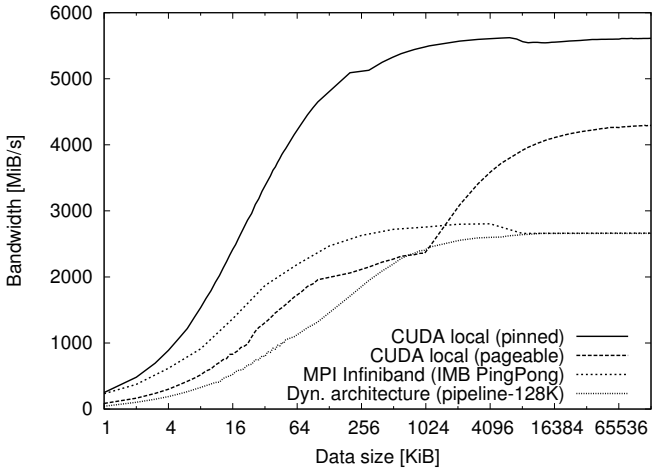


Figure 8. Device-to-host bandwidth comparison between node-attached and network-attached GPU.

memory requirements are independent of the actual message size. Moreover, as illustrated in Figure 5, for large messages all pipeline implementations offer superior bandwidth in comparison to the naive approach. A closer look at the figure also reveals that the performance of the pipeline protocol depends on the size of the blocks into which the message payload is split. Although a 128 KiB block size yields better results than larger block sizes for messages ranging from 500 KiB to 8 MiB, increasing the block size improves the peak bandwidth for messages larger than 8 MiB.

The reason for this is twofold. Compared to large messages, small messages seem to exhibit a higher overlap between network transfers and DMA copy operations from host to device memory. However, for large messages, the overhead caused by posting many small MPI send and host-to-device DMA memory copy operations becomes too large. A simple solution is to adjust pipeline block sizes depending on the message size. In our testbed, the best results were achieved when using 128 KiB sized blocks for messages smaller than 9 MiB and 512 KiB blocks for larger messages. Of course, these parameters are highly system dependent, but tuning them has to be done only once. Afterwards, every user can benefit from better performance. Such initial optimizations are common practice for communication libraries, with MPI probably being the most prominent example.

Similar to Figure 5, Figure 6 depicts the bandwidth results for device-to-host data transfers. As with host-to-device transfers, the pipeline implementation performs better than the naive approach for large messages. However, for the device-to-host copy operations, a single block size of 128 KiB provides the best results. Again, considering our upper bandwidth limit which is determined by MPI, the typical message sizes used in host-to-device and device-to-host data transfers achieve close to full MPI bandwidth performance.

Given our pipeline bandwidth results, the question is now how these compare to node-local CUDA results obtained with cudaMemcpy(). Figure 7 illustrates measurements in host-to-device direction. As can be seen, the node-local GPU results are provided for two cases: pageable memory and pinned memory. Note that the transfers for pageable memory are performed by the CPU through programmed I/O (PIO), whereas the pinned-memory transfers are done by the GPU through direct memory access (DMA) operations. For a 64 MiB payload, the node-local GPU achieved a peak bandwidth of about 5700 MiB/s and about 4700 MiB/s with DMA and PIO, respectively. When we compare this to our architecture with approximately 2600 MiB/s, we notice a clear bandwidth advantage of node-local GPUs. A similar picture is shown for device-to-host memory copy operations in Figure 8. Nevertheless, as we will show farther below, the impact on the overall performance of applications can be quite small.

B. Linear-Algebra Kernels

The advantage of our architecture is that the number of accelerators per compute node is not hardwired. Thus, compute nodes can be assigned more or less accelerators—matching the computational demand much more precisely. We demonstrate the benefits of this feature using two kernels from the MAGMA library, which offers dense linear algebra routines for heterogeneous/hybrid architectures [10]. To enable seamless porting of LAPACK-dependent software components to MAGMA, the design of MAGMA closely follows that of LAPACK. This applies to functionality, data storage, and the interface. MAGMA version 1.1 provides support for distributing computations among multiple GPUs. The following two multi-GPU routines were considered:

- *magma_dgeqrf2_mgpu()*: Computes a QR factorization of a real M-by-N matrix.
- *magma_dpotrf_mgpu()*: Computes the Cholesky factorization of a real symmetric positive definite matrix.

