ETH zürich

ETH Library

dCUDA: GPU Cluster Programming using IB Verbs

Master Thesis

Author(s):
Kuster, Lukas

Publication date:
2017

Permanent link:
https://doi.org/10.3929/ethz-a-010889936

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

dCUDA: GPU Cluster Programming using IB Verbs

Lukas Kuster

Master Thesis
March 2017

Supervised by
Prof. Torsten Hoefler
Tobias Gysi

at the
Scalable Parallel Computing Laboratory
Department of Computer Science
ETH Zurich

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Over the last decade, the usage of GPU hardware to accelerate high performance computing tasks increased due to the parallel computing nature of GPUs and their floating point performance. Despite the popularity of GPU accelerated compute clusters for high performance applications like atmospheric simulations, no unified programming model for GPU cluster programming is established in the community. dCUDA, a unified programming model for GPU cluster programming designed at ETH Zürich, is a candidate to fill this gap. dCUDA provides a device-side library for Remote Memory Access (RMA) of a global address space and supports fine-grained communication and synchronisation on the level of CUDA thread blocks. The dCUDA programming model enables native overlap of communication and computation for latency hiding and better utilisation of GPU resources by oversubscribing the system. dCUDA provides a well designed communication library and outperforms traditional GPU cluster programming approaches.

We extend the dCUDA library and improve the performance of the dCUDA programming model. New functionality enables new low latency synchronisation mechanisms and several optimisations increase the overall performance of dCUDA. The replacement of MPI based communication by a newly designed InfiniBand Verbs based network manager decreases dCUDA's remote communication latency by a factor of 2 to 3. Device local communication latency was improved by a factor of almost 3 thanks to a redesign of the dCUDA notifications system. Performance benchmarks demonstrate that the optimised framework using the InfiniBand network manager outperforms MPI based dCUDA implementations. Depending on communication patterns, it shows about 40% to 90% improved performance to traditional state of the art GPU cluster programming approaches.

Acknowledgements

I would like to thank my advisors Tobias Gysi and Professor Torsten Hoefler of the Scalable Parallel Computing lab at ETH Zürich. The constructive discussions motivated my work and set me on the right tracks.

I would also like to thank Hussein N. Harake, the HPC system manager at CSCS. He set up and configured the test system on the *greina* cluster. His support on technical issues was vital for the success of the project.

Finally, I would like to thank Mark Silberstein, professor at Technion - the Israel Institute of Technology, for the exchange of ideas and the clarifications regarding GPUrdma.

Contents

List of Figures vii

List of Tables ix

List of Listings xi

1 Introduction 1

2 Background 5
- 2.1 CUDA Programming Model 5
- 2.2 MPI 6
- 2.3 RDMA 7
- 2.4 InfiniBand 7
- 2.5 GPUDirect 10

3 Programming Model 13
- 3.1 dCUDA Programming Model 13
- 3.2 dCUDA Code Example 14
  - 3.2.1 Host Code 15
  - 3.2.2 Device Code 17
- 3.3 dCUDA Interface 20
  - 3.3.1 Host API 20
  - 3.3.2 Device API 21
  - 3.3.3 Interface Changes 25

4 Implementation 29
4.1 Architecture Overview 29
4.2 Network Manager 32
4.3 Host-Device Communication 35
4.4 Notification Management 36
4.5 Notified Access 37
4.6 Deployment 40

5 Evaluation 45
5.1 Test Environment and Timing 45
5.2 System Latency Benchmarks 47
5.2.1 Network Communication Latency 49
5.3 Bandwidth 50
5.4 Case Study: Particles 51
5.5 Case Study: Stencil Program 55
5.6 Case Study: Sparse Matrix Vector Multiplication 58
5.7 Case Study: Power Iteration 61

