# dCUDA: Hardware Supported Overlap of Computation and Communication

Tobias Gysi
Department of Computer Science
ETH Zurich
tobias.gysi@inf.ethz.ch

Jeremia Bär
Department of Computer Science
ETH Zurich
jeremia.baer@alumni.ethz.ch

Torsten Hoefler
Department of Computer Science
ETH Zurich
htor@inf.ethz.ch

***Abstract*—Over the last decade, CUDA and the underlying GPU hardware architecture have continuously gained popularity in various high-performance computing application domains such as climate modeling, computational chemistry, or machine learning. Despite this popularity, we lack a single coherent programming model for GPU clusters. We therefore introduce the dCUDA programming model, which implements device-side remote memory access with target notification. To hide instruction pipeline latencies, CUDA programs over-decompose the problem and over-subscribe the device by running many more threads than there are hardware execution units. Whenever a thread stalls, the hardware scheduler immediately proceeds with the execution of another thread ready for execution. This latency hiding technique is key to make best use of the available hardware resources. With dCUDA, we apply latency hiding at cluster scale to automatically overlap computation and communication. Our benchmarks demonstrate perfect overlap for memory bandwidth-bound tasks and good overlap for compute-bound tasks.**

***Index Terms*—Distributed memory, gpu, latency hiding, programming model, remote memory access**

## I. Introduction

Today, we typically target GPU clusters using two programming models that separately deal with inter-node and single-node parallelization. For example, we may use MPI [7] to move data in between nodes and CUDA [19] to implement the on-node computation. MPI provides point-to-point communication and collectives that allow to synchronize concurrent processes executing on different cluster nodes. Using a fork-join model, CUDA allows to offload compute kernels from the host to the massively parallel device. To combine the two programming models, MPI-CUDA programs usually alternate sequentially between on-node kernel invocations and inter-node communication. While being functional, this approach also entails serious disadvantages.

The main disadvantage is that application developers need to know the concepts of both programming models and understand several intricacies to work around their inconsistencies. For example, the MPI software stack in meantime has been adapted to support direct device-to-device [24] data transfers. However, the control path remains on the host which causes frequent host-device synchronizations and redundant data structures on host and device.

On the other hand, the sequential execution of on-node computation and inter-node communication inhibits efficient utilization of the costly compute and network hardware. To mitigate this problem, application developers can implement manual overlap of computation and communication [23], [27]. In particular, there exist various approaches [13], [22] to overlap the communication with the computation on an inner domain that has no inter-node data dependencies. However, these code transformations significantly increase code complexity which results in reduced real-world applicability.

High-performance system design often involves trading off sequential performance against parallel throughput. The architectural difference between host and device processors perfectly showcases the two extremes of this design space. Both architectures have to deal with the latency of hardware components such as memories or floating point units. While the host processor employs latency minimization techniques such as prefetching and out-of-order execution, the device processor employs latency hiding techniques such as over-subscription and hardware threads.

To avoid the complexity of handling two programming models and to apply latency hiding at cluster scale, we introduce the dCUDA (distributed CUDA) programming model. We obtain a single coherent software stack by combining the CUDA programming model with a significant subset of the remote memory access capabilities of MPI [12]. More precisely, a global address space and device-side put and get operations enable transparent remote memory access using the high-speed network of the cluster. We thereby make use of over-decomposition to over-subscribe the hardware with spare parallelism that enables automatic overlap of remote memory accesses with concurrent computation. To synchronize the program execution, we additionally provide notified remote memory access operations [3] that after completion notify the target via notification queue.

We evaluate the dCUDA programming model using a stencil code, a particle simulation, and an implementation of sparse matrix-vector multiplication. To compare performance and usability, we implement dCUDA and MPI-CUDA versions of these mini-applications. Two out of three mini-applications show excellent automatic overlap of communication and computation. Hence, application developers not only benefit from the convenience of device-side remote memory

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00 

access. More importantly, dCUDA enables automatic overlap of computation and communication without costly, manual code transformations. As dCUDA programs are less network latency sensitive, our development might even motivate more throughput oriented network designs. In brief, we make the following contributions:

- We implement the first device-side communication library that provides MPI like remote memory access operations and target notification for GPU clusters.
- We design the first GPU cluster programming model that makes use of over-subscription and hardware threads to automatically overlap inter-node communication with on-node computation.

## II. Programming Model

The CUDA programming model and the underlying hardware architecture have proven excellent efficiency for parallel compute tasks. To achieve high performance, CUDA programs offload the computation to an accelerator device with many throughput optimized compute cores that are over-subscribed with many more threads than they have execution units. To overlap instruction pipeline latencies, in every clock cycle the compute cores try to select among all threads in flight some that are ready for execution. To implement context switches on a clock-by-clock basis, the compute cores split the register file and the scratchpad memory among the threads in flight. Hence, the register and scratchpad utilization of a code effectively limit the number of threads in flight. However, having enough parallel work is of key importance to fully overlap the instruction pipeline latencies. Little's law [1] states that this minimum required amount of parallel work corresponds to the product of bandwidth and latency. For example, we need 200kB of data on-the-fly to fully utilize a device memory with 200GB/s bandwidth and 1µs latency. Thereby, 200kB translate to roughly 12,000 threads in flight each of them accessing two double precision floating point values at once. We show in Section IV that the network of our test system has 6GB/s bandwidth and 19µs latency. We therefore need 114kB of data or roughly 7,000 threads in flight to fully utilize the network. Based on the observation that typical CUDA programs make efficient use of the memory bandwidth, we conclude there should be enough parallelism to overlap network operations. Consequently, we suggest to use hardware supported overlap of computation and communication to program distributed memory systems.

### A. Distributed Memory

One main challenge of distributed memory programming is the decomposition and distribution of program data to the different memories of the machine. Currently, most distributed memory programming models rely on manual domain decomposition and data synchronization since handling distributed memory automatically is hard.

Today, MPI is the most widely used distributed memory programming model in high-performance computing. Many codes thereby rely on two sided communication that simultaneously involves sender and receiver. Unfortunately, this combination of data movement and synchronization is a bad fit for extending the CUDA programming model. On the one hand, CUDA programs typically perform many data movements in the form of device memory accesses before synchronizing the execution using global barriers. On the other hand, CUDA programs over-subscribe the device by running many more threads than there are hardware execution units. As sender and receiver might not be active at the same time, two sided communication is hardly practical. To avoid active target synchronization, MPI alternatively provides one sided put and get operations that implement remote memory access [12] using a mapping of the distributed memory to a global address space. We believe that remote memory access programming is a natural extension of the CUDA programming model.