We ported these two routines to our architecture by substituting our remote API calls for the CUDA calls originally used in the code. We conducted our measurements with MAGMA 1.1-RC1 and used the provided testing routines as benchmark. We executed each routine on a single compute node, either using the local GPU or up to three network-attached GPUs. Figure 9 and Figure 10 present the results for the QR and the Cholesky factorization, respectively. It can be seen that both routines suffer slightly from the bandwidth penalty. Comparing one local GPU with one network-attached GPU, QR is shown to be more sensitive to communication bandwidth than Cholesky. Furthermore, it can be observed that, depending on the problem size, the flexibility of our architecture to assign as many accelerators to compute nodes as needed, can be exploited to deliver results faster than would be possible with just a single node-attached accelerator. For example, with problem size 10240 and three network-attached GPUs, the QR factorization achieved a speedup of about 2.2 in comparison to one local GPU. Running this kernel equally fast on a static cluster with one GPU per node would require using multiple compute nodes in parallel, presumably via MPI. However, this extra parallelism comes at a significant cost in terms of development effort.

Of course, being detached from compute nodes, dynamic accelerators have to be allocated separately before they can be used. In a production environment, a user would therefore specify the number of accelerators requested per node in his or her batch script. The job would start once the requested number of compute and accelerator nodes becomes available. This corresponds to the static assignment strategy described in Section III. However, since no application uses more accelerators than it actually needs, the availability of accelerators is maximized. Thus, the flexibility of our architecture has also economic advantages from the perspective of the system provider.

C. MP2C

In the remainder of this section, we investigate how the bandwidth limitation of using remote GPUs impacts the performance of a real-world application that is designed for only a single GPU per node and, thus, can hardly benefit from the dynamic nature of our scheme. The molecular-dynamics simulation MP2C is a highly scalable multi-scale code, which couples a mesoscopic fluid method based on multi-particle collision dynamics with molecular dynamics. MP2C leverages message-passing parallelism through MPI and geometrical domain decomposition. It consists of a molecular-dynamics part and a multi-particle collision dynamics part, which implements the stochastic rotation dynamics method (SRD) [11], a highly local algorithm, in CUDA. For the measurements with our architecture, the CUDA calls were replaced by the corresponding calls to our API. The CUDA measurements were performed with two processes running

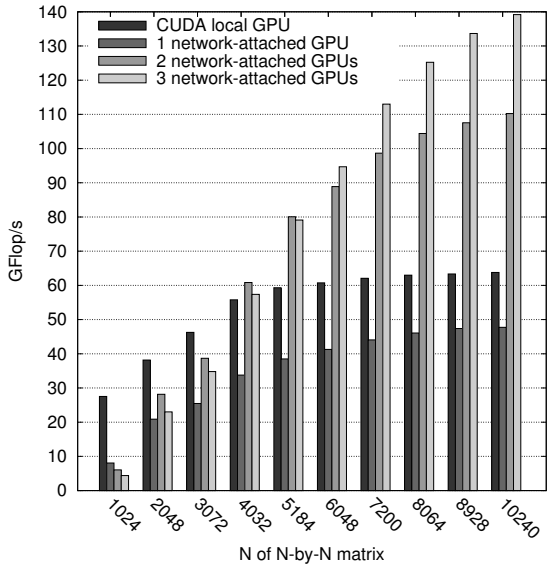


Figure 9. MAGMA QR factorization comparing node-local GPU with network-attached GPUs.

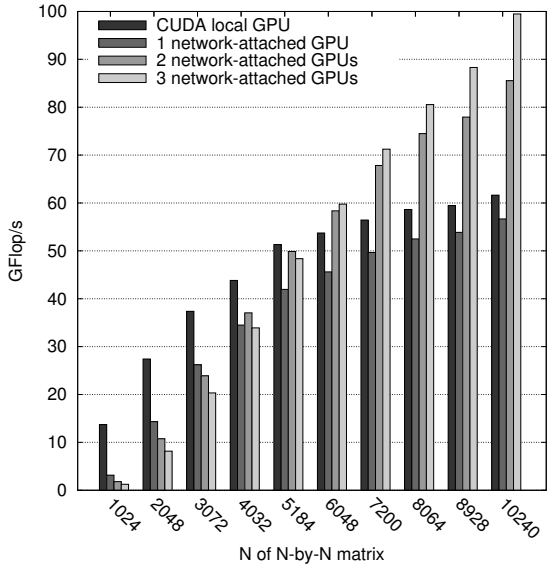


Figure 10. MAGMA Cholesky factorization comparing node-local GPU with network-attached GPUs.

on separate nodes and using their local GPUs. Instead of using local GPUs, in measurements emulating our dynamic architecture each process used its own dedicated remote GPU. We ran MP2C with three different numbers of particles (5120000, 7290000, 10000000). The number of particles per collision cell is 10 and the SRD method is executed in every 5-th step of 300 steps in total. Figure 11 illustrates that for all three different input configurations, our dynamic cluster architecture prototype prolongs execution by at most 4%. This shows that while providing clear benefits to some applications, our architecture does not necessarily harm the performance of others whose GPU demand can be satisfied well in more traditional environments.