6 Related Work 67

7 Conclusions 69

Bibliography 73

List of Figures

2.1 Fully connected system with different connection modes. Illustrations inspired by Subramoni et al. [25]. 10

4.1 dCUDA architecture overview: Device library instances with corresponding block managers in the host runtime system. 30
4.2 InfiniBand Network Manager 33
4.3 Sequence diagram of notified memory put using InfiniBand Verbs 39
4.4 Sequence diagram of notified memory put using MPI 40

5.1 Test system configuration 46
5.2 Particle algorithm communication pattern 52
5.3 Particles weak scaling with InfiniBand network manager. 53
5.4 Particles weak scaling with MPI. 54
5.5 Particles weak scaling with two versions of dCUDA. 55
5.6 stencils weak scaling with InfiniBand network manager. 56
5.7 Stencils weak scaling with MPI. 57
5.8 Stencils weak scaling with two versions of dCUDA. 57
5.9 Sparse Matrix Vector Multiplication domain decomposition. 59
5.10 Matrix vector multiplication weak scaling with InfiniBand network manager. . 60
5.11 Matrix vector multiplication weak scaling with MPI and IB-CUDA as a reference. 60
5.12 Power Iteration work steps 62
5.13 Sequence diagram of the power iteration algorithm. 63
5.14 Power iteration weak scaling with InfiniBand network manager. 65
5.15 Power iteration weak scaling with MPI and IB-CUDA as reference. 66

List of Tables

5.1 Notified access and notify latency in μs. 47
5.2 Host-Device ping-pong on flag in host memory, flag in device memory and using dCUDAs flagged queues. Results are round trip times in μs. 48
5.3 RMA ping-pong mean latencies in μs 50
5.4 Bandwidth of one-directional notified accesses in GB/s. 51

List of Listings

3.1 Host Example Code: dCUDA initialisation and kernel invocation. 16
3.2 Device Example Code: Distributed reduction 18
3.3 DCuda Host API 21
3.4 dCUDA device API, initialisation and timing 22
3.5 dCUDA Device API: Notified RMA 23

4.1 Initiator QP pool algorithm 34
4.2 dCUDA wait notifications library function 36

1

Introduction

Supercomputers are specialized computers with tremendous computing power that are used by scientists and engineers to perform computationally intensive tasks like molecular dynamics [26], atmospheric models [3], turbulent combustion simulation [10] and other scientifically important tasks. They consist of up to thousands of individual computing nodes that are interconnected by a high speed network like Infiniband. To efficiently use all of the compute nodes, a programmer has to divide his problem into smaller sub-problems that are computed on individual nodes and minimise communication with other computing nodes. For example, in an atmospheric model, each computing node simulates the atmosphere for only a small region. But the state of the atmosphere at the border of a region depends also on the state of the neighbouring regions. Therefore, the state of the borders of the regions have to be exchanged with the neighbouring nodes in each simulation step. Programming models help the programmer to design the communication and synchronisation of the individual nodes that compute the problem in parallel. In this thesis, we analyse a new programming model called dCUDA (distributed Cuda) [8] designed for GPU accelerated computing clusters. We optimize the implementation described in [8] to get improved performance as we show in multiple case studies. We compare the optimized dCUDA to traditional programming models and show its advantages and disadvantages.

GPUs (Graphics Processing Units) became popular in the 1990s for 2D and 3D graphic rendering in personal computers. GPUs are specialised units that were originally designed to process 2D and 3D graphics. Graphics processing requires a high floating point operation throughput, which can be highly parallelised since each vertex or pixel can be computed independently. Scientists quickly realised that the floating point and parallelisation capabilities of GPUs are well suited to accelerate scientific computations. First, scientists used OpenGL to develop general purpose programs for GPUs, but this changed in 2007 with the introduction of the CUDA development environment and Tesla, the first general purpose GPU (GPGPU), by NVIDIA.

The programming model of CUDA allowed programmers to accelerate their high performance applications with the capabilities of GPUs with only little programming effort.