Finally, programming large distributed memory machines requires more sophisticated synchronization mechanisms than the barriers implemented by CUDA. We propose a notification based synchronization infrastructure [3] that after completion of the put and get operations enqueues notifications on the target. To synchronize, the target waits for incoming notifications enqueued by concurrent remote memory accesses. This queue based system enables to build linearizable semantics.

### B. Combining MPI & CUDA

CUDA programs structure the computation using kernels, which embody phases of concurrent execution followed by result communication and synchronization. More precisely, kernels write to memory with relaxed consistency and only after an implicit barrier synchronization at the end of the kernel execution the results become visible to the entire device. To expose parallelism, kernels use a set of independent thread groups called blocks that provide memory consistency and synchronization among the threads of the block. The blocks are scheduled to the different compute cores of the device. Once a block is running its execution cannot be interrupted as todays devices do not support preemption [30]. While each compute core keeps as many blocks in flight as possible, the number of concurrent blocks is constraint by hardware limits such as register file and scratchpad memory capacity. Consequently, the block execution may be partly sequential and synchronizing two blocks might be impossible as one runs after the other.

MPI programs expose parallelism using multiple processes called ranks that may execute on the different nodes of a cluster. Similar to CUDA, we structure the computation in phases of concurrent execution followed by result communication and synchronization. In contrast to CUDA, we write the results to a distributed memory and we use either point-to-point communication or remote memory accesses to move data between the ranks.

Todays MPI-CUDA programs typically assign one rank to every device and whenever necessary insert communication in between kernel invocations. However, stacking the communication and synchronization mechanisms of two programming models makes the code unnecessarily complex. Therefore, we

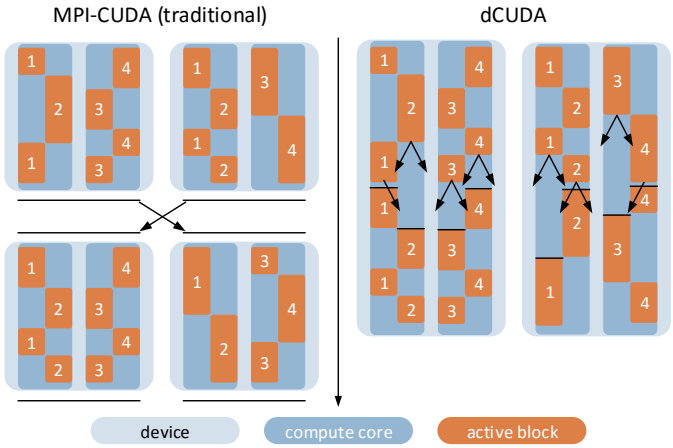


Fig. 1: Block scheduling for MPI-CUDA and dCUDA.

suggest to combine the two programming models into a single coherent software stack.

dCUDA programs implement the application logic using a single CUDA kernel that performs explicit data exchange during execution. To enable synchronization, we limit the over-subscription to the maximal number of concurrent hardware threads supported by the device. To move data between blocks no matter if they run on the same or on remote devices, we use device-side remote memory access operations. We identify each block with a unique rank identifier that allows to address data on the entire cluster. We map MPI ranks to CUDA blocks, as they represent the most coarse grained execution unit that benefits from automatic latency overlap due to hardware threading. Hereafter, we use the terms rank and block interchangeably. To synchronize the rank execution, we implement remote memory access operations with target notification. An additional wait method finally allows to synchronize the target execution with incoming notifications.

Figure 1 compares the execution of an MPI-CUDA program to its dCUDA counterpart. We illustrate the program execution on two dual-core devices each of them over-subscribed with two blocks per core. We indicate communication using black arrows and synchronization using black lines. Both programs implement sequential compute and communication phases. While the dCUDA program uses over-subscription to automatically overlap the communication and compute phases of competing blocks, the MPI-CUDA program leaves this optimization potential unused.

### C. Example

Figure 2 shows an example program that uses dCUDA to implement a two-dimensional stencil computation. Using pointers adjusted to rank local memory, the program reads from an ”in” array and writes to an ”out” array. To distribute the work, the program performs a one-dimensional domain decomposition in the j-dimension. To satisfy all data dependencies, in every iteration the program exchanges one halo line with the left and right neighbor rank. For illustration purposes,

```
1  __shared__ dcuda_context ctx;
2  dcuda_init(param, ctx);
3  dcuda_comm_size(ctx, DCUDA_COMM_WORLD, &size);
4  dcuda_comm_rank(ctx, DCUDA_COMM_WORLD, &rank);
5
6  dcuda_win win, wout;
7  dcuda_win_create(ctx, DCUDA_COMM_WORLD,
8     &in[0], len + 2 * jstride, &win);
9  dcuda_win_create(ctx, DCUDA_COMM_WORLD,
10    &out[0], len + 2 * jstride, &wout);
11
12 bool lsend = rank - 1 >= 0;
13 bool rsend = rank + 1 < size;
14
15 int from = threadIdx.x + jstride;
16 int to = from + len;
17
18 for (int i = 0; i < steps; ++i) {
19   for (int idx = from; idx < to; idx += jstride)
20     out[idx] = -4.0 * in[idx] +
21       in[idx + 1] + in[idx - 1] +
22       in[idx + jstride] + in[idx - jstride];
23
24   if (lsend)
25     dcuda_put_notify(ctx, wout, rank - 1,
26       len + jstride, jstride, &out[jstride], tag);
27   if (rsend)
28     dcuda_put_notify(ctx, wout, rank + 1,
29       0, jstride, &out[len], tag);
30
31   dcuda_wait_notifications(ctx, wout,
32     DCUDA_ANY_SOURCE, tag, lsend + rsend);
33
34   swap(in, out); swap(win, wout);
35 }
36
37 dcuda_win_free(ctx, win);
38 dcuda_win_free(ctx, wout);
39 dcuda_finish(ctx);
```

Fig. 2: Stencil program with halo exchange communication.

the program listing highlights all methods and types implemented by the dCUDA framework. The calling conventions require that all threads of a block call the framework methods collectively with the same parameter values. To convert the example into a working program, we need additional boilerplate initialization logic that, among other things, performs the input/output and the domain decomposition.

On line 2, we initialize the context object using the ”param” kernel parameter that contains framework information such as the notification queue address. The context object stores the shared state used by the framework methods.

On lines 3–4, we get size and identifier of the rank with respect to the world communicator. A communicator corresponds to a set of ranks that participate in the computation. Thereby, each rank has a unique identifier with respect to this communicator. We currently provide two predefined communicators that either represent all ranks of the cluster called ”world communicator” or all ranks of the device called ”device communicator”.