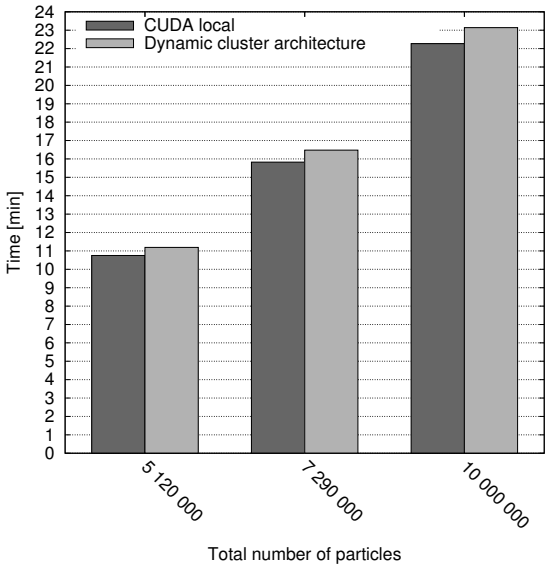


Figure 11. MP2C real-world application.

VI. Conclusion and Outlook

In current accelerator-enhanced clusters, graphics devices are directly attached and, therefore, exclusively dedicated to a distinct general-purpose compute node. However, this static $N-\text{to}-1$ mapping between accelerators and compute nodes may lead to underutilization and a mismatch between available hardware and application requirements. In this paper, we have proposed a dynamic accelerator-cluster architecture that overcomes these limitations by supporting the dynamic allocation of accelerators over a network. In our scheme, an accelerator is attached to a high-speed interconnection network and can be assigned to each compute node in the cluster on demand. Although initially implemented for CUDA, the architecture is extensible to any accelerator programming interface. For instance, our generic software stack would also easily allow Intel's emerging Many Integrated Core (MIC) architecture to be supported.

Our experimental evaluation indicates that, depending on the input problem size, some applications can exploit the increased flexibility of our approach to achieve additional speedup per node. This lifts the threshold beyond which explicit MPI parallelism across multiple compute nodes becomes necessary, avoiding premature and expensive hybridization. However, this flexibility also comes along with an extra network-transfer overhead, whose performance impact is application dependent. To account for both the higher flexibility and the additional network-transfer overhead, cluster systems could, on the one hand, offer a pool of compute nodes with local GPUs for applications highly sensitive to communication performance and, on the other hand, provide an additional pool of network-attached accelerators available to all compute nodes. In the near future, we plan to further investigate the dynamic accelerator assignment strategy and how it can help speed up applications with phases of differing computational demand.

Acknowledgment

The authors thankfully acknowledge the technical expertise and assistance provided by the Computational Science Division at the Jülich Supercomputing Centre. We especially would like to express our gratitude to Godehard Sutmann for his generous help.

References

[1] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, "File server scaling with network-attached secure disks," *ACM SIGMETRICS Performance Evaluation Review*, vol. 25, pp. 272–284, June 1997.

[2] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Proc. of the International Conference on High Performance Computing and Simulation (HPCS)*. IEEE, 2010.

[3] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU accelerated high performance computing in virtual machines," in *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2009.

[4] Zillians. (2011) V-GPU. [Online]. Available: http://www.zillians.com/vgpu/

[5] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *Proc. of the International Conference on Cluster Computing (PPAAC Workshop)*. IEEE, Sep. 2010.

[6] C. Müller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl, "A compute unified system architecture for graphics clusters incorporating data locality," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 15, no. 4, pp. 605–617, 2009.

[7] V. Krishnan, "Towards an integrated IO and clustering solution using PCI express," in *Proc. of the International Conference on Cluster Computing*. IEEE, Sep. 2007.

[8] NextIO. (2011) I/O Virtualization. [Online]. Available: http://www.nextio.com/

[9] J. Freche, W. Frings, and G. Sutmann, "High throughput parallel-i/o using sionlib for mesoscopic particle dynamics simulations on massively parallel computers," in *Proc. of Parallel Computing: From Multicores and GPU's to Petascale*, ser. Advances in Parallel Computing, vol. 19. IOS Press, 2010, pp. 371 – 378.

[10] I. C. L. at University of Tennessee. (2011) Magma. [Online]. Available: http://icl.cs.utk.edu/magma/

[11] G. Gompper, T. Ihle, D. Kroll, and R. Winkler, "Multi-particle collision dynamics: A particle-based mesoscale simulation approach to the hydrodynamics of complex fluids," vol. 221, pp. 1–87, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87706-6_1