Today, supercomputers often heavily rely on the performance benefits of GPUs. Programmers use the CUDA programming model to manage CPU and GPU execution as well as memory transfers between CPU and GPU, which use different virtual memory spaces. In the CUDA programming model, the CPU initiates all memory transfers from and to GPU memory. The CPU also manages the execution on the GPU by starting so called CUDA kernels, which are special functions executed on the GPU. As opposed to the CPU, the GPU executes code not on a single thread, but up to thousands of threads execute the same code in parallel. This execution model is called Single-Instruction, Multiple-Threads architecture (SIMT). Threads are grouped into CUDA thread blocks. These thread blocks can access special block shared memory and have special synchronisation mechanisms. CUDA lacks any synchronisation primitives to synchronise threads in different thread blocks as it lacks a consistent memory model for global memory. Synchronisation and memory consistency is only guaranteed at the start and the end of a kernel function. Section 2.1 provides more detail on CUDA.

Compute clusters like supercomputers require another programming model that defines a standard for inter-node communication. The most common model is the Message Passing Interface (MPI), which defines a standard interface for message passing and remote memory access in any kind of network. MPI represents processes by ranks that can communicate with each other over a communicator using two-sided as well as one-sided message passing patterns. This model seems appropriate since the processes usually reside on different nodes and do not share the same memory space, therefore, messages have to be passed anyway. Two sided communication requires sender and receiver to be synchronised. The sender cannot complete a transaction without the receiver being ready to receive the message by calling the appropriate receive function. On the other hand, one sided communication does not synchronise source and target nodes. The target node does not participate in the communication at all. The sender copies data directly into the target's memory without any help from the target's CPU, using a Remote Memory Access (see section 2.3). For more information on MPI see section 2.2.

Modern high-performance computation applications often run on large compute clusters with GPU accelerated compute nodes. Programmers commonly use two programming models to program these kinds of applications. While the CUDA runtime can access GPU memory and execute code on the GPU, MPI is used to communicate with other nodes that participate in completing the task. Traditionally, an MPI-CUDA application alternately executes a CUDA kernel followed by a communication step using MPI on the CPU. The CUDA kernel computes the intermediate result of the current step using the massive parallelism of the GPU. At the end of the CUDA kernel execution, all threads are synchronised and the CPU initiates the necessary memory transfers of the intermediate result to and from other nodes. When the new data is read, the CPU can start the next CUDA kernel. Note that by using a CUDA aware MPI implementation, one does not need to copy the intermediate result to CPU in order to send it to another node. CUDA aware MPI can access GPU memory directly using a technology called GPUDirect (Section 2.5). In a typical MPI-CUDA application there is no local overlap of computation and communication. The CPU does not start to send or receive any data while the kernel is running and while the CPU is communicating with other nodes, the GPU is idle. Therefore, the computational power of the GPU is not optimally utilised by employing the traditional MPI-CUDA design.

The dCUDA programming model tries to combine the two models (MPI and CUDA) into one, opening up more optimisation possibilities and better utilisation of the GPU's resources. dCUDA provides message passing functionality on the level of CUDA blocks. Therefore, we do not have to finish a CUDA kernel before starting communication. This natively enables more fine-grained thread synchronisation and overlap of computation and communication, which speeds up the overall execution time. In the dCUDA programming model a rank represents a single CUDA block instead of the whole GPU device. Additionally, dCUDA enables local communication from one thread block to another block on the same device. This enables fine grained synchronisation of single blocks inside a kernel. dCUDA provides one-sided communication patterns on the level of thread blocks similar to the one-sided communication on the level of compute nodes that MPI provides. dCUDA combines remote memory access and notifications. The target of a remote memory access can be notified with a notification message, which can be queried by the receiver. The notification is very useful to synchronise the communication, which has to be done by additional two-sided messages in MPI. Notified remote memory access speeds up the execution time compared to traditional one-sided communication [2].