On lines 6–10, we create two windows that provide remote memory access to the ”in” and ”out” arrays. When creating a window all participating ranks register their own local memory

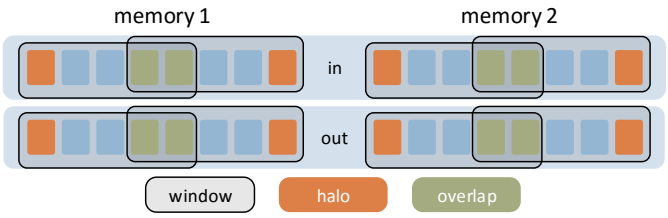


Fig. 3: Overlapping windows of four ranks in two memories.

range with the window. The individual window sizes may differ and windows of shared memory ranks might overlap. We use windows to define a global address space where rank, window, offset tuples denote global distributed memory addresses. Figure 3 illustrates the overlapping windows of the example program. Each cell represents the memory of one j-position that stores "jstride" values in the i-dimension. Colors mark cells that belong to the domain boundaries of the rank. More precisely, the windows of shared memory ranks overlap and the windows of distributed memory ranks allocate additional halo cells that duplicate the domain boundaries.

On lines 24–30, we move the domain boundaries of the "out" array to the windows of the neighbor ranks. We address the remote memory using the window, rank, and offset parameters. Once the data transfer completes, the put operation additionally places a notification in the notification queue of the target rank. We can mark the notification with a tag that, in case of more complex communication patterns, allows disentangling different notification sources.

On lines 31–32, we wait until the notification of the neighboring ranks arrive in the notification queue. Our program waits for zero, one, or two notifications depending on the values of the lsend and rsend flags. We consider only notifications with specific window, rank, and tag values. To match multiple notifications at once, we can optionally use wildcard values that allow to match notifications with any window, rank, or tag value.

On lines 37–39, we free the window objects to cleanup after the program execution. Overall, our implementation closely follows the MPI remote memory access specification [12]. On top of the functionality demonstrated by the example, we implement the window flush operation that allows to wait until all pending window operations are done. Furthermore, we cannot only put data to remote windows but also get data from remote windows. Finally, the barrier collective allows to globally synchronize the rank execution.

### D. Discussion

Compared to MPI-CUDA, dCUDA slightly simplifies the code by moving the communication control to the device. For example, we have direct access to the size information of dynamic data structures and there is less need for separate pack kernels that bundle data for the communication phase. While the distributed memory handling causes most of the code complexity for both programming models, with dCUDA we remove one synchronization layer and implement everything

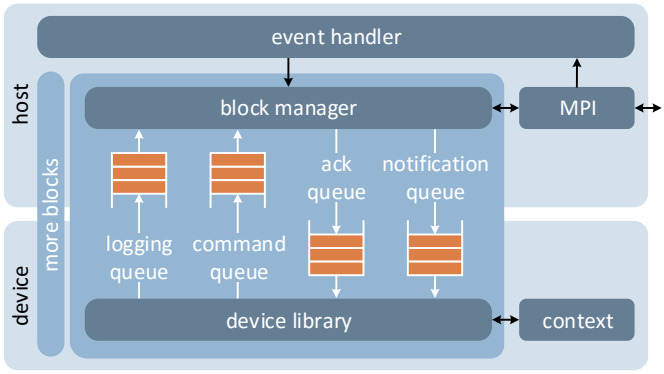


Fig. 4: Architecture overview of the dCUDA runtime system.

with a distributed memory view. We may thereby generate redundant put and get operations in shared memory, but our runtime can optimize them out.

## III. Implementation

Moving the MPI functionality to the device-side raises multiple challenging implementation questions. To our knowledge so far there is no device-side MPI library, which might be partly attributed to the fact that calling MPI from the kernel conflicts with multiple CUDA mantras. On the one hand, the weak consistency memory model prevents shared memory communication during kernel execution. On the other hand, the missing block scheduling guarantees complicate the block synchronization.

### A. Architecture Overview

Our research prototype consists of a device-side library that implements the actual programming interface and a host-side runtime system that controls the communication. More precisely, we run one library instance per rank and one runtime system instance per device. Connected via MPI, the runtime system instances control data movement and synchronization of any two ranks in the system. However, the data movement itself either takes place locally on the device or using direct device-to-device communication [24].

While a design without host involvement seems desirable, existing attempts to control the network interface card directly from the device are not promising [21] in terms of performance and system complexity. Furthermore, the host is a good fit for the synchronization required to order incoming notifications from different source ranks. To avoid device-side synchronization, we go even one step further and loop device local notifications through the host as well.

Moving data between ranks running on the same device requires memory consistency. In CUDA atomics are the only coherent memory operations at device-level. However, we did not encounter memory inconsistencies on Kepler devices with disabled L1 cache (which is the default setting). When polling device memory, we additionally use the volatile keyword to make sure the compiler issues a load instruction for every variable access.

To implement collectives such as barrier synchronization [31], all participating ranks have to be scheduled actively. Otherwise, the collective might deadlock. As discussed in Section II-B, hardware constraints, such as the register file size and the lack of preemption, result in sequential block execution once we exceed the maximal number of concurrent hardware threads. Our implementation therefore limits the number of blocks to the maximum the device can have in flight at once. However, we might still encounter starvation as there are no guarantees regarding the hardware thread scheduling implemented by the compute cores. For example, the compute cores might only run the threads that are waiting for notifications and pause the threads that send notifications.

Figure 4 illustrates the software architecture of the dCUDA runtime system. A host-side event handler starts and controls the execution of the actual compute kernel. To communicate with the blocks of the running kernel, we create separate block manager instances that interact with the device-side library components using queues implemented as circular buffers. The event handler dispatches incoming remote memory access requests to the matching target block manager and continuously invokes the block manager instances to process incoming commands and pending MPI requests. More precisely, using the command queue the device-side library triggers block manager actions such as window creation, notified remote memory access, and barrier synchronization. To guarantee progress using a single worker thread, the block manager implements these actions using non-blocking MPI operations. Once the pending MPI request signals completion, the block manager notifies the device-side library using separate queues to acknowledge commands and to post notifications. An additional logging queue allows to print debug information during kernel execution. The device-side library uses a context object to store shared state such as queue or window information. Most of the times, the device-side library initiates actions on the host and waits for their completion. However, all remote memory accesses to shared memory ranks are directly executed on the device. We thereby perform no copy if source and target address of the remote memory access are identical, which commonly happens for overlapping shared memory windows. Furthermore, the device-side library implements the notification matching.