The dCUDA implementation from the original paper [8] uses MPI for communication. MPI is a high level interface for inter-node communication and abstracts system specific low level interfaces. Supercomputers often employ an InfiniBand high-performance network. The InfiniBand Verbs interface can be used to access the InfiniBand network stack. In this thesis we optimize dCUDA by replacing MPI entirely by InfiniBand Verbs. We aim at lower message latency due to reduction of potential MPI overhead. Section 2.4 provides more detailed information on InfiniBand Verbs. As not every system employs an InfiniBand network, the final implementation of dCUDA as described in this thesis includes a compile time switch to activate MPI. This enables running dCUDA on machines in a system without an InfiniBand interconnect.

In addition to the replacement of MPI by InfiniBand Verbs, we applied other performance and interface optimisations. We simplified the notification interface to enable performance optimisations in the notification handling process, we removed support for remote memory read accesses but added more functionality for memory put and for notify operations. The performance was improved by introducing multi-threading into the CPU code and a new timing infrastructure reduces programming effort for monitoring the execution time of GPU code.

In summary, we investigate different programming models used in high-performance computing applications on GPU accelerated computing clusters. In particular, we study the dCUDA programming model in detail and inspect and re-evaluate several design decisions, which in some cases got revamped in order to achieve better performance. We thereby focus on scalability and performance as well as usablility of the dCUDA library. We provide insights into the process of designing a programming model and implementing the dCUDA library and all of its challenges. Next, we evaluate the performance of the new implementation of dCUDA and compare it to the implementation of dCUDA as described in its original paper. We give a detailed analysis of the communication latencies in the system and report bandwidth for different scenarios. Finally, we will show four case studies. Each case study focuses on one particular mathematical problem, which is solved by a dCUDA application. In addition to the dCUDA implementation, each problem was also solved using traditional approaches using two models. One implementation uses MPI for its communication (MPI-CUDA) and a second implementation uses a InfiniBand Verbs based network manager instead (IB-CUDA). The four problems that we investigate in the

case studies are a Particle simulation, a stencil code, a sparse matrix vector multiplication, and a power iteration. The former three of these problems were already discussed in the original paper of dCUDA [8]. This thesis shows that the new dCUDA implementation is superior to the original one and outperforms traditional state of the art programming models. The power iteration algorithm was designed and implemented as part of this thesis and will be discussed in more detail than the other case studies. It shows new aspects of dCUDA and its advantages and disadvantages. dCUDA was developed on the *greina* cluster at the Swiss National Supercomputing Centre CSCS in Lugano using computing nodes that employ Nvidia K80 GPUs and use a InfiniBand high-performance interconnect.

2

Background

During the last decade, developers and researchers discovered the potential of GPU accelerated programming for highly parallel programs. Today, modern HPC clusters and super computers often employ GPUs to accelerate their parallel tasks. Here we review the state of the art of GPU cluster programming and present the current networking technologies used in today's compute clusters.

2.1 CUDA Programming Model

GPUs were designed to accelerate graphics processing and originally supported only graphics specific functions. In the late 1990s the GPUs became more programmable and could be used for general purpose programs. Initially, making use of the high floating point performance of GPUs was complex and required a lot of effort. In 2006 NVIDIA introduced CUDA (Compute Unified Device Architecture), a parallel computing platform and programming model for general purpose computing on GPUs. CUDA allows developers to write fast high level C code that is executed on the GPU [20, 21].

CUDA extends the C language with kernel functions that are executed on the GPU. Instead of being executed only by one thread like usual C functions, a CUDA kernel is executed N times in parallel by N distinct threads. Each thread executes the same instructions, whereas the only thing that distinguishes one thread from another are the thread and block indices. The threads are grouped into so called thread blocks. The threads in a block are organised in an up to three-dimensional grid and are identified by thread indices. Furthermore, the thread blocks are again organised in a grid of up to three dimensions. The programmer can decide on the dimensions of the thread hierarchy for each individual kernel call.

Each thread can access thread local memory, shared memory and global memory. Local memory is usually located in registers as long as there are enough registers to hold all local data. Therefore, memory access to local memory is the fastest option. Local variables are stored in thread local memory. Variables that are declared with the keyword `__shared__` are stored in shared memory that can be accessed by all threads in the same block. Shared memory access is much faster than global memory and should be used wherever it is suitable. To synchronize shared memory access, CUDA provides block synchronisation intrinsics like `__synchthreads()` that act like a barrier for all threads in the block. Global memory has the longest access time but can be accessed by all threads in a kernel as well as by the CPU with special CUDA functions. Therefore, global memory is usually used to transfer the initial data to the GPU and the result back to the CPU. Global memory can be allocated by CUDA memory allocation functions. The CUDA memory model does not guarantee any memory consistency on global memory and different thread blocks may see memory accesses in different orders. To prevent read-write and other memory conflicts, one has to use atomic operations to concurrently modify global memory and memory fence instructions to maintain consistent memory. At the start and the end of a kernel execution global memory is consistent and CUDA thread blocks are synchronized. CUDA applications often run multiple kernels consecutively to get into a consistent state in-between kernel invocations.

The hardware of a modern Nvidia GPU consists of multiple multi-threaded streaming multiprocessors (SM). When a kernel is launched, thread blocks are distributed to SMs with available capacity. All threads of one thread block are executed on the same multiprocessor, but multiple blocks can be executed on the same multiprocessor concurrently. To manage hundreds of concurrent threads the multiprocessors have a special architecture called Single-Instruction, Multiple-Thread (SIMT) architecture. A multiprocessor executes threads in groups of 32 threads, which are called warps. The threads of a warp are always scheduled at the same time and simultaneously execute the same instructions. If some threads in a warp are in different branches, each branch is executed sequentially and threads that are not part of the current branch are deactivated. For the best performance, a programmer should always try to avoid warp divergence.

2.2 MPI

Fast communication in a high performance cluster is essential for parallel programs. The Message Passing Interface (MPI) provides a high level interface for communication between processes running on different machines [18]. MPI does not define a protocol or implementation but defines a set of send/receive and collective communication functions. The interface abstracts all underlying network and protocol specific implementation details and the same code runs on any kind of system. Various MPI implementations exist that are continuously updated to support the most recent hardware and network technologies.

An MPI program starts N processes on M nodes. Each process has its own virtual memory. MPI groups all processes into a world communicator. The user can create additional custom communicators that contain a subset of the processes. Each process in a communicator has a unique rank index associated with that communicator. MPI provides several different functions to communicate with other ranks in a communicator, e.g. the straightforward send/receive

functions. MPI defines various collective functions that have to be called by all members of a communicator collectively. For example, the gather function can be used to collectively gather the individual results of all the nodes and collect them at the main process for further processing. For all these functions exist a blocking and a non-blocking variant. Blocking functions do not return until all participating ranks have called and finished the corresponding function, while non-blocking functions only initiate the transaction and do not wait for completion of the operation. This is useful as the corresponding processes can do other work while the data is transferred. The programmer has to keep in mind that send and receive buffers can not be used until the non-blocking operation has completed. To test if non-blocking operations have completed, one can call blocking wait routine or non-blocking test routines.

In addition to the two-sided communication, MPI-3 introduced one-sided communication patterns. One-sided communication does only need one process to actively participate in the communication. One-sided communication needs every rank in the communicator to create a shared window that specifies memory that is exposed to the other ranks in the communicator for remote memory access (RMA). After the window creation, each rank can put and get memory to and from a remote memory window without target process interaction. There is no synchronisation of the calling process and the remote process. Often additional two-sided messages are needed to synchronise the ranks after one-sided operations have completed.

2.3 RDMA

Remote Direct Memory Access or short RDMA is a technology that allows a host to access remote memory of another machine without participation of the remote CPU. In other words, it's a technique to implement RMA (Remote Memory Access). Note that RMA can also be implemented without RDMA. RMA is a concept that allows a process to access remote memory of a remote process. The remote process does not need to actively participate in the memory transaction. RMA does not define how it is implemented. A possibility would be that a background thread executes a routine that receives incoming messages and reads/writes local memory on the remote machine. RDMA implements RMA without any CPU usage on the target machine for the memory transactions.