### B. Communication Control

Figure 5 shows the end-to-end control flow for a distributed memory put operation with target notification. Initially, the origin device-library assembles a tuple containing data pointer, size, target rank, target window and offset, tag, and flush identifier of the transfer and 1) sends this meta information to the associated block manager. Using two non-blocking sends, 2) the origin block manager forwards the meta information to the target event handler and 3) copies the actual data directly from the origin device memory to the target device memory. Once the MPI requests signal that the send buffers are not in use anymore, 4) the origin block manager frees the meta information and updates the flush counter on the device.

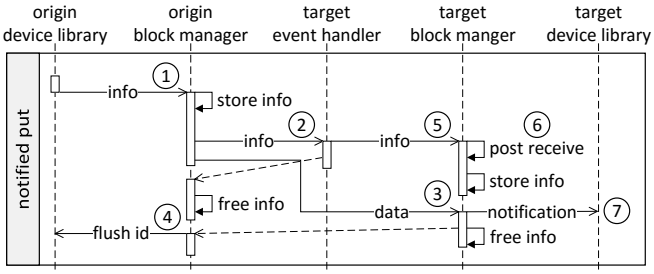


Fig. 5: Sequence diagram of a notified distributed memory put.

Using pre-posted receives, the target event handler waits for meta information arriving from an arbitrary origin rank and 5) immediately forwards the incoming meta information to the associated target block manager. Finally, 6) the target block manager posts a non-blocking receive for the actual data transfer and 7) after completion notifies the target device-side library and frees the meta information.

The control flow for shared memory is simpler. Initially, the origin device-side library performs the actual data transfer. We thereby perform no copy in case source and target pointers are identical. Finally, we notify the target device-side library via the origin block manager.

The device-side library uses a counter to generate unique window identifiers. These counters get out of sync whenever only a subset of the ranks participates in the window creation. The block manager therefore uses a hash map to translate the device-side identifiers to globally valid identifiers. Similarly, we use a counter to generate unique flush identifiers for remote memory access operations. The block manager keeps a history of the processed remote memory access operations and updates the device about the progress using a single variable set to the flush identifier of the last processed remote memory access operation whose predecessors are done as well.

### C. Performance Optimization

As an efficient host-device communication is of key importance for the performance of our runtime system, we spend considerable effort in optimizing queue design and notification matching. Due to the Little's law assumption, we rather focus on throughput than on latency optimizations.

*Memory Mapping:* To move large amounts of data between host and device, the copy methods provided by the CUDA runtime are the method of choice. However, the DMA engine setup causes a considerable startup latency. Alternatively, we can directly access the device memory mapped in the address space of the host memory and vice versa. This approach is a much better fit for the small data transfers prevalent in queue operations. While CUDA out of the box provides support to map host memory in the device address space, we can map device memory in the host address space using an additional kernel module [26].

*Queue Design:* On todays machines the PCI-Express link between host and device is a major communication bottleneck.

We therefore employ a circular buffer based queue design that provides an enqueue operation with an amortized cost of a single PCI-Express transaction. To facilitate efficient polling, we place the circular buffer including its tail pointer in receiver memory. For example, Figure 4 shows that we allocate the notification queue in device memory and the command queue in host memory. To implement the enqueue operation using a single PCI-Express transaction, we embed an additional sequence number with every queue entry. The receiver then determines valid queue entries using the sequence number instead of the head pointer. Furthermore, we use a credit-based system to keep track of the available space. The sender starts with a free counter that is set to the queue size. With every enqueue operation, we decrement the free counter until it is zero. To recompute the available space, we then load the tail pointer from the receiver memory. The number of free counter updates depends on queue size and queue utilization. Overall, every enqueue operation requires one PCI-Express transaction to write the queue entry including its sequence number and an occasional PCI-Express transaction to update the free counter. We thereby assume queue entry accesses using a single vector instruction are atomic. On our test system, we never encountered inconsistencies when limiting the queue entry size to the vector instruction width.

*Notification Matching:* The notification matching is the most complex device-side library component. Two methods allow to wait or test for a given number of incoming notifications. We can thereby filter the notifications depending on window identifier, source rank, and tag [3]. The matching happens in the order of arrival and after completion we remove the matched notifications. To fill potential gaps, we additionally compress the notification queue starting from the tail. Our implementation performs the matching using eight threads that work on separate four byte notification chunks. We read incoming notifications using coalesced reads and once the sequence number matches each thread compares the assigned notification chunk to a thread private query value. We initialize the query value depending on the thread index position with the window identifier, the source rank, the tag, or with a wild card value. To determine if the matching was successful, we reduce the comparison result using shuffle instructions. In case of a mismatch each thread buffers his notification chunk in a stack-allocated array. Otherwise, we increment a counter that keeps track of the successful matches. Finally, we remove the processed notification from the queue and repeat the procedure until we have enough successful matches. Once this is done, we copy the mismatched notifications back from the stack-allocated array to the queue. We thereby assume the number of mismatched notifications is low enough for the stack-allocated array to fit in the L1 cache.

### D. Discussion

To make our programming model production ready, additional modifications may be necessary. For example, we partly rely on undocumented hardware behavior and we could further optimize the performance. To develop a more reliable and efficient implementation, we suggest the following improvements to the CUDA environment.

*Scheduling of Computation & Communication:* Our programming model packs the entire application logic in a single kernel. As discussed in Section III-A, this approach conflicts with the scheduling guarantees and the weak memory consistency model of CUDA. For example, we might encounter starvation because the scheduler does not consider the ranks that are about to send notifications, or we might work with outdated data since there is no clean way to guarantee device-level memory consistency during kernel execution. We suggest an execution model with one master thread per rank that handles the communication using remote memory access and notifications. Similar to the dynamic parallelism feature of CUDA, the master thread additionally launches parallel compute phases in between the communication phases. To guarantee memory consistency, our execution model clears the cache before every compute phase. To prevent starvation, we suggest a yield call that guarantees execution time for all other ranks running on the same processing element. Hence, the master thread can yield the other ranks while waiting for incoming notifications. Currently, the compute phase with maximal register usage limits the available parallelism for the entire application. With the proposed execution model, we can adapt the number of threads for every compute phase and increase the overall resource usage.

*Notification System:* An effective and low overhead notification system is crucial for the functioning of our programming model. Despite our optimization efforts, the current notification matching discussed in Section III-C increases register pressure and code complexity and consequently may impair the application performance. We suggest to at least partly integrate the notification infrastructure with the hardware. On the one hand, the network may send data and notifications using a single transmission. Low level interfaces, such as uGNI [14] or InfiniBand Verbs, already provide the necessary support. On the other hand, the device may provide additional storage and logic for the notification matching or hardware support for on-chip notifications.

*Communication Control:* While we move data directly from device-to-device, we still rely on the host to control the communication. We expect that moving this functionality to the device improves the overall performance of the system. Mellanox and NVIDIA recently announced a technology called GPUDirect Sync [9] that will enable device-side communication control.

## IV. Evaluation

To analyze the performance of our programming model, we implement a set of microbenchmarks that measure latency, bandwidth, and the overlap of computation and communication for compute and memory bound tasks. We additionally compare the performance of mini-applications implemented using both dCUDA and MPI-CUDA.

### A. Experimental Setup & Methodology

We perform all our experiments on the Greina compute cluster at the Swiss National Supercomputing Center CSCS. In total, Greina provides ten Haswell nodes equipped with one Tesla K80 GPU per node and connected via 4x EDR Infiniband. Furthermore, we use version 7.0 of the CUDA toolkit, the CUDA-aware version 1.10.0 of OpenMPI, and the gdrcopy kernel module [26]. We run all experiments using a single GPU per node with default device configuration. In particular, auto boost remains active which makes device-side time measurements unreliable.

To measure the performance of dCUDA and MPI-CUDA codes, we time the kernel invocations on the host-side and collect the maximum execution time found on the different nodes. In contrast, dCUDA programs pack the application code in a single kernel invocation that also contains a fair amount of setup code such as the window creation. To get a fair comparison, we therefore time multiple iterations and subtract the setup time estimated by running zero iterations. Furthermore, we repeat each time measurement multiple times and compute the median and the nonparametric confidence interval. More precisely, we perform 20 independent measurements of 100 and 5,000 iterations for the mini-applications and the microbenchmarks respectively. Our plots visualize the 95% confidence interval using a gray band.

The dCUDA programming model focuses on multi-node performance. To eliminate measurement noise caused by single-node performance variations, we use the same launch configuration for all kernels (208 blocks per device and 128 threads per block), and we limit the register usage to 63 registers per thread which guarantees that all 208 blocks are in flight at once.

### B. Microbenchmarks

To evaluate latency and bandwidth of our implementation, we run a ping-pong benchmark that in every iteration moves a data packet forth and back between two ranks using notified put operations. We either place the two ranks on the same device and communicate via shared memory, or we place the ranks on different devices and communicate via the network. We then derive the latency as half the execution time of a single ping-pong iteration and divide the packet size by the latency to compute the bandwidth. Figure 6 plots the put-bandwidth for shared and distributed memory ranks as function of the packet size. The low put-bandwidth of shared memory ranks can be explained by the fact that a single block cannot saturate the memory interface of the device. However, in real-world scenarios hundreds of blocks are active concurrently resulting in high aggregate bandwidth. For empty data packets, we measure a latency of 7.8µs and 19.3µs for shared and distributed memory respectively. Hence, the latency of a notified put tops the device memory access latency [17] by one order of magnitude. We are aware that these numbers motivate further tuning. However, in the following we demonstrate that the dCUDA programming model is extremely latency agnostic.

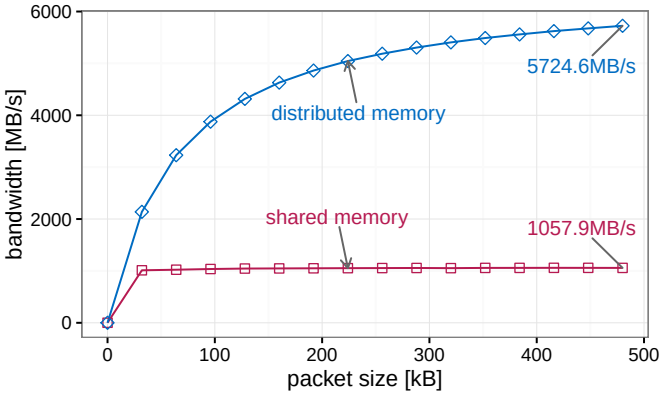


Fig. 6: Put-bandwidth of shared and distributed memory ranks.

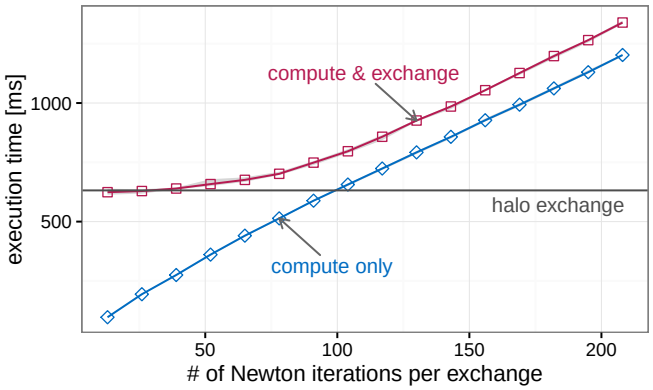


Fig. 7: Overlap for square root calculation (Newton-Raphson).

The dCUDA programming model promises automatic overlap of computation and communication. To measure this effect, we design a benchmark that iteratively executes a compute phase followed by a halo exchange phase. To determine the overlap, we implement runtime switches that allow to separately disable the compute and halo exchange phases. We use runtime switches to avoid code generation effects that might influence the overall performance of the benchmark. We expect that the execution time of the full benchmark varies between the maximum of compute and halo exchange time for perfect overlap and the sum of compute and halo exchange time for no overlap. To investigate the effect of different workloads, we additionally implement square root calculation (Newton-Raphson) and memory-to-memory copy as examples for compute-bound and memory bandwidth-bound computations. To demonstrate the overlap of computation and communication, Figure 7 and Figure 8 compare the execution time with and without halo exchange for increasing amounts of computation. An additional horizontal line marks the halo exchange only time. We run all experiments on eight nodes of our cluster. Thereby, each halo exchange moves 1kB packets, each copy iteration moves 1kB of data, and each square root iteration performs 128 divisions per rank. We measure perfect overlap for memory bandwidth-bound workloads and good

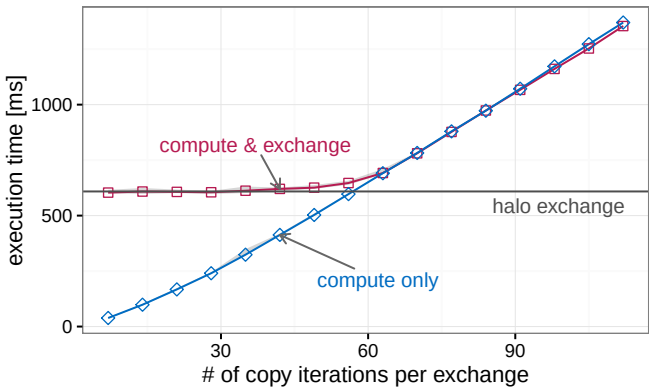


Fig. 8: Overlap for memory-to-memory copy.

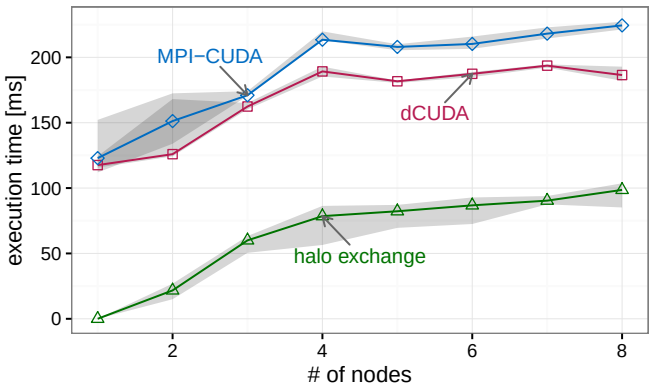


Fig. 9: Weak scaling for the particle simulation example.

overlap for compute-bound workloads. We explain the slightly lower overlap for compute-bound workloads by the fact that the notification matching itself is relatively compute heavy.

### C. Mini-applications

To evaluate the absolute performance of our programming model, we compare MPI-CUDA and dCUDA variants of mini-applications that implement a particle simulation, a stencil program, and sparse matrix-vector multiplication. Three algorithmic motifs that are prevalent in high-performance computing. The main loops of the MPI-CUDA variants run on the host, invoke kernels, and communicate using two-sided MPI, while the main loops of the dCUDA variants run on the device and communicate using notified remote memory access. Otherwise, the implementation variants share the entire application logic and the overall structure. None of them implements manual overlap of computation and communication.

*Particle Simulation:* Our first mini-application simulates particles in a two-dimensional space that interact via short-range repulsive forces. We integrate the particle positions using simplified Verlet integration considering only forces between particles that are within a parameterizable cutoff distance. Just like the particle-in-cell method used for plasma simulations [5], we decompose our wide rectangular domain into cells that are aligned along the wide edge of the domain. Furthermore, we chose the cell width to be lower or equal to the cutoff distance and consequently only compute forces between particles that are either in the same cell or in neighboring cells. After each integration step we update the particle positions and move them to neighboring cells if necessary.

We organize the data using a structure of arrays that hold position, velocity, and acceleration of the particles. We thereby assign the cells to fixed-size, non-overlapping index ranges and use additional counters to keep track of the number of particles per cell. To deal with non uniform particle distributions among the cells, we allocate four times more storage than necessary to fit all particles. To support distributed memory, we decompose the arrays and allocate an additional halo cell at each subdomain boundary.

The main loop of the particle simulation performs the following steps: 1) we perform a halo cell exchange between neighboring ranks, 2) we compute the forces and update the particle positions, 3) we sort out the particles that move to a neighbor cell, 4) we communicate the particles that move to a neighbor rank, and 5) we integrate the particles that arrived from a neighbor cell. To copy the minimal amount of data, the MPI-CUDA variant continuously fetches the book keeping counters to the host memory. In contrast, the main loop of the dCUDA variant runs on the device and has direct access to all data. Thereby, each rank registers one window per array that spans the cells assigned to the rank plus two halo cells. The windows of neighboring shared memory ranks physically overlap, which means, as in case of the MPI-CUDA variant, actual data movement only takes place for distributed memory ranks. However, in contrast to MPI-CUDA the synchronization is much more fine grained enabling overlap of computation and communication.

Figure 9 shows weak scaling for both implementation variants as well as the halo exchange time measured by the MPI-CUDA variant. We thereby use a constant workload of 416 cells and 41,600 particles per node. Typically, the simulation would be compute-bound, but as we are interested in communication we reduced the cutoff distance so that there are very few particle interactions. Consequently, the simulation becomes more memory bandwidth-bound. We perform two memory accesses in the innermost loop that computes the particles distances. Aggregated over 100 iterations and assuming a total execution time of 200ms, we get an estimated bandwidth requirement of roughly 100GB/s compared to 240GB/s peak bandwidth. Keeping in mind that the code also performs various other steps this shows that our implementation performs reasonably. While the two implementation variants perform similarly up to three nodes, the dCUDA variant clearly outperforms the MPI-CUDA variant for higher node counts. The scaling costs of the MPI-CUDA variant roughly correspond to the halo exchange time, while the dCUDA variant can partly overlap the halo exchange costs. However, the particle simulation is dynamic and during execution load imbalances evolve. For example, the minimal and maximal halo exchange

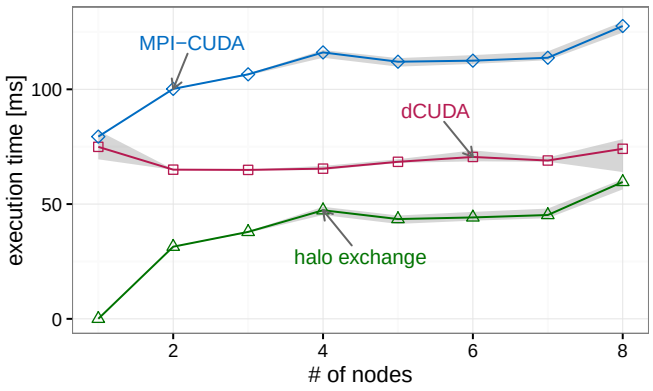


Fig. 10: Weak scaling of the stencil program example.

times measured on eight nodes differ by a factor of two. We therefore do not expect an entirely flat scaling.

*Stencil Program:* Our second mini-application iteratively executes a simplified version [11] of the horizontal diffusion kernel derived from the COSMO atmospheric model [2]. Just like COSMO, we consecutively apply the four dependent stencils to a three-dimensional regular grid with a limited number of vertical levels. The stencils themselves are rather small and consume between two and four neighboring points in the horizontal ij-plane.

Our implementation organizes the data using five three-dimensional arrays that are stored in column-major order. We perform a one-dimensional domain decomposition along the j-dimension and extend the sub-domains with a one-point halo in both j-directions. Consequently, the halos consist of one continuous storage segment per vertical k-level.

The main loop of the stencil program contains three compute phases each of them followed by a halo exchange. In total, we execute four stencils and communicate four one-point halos per loop iteration. To apply the stencils, we assign each block to an ij-patch that covers the full i-dimension. For each array, the dCUDA variant registers a window that spans the ij-patch assigned to the rank plus one halo line in each j-direction. The windows of neighboring shared memory ranks overlap and data movements only take place between distributed memory ranks. To improve the performance, the MPI-CUDA variant additionally copies the data to a continuous communication buffer that allows to wrap the entire halo exchange in a single message.

Figure 10 shows weak scaling for both implementation variants as well as the halo exchange time measured by the MPI-CUDA variant. We chose a domain size of $128\times320\times26$ grid points per device. The stencil program accesses eight different arrays per iteration. Aggregated over 100 iterations and assuming a total execution time of 70ms, we compute an approximate bandwidth requirement of 100GB/s compared to 240GB/s peak bandwidth. Hence, the overall performance of our implementation is reasonable. While both implementation variants have similar single-node performance, the dCUDA variant excels in multi-node setups. The scaling costs of the MPI-CUDA variant roughly correspond to the halo exchange time, while the dCUDA variant can completely overlap the significant halo exchange costs. This is possible as the stencil program is perfectly load balanced and the halo exchange costs are the only contribution the scaling costs. To achieve better bandwidth, OpenMPI by default stages messages larger than 30kB through the host. The MPI-CUDA variant sends one 26kB message per halo, while the dCUDA variant sends 26 separate 1kB messages (one per vertical layer). Hence, with the given configuration both implementation variants perform direct device-to-device communication. However, introducing additional vertical layers improves the relative performance of the MPI-CUDA variant as it benefits from the higher bandwidth of host staged transfers.

*Sparse Matrix-vector Multiplication:* Our third mini-application implements sparse matrix-vector multiplication followed by a barrier synchronization. The barrier synchronization thereby emulates possible follow up steps that synchronize the execution, the worst-case for dCUDA's overlap philosophy. For example, the normalization of the output vector performed by the power method.

We store the sparse matrix using the compressed row storage (CSR) format and distribute the data using a two-dimensional domain decomposition that splits the matrix into square sub-domains. Furthermore, we store the sparse input and output vectors along the first row and the first column of the domain decomposition respectively. To process the matrix sub-domains, we assign each row of the matrix patch to exactly one block.

The main loop of the application performs the following steps: 1) we broadcast the input vector along the columns of the domain decomposition, 2) each rank locally computes the matrix-vector product, 3) we aggregate the result vectors along the rows of the domain decomposition, and 4) we synchronize the execution of all ranks. We thereby manually implement the broadcast and reduction collectives using a binary tree communication pattern. The dCUDA variant over-decomposes the problem along the columns of the domain decomposition. Hence, the depth of the broadcast tree is higher while the message size corresponds to the MPI-CUDA variant. In contrast, along the rows of the domain decomposition the reduction tree has the same depth while the dCUDA variant sends more but smaller messages.

Figure 11 shows the weak scaling for both implementation variants as well as the halo exchange time measured by the MPI-CUDA variant. We run our experiments using a $10{,}816\times10{,}816$ element matrix per device and randomly populate 0.3% of the elements. Our matrix-vector multiplication performs roughly a factor two slower than the cuSPARSE vendor library. While the MPI-CUDA variant performs slightly better for small node counts, the dCUDA variant seems to catch up for larger node counts. However, due to the tight synchronization, we do not observe relevant overlap of computation and communication. The scaling cost for both implementation variants corresponds roughly to the communication time. We therefore conjecture that the short and tightly

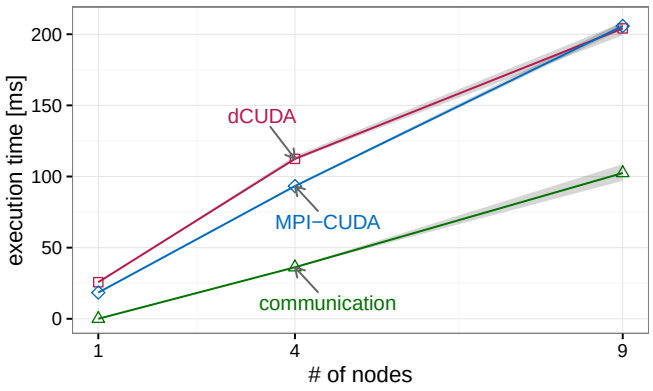


Fig. 11: Weak scaling of the sparse matrix-vector example.

synchronized compute phases provide not enough room for overlap of computation and communication. Furthermore, the MPI-CUDA variant stages the reduction messages through the host, while the dCUDA variant due to the higher message rate uses direct device-to-device communication. Therefore, the dCUDA variant suffers from lower network bandwidth which might overcompensate potential latency hiding effects. We show this example to demonstrate that, even in the worst-case of very limited overlap, dCUDA performs comparable to MPI-CUDA. Advanced algorithmic methods could be used to enable automatic overlap even in Krylov subspace solvers [8].

## V. Discussion

To further improve the expressiveness and the performance of the dCUDA programming model, we briefly discuss possible enhancements.

*Collectives:* Over-decomposition makes collectives more expensive as the their cost typically increase with the number of participating ranks. We suggest to implement highly-efficient collectives that leverage shared memory [10], [16]. Furthermore, one can imagine nonblocking collectives that run asynchronously in the background and notify the participating ranks after completion.

*Multi-Dimensional Storage:* Our implementation currently only supports one-dimensional storage similar to dynamically allocated memory in C programs. We suggest to add support for multi-dimensional storage as it commonly appears in scientific applications. For example, we could provide a variant of the put method that copies a rectangular region of a two-dimensional array.

*Shared Memory:* With our programming model hundreds of ranks work on the same shared memory domain. We suggest to add functionality that makes better use of shared memory. For example, we could provide a variant of the put method that transfers data only once and then notifies all ranks associated to the target memory.

*Host Ranks:* To fully utilize the compute power of host and device, we suggest to extend our programming model with host ranks that like the device ranks communicate using notified remote memory access.

## VI. Related Work

Over the past years, various GPU cluster programming models and approaches have been introduced. For example, the rCUDA [6] virtualization framework makes all devices of the cluster available to a single node. The framework therefore intercepts calls to the CUDA runtime and transparently forwards them to the node that hosts the corresponding device. Consequently, CUDA programs that support multiple devices require no code changes to make use of the full cluster. Reaño et al. [25] provide an extensive list of the different virtualization frameworks currently available.

Multiple programming models provide some sort of device-side communication infrastructure. The FLAT [18] compiler transforms code with device-side MPI calls into traditional MPI-CUDA code with host-side MPI calls. Consequently, the approach provides the convenience of device-side MPI calls without actually implementing them. GPUNet [15] implements device-side sockets that enable MapReduce-style applications with the device acting as server for incoming requests. Key design choices of dCUDA, such as the circular buffer based host-device communication and the mapping of ranks to blocks, are inspired by GPUNet. DCGN [28] supports device-side as well as host-side compute kernels that communicate using message passing. To avoid device-side locking, the framework introduces the concept of slots that limit the maximum number of simultaneous communication requests. In a follow-up paper [29] the authors additionally discuss different rank to accelerator mapping options. GGAS [20] implements device-side remote memory access using custom-built network adapters that enable device-to-device communication without host interaction. However, the programming model synchronizes the device execution before performing remote memory accesses and therefore prevents any hardware supported overlap of computation and communication. GPUrdma [4] was developed in parallel with dCUDA and implements device-side remote memory access over InfiniBand using firmware and driver modifications that enable device-to-device communication without host interaction.

Multiple works discuss technology aspects that are relevant for the programming model design. Oden et al. [21] control InfiniBand network adapters directly from the device without any host interaction. Their implementation relies on driver manipulations and system call interception. However, the host controlled communication nevertheless excels in terms of performance. Furthermore, Xiao and Feng [31] introduce device-side barrier synchronization and Tanasic et al. [30] discuss two different hardware preemption techniques.

## VII. Conclusion

With dCUDA we introduce a unified GPU cluster programming model that follows a latency hiding philosophy. Therefore, we enhance the CUDA programming model with device-side remote memory access functionality. To hide memory and instruction pipeline latencies, CUDA programs over-decompose the problem and run many more threads than there are hardware execution units. In case there is enough

spare parallelism, this technique enables efficient resource utilization. Using the same technique, dCUDA additionally hides the latency of remote memory access operations. Our experiments demonstrate the usefulness of the approach for mini-applications that implement different algorithmic motifs. We expect that real-word applications will draw significant benefit from automatic cluster-wide latency hiding and overlap of computation and communication. Especially since implementing manual overlap results in seriously increased code complexity. Overall, dCUDA stands for a paradigm shift away from coarse-grained sequential communication phases towards more fine-grained overlapped communication. Although the high message rate of fine-grained communication is clearly challenging for todays networks, our experiments show that the potential of the approach outweighs this drawback.

## ACKNOWLEDGMENT

This publication has been funded by Swissuniversities through the Platform for Advanced Scientific Computing initiative (PASC). We thank CSCS and especially Hussein N. Harake (CSCS) for granting access to the Greina compute cluster and more importantly for remodeling it according to our needs. We thank Peter Messmer (NVIDIA) for constructive discussions and Axel Koehler (NVIDIA) for providing access to the PSG cluster. Finally, we thank the anonymous reviewers of SC16 for their helpful feedback.

## REFERENCES

[1] A proof for the queuing formula: L = $\lambda$w. *Operations Research*, 9(3):383–387, 1961.

[2] Michael Baldauf, Axel Seifert, Jochen Frstner, Detlev Majewski, Matthias Raschendorfer, and Thorsten Reinhardt. Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities. *Monthly Weather Review*, 139(12):3887–3905, 2011.

[3] R. Belli and T. Hoefler. Notified access: Extending remote memory access programming models for producer-consumer synchronization. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 871–881, May 2015.

[4] Feras Daoud, Amir Watad, and Mark Silberstein. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, pages 6:1–6:8, New York, NY, USA, 2016. ACM.

[5] John M. Dawson. Particle simulation of plasmas. *Rev. Mod. Phys.*, 55:403–447, Apr 1983.

[6] José Duato, Francisco D. Igual, Rafael Mayo, Antonio J. Peña, Enrique S. Quintana-Ortí, and Federico Silla. An efficient implementation of GPU virtualization in high performance clusters. In *Proceedings of the 2009 International Conference on Parallel Processing*, Euro-Par'09, pages 385–394, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] MPI Forum. MPI: A message-passing interface standard. version 3.1. http://www.mpi-forum.org, June 4th 2015.

[8] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Comput.*, 40(7):224–238, July 2014.

[9] Dror Goldenberg. Co-desing architecture. http://slideshare.net/insideHPC/co-design-architecture-for-exascale, March 2016.

[10] Richard L. Graham and Galen Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 177–186, New York, NY, USA, 2015. ACM.

[12] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote memory access programming in MPI-3. *ACM Trans. Parallel Comput.*, 2(2):9:1–9:26, June 2015.

[13] J. B. White III and J. J. Dongarra. Overlapping computation and communication for advection on hybrid parallel computers. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 59–67, May 2011.

[14] Cray Inc. Using the GNI and MAPP APIs. Ver. S-2446-52. http://docs.cray.com/, March 2014.

[15] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216, Berkeley, CA, USA, 2014. USENIX Association.

[16] Shigang Li, Torsten Hoefler, and Marc Snir. NUMA-aware shared-memory collective communication for MPI. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 85–96, New York, NY, USA, 2013. ACM.

[17] Paulius Micikevicius. GPU performance analysis and optimization. http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf, 2012.

[18] Takefumi Miyoshi, Hidetsugu Irie, Keigo Shima, Hiroki Honda, Masaaki Kondo, and Tsutomu Yoshinaga. FLAT: A GPU programming framework to provide embedded MPI. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 20–29, New York, NY, USA, 2012. ACM.

[19] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[20] L. Oden and H. Fröning. GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, Sept 2013.

[21] L. Oden, H. Fröning, and F. J. Pfreundt. Infiniband-verbs on GPU: A case study of controlling an infiniband network device from the GPU. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 976–983, May 2014.

[22] E. H. Phillips and M. Fatica. Implementing the himeno benchmark with CUDA on GPU clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, April 2010.

[23] J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–9, Nov 2008.

[24] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 80–89, Oct 2013.

[25] C. Reao, R. Mayo, E. S. Quintana-Ort, F. Silla, J. Duato, and A. J. Pea. Influence of InfiniBand FDR on the performance of remote GPU virtualization. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, Sept 2013.

[26] Davide Rosetti. A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology. https://github.com/NVIDIA/gdrcopy, 2015.

[27] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-fold speedup, 15.0 tflops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov 2010.

[28] J. A. Stuart and J. D. Owens. Message passing on data-parallel architectures. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[29] Jeff A. Stuart, Pavan Balaji, and John D. Owens. Extending MPI to accelerators. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ASBD '11, pages 19–23, New York, NY, USA, 2011. ACM.

[30] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 193–204, June 2014.

[31] S. Xiao and W. C. Feng. Inter-block GPU communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.