A process can register local memory at a network adapter that supports RDMA. Registered Memory will be exposed to remote memory accesses and it is associated with a key that has to be sent to all peers before they can access this memory region. If a process knows the address and the key of a remote memory region, it can send a request to read or write that memory. The network adapter of the remote process will receive the request and will process it without interrupting or notifying the CPU. The network adapter can bypass the CPU and accesses the main memory directly using the local Direct Memory Access (DMA) engine.

2.4 InfiniBand

InfiniBand (IB) is a high-bandwidth, low latency network architecture. It is used in high performance clusters and enterprise data centers. It competes with traditional networking architectures

like Ethernet.

In traditional network architectures, the network adapter resources are owned and managed by the operation system (OS). When a process needs to send a message, a context switch to the OS is needed. The OS moves the data from the virtual buffer space though the network stack onto the wire. InfiniBand on the other hand, directly provides the applications with the networking services without OS interaction. It provides protected channels that allow applications to use network resources directly while ensuring data isolation and protection [15]. The endpoints of these channels are called QPs (Queue Pairs). By accessing a QP, an application can accesses the resources of the network adapter card directly. On creation, the QPs are mapped to the applications user space. An application can own multiple QPs if it needs multiple channels. Usually a channel is bidirectional, which is the reason why a queue pair consists of a send queue and a receive queue. The InfiniBand Verbs API provides the user with functions to create and access all relevant resources to control the InfiniBand network adapter.

The send queue is used to post send requests. The InfiniBand HCA (Host Channel Adapter) polls and processes all the requests in the send queue. If an application expects to receive messages, it has to post receive requests on the receive queue. Whenever the HCA receives a message, it polls a receive request from the receive queue and uses the information from the request to process the receiving message. Both the send queue and the receive queue have a CQ (Completion Queue) associated with it. Whenever a request is finished, the HCA produces a completion entry and pushes it on the corresponding CQ. The application has to poll these CQs to know if a send operation has completed or to know if it has received a message.

InfiniBand provides two different messaging methods. First there are the traditional two-sided send/receive functions. The sender posts a send work request on the send queue of the QP. The work request contains the send buffer location. The HCA reads the buffer memory and puts it on the wire. The receiver has to pre-post a receive work request on the receive queue of the QP. The work request contains the location of the buffer where the message will be stored by the HCA. It is important that there is always a receive work request in the receive queue when such a message arrives because otherwise, the HCA does not know how to process the data and the QP goes into an error state. The receiver can poll the CQ of his receive queue to get informed about incoming messages.

The second data transfer method is called RDMA write and RDMA read (Remote Direct Memory Access). When using this method, only the CPU of the initiator of the data transfer is involved. The initiator posts a RDMA write or read work request giving the local buffer location as well as the remote buffer location. On the receiving side, the HCA directly writes or reads at the specified memory location without interrupting or notifying the CPU of that node. To prevent memory corruption by remote applications, RDMA functions can only access remote memory that was registered using the InfiniBand Verbs API. Registered memory is associated with a remote key and a local key for remote and local accesses. Remote memory can only be accessed if the application knows the remote key of that memory region. The remote keys have to be exchanged at application start and prevent unauthorized memory accesses.

The target of a RDMA write or read is not notified and the process does not know when data is transferred. An additional round trip time is needed to synchronize the processes before or after a set of RDMA operations. As stated by R. Belli and T. Hoefler, notified RMA accesses can provide up to 50% application speedup for small messages [2]. InfiniBand provides a function

that in addition to a RDMA write also notifies the target process. This function called *RDMA write with immediate* does the same as RDMA write but also sends a 32 bit immediate value within the message. The target process can poll the receive CQ to get the immediate value as soon as the RDMA write is completed. A similar function for RDMA reads does not exist in the InfiniBand Verbs standard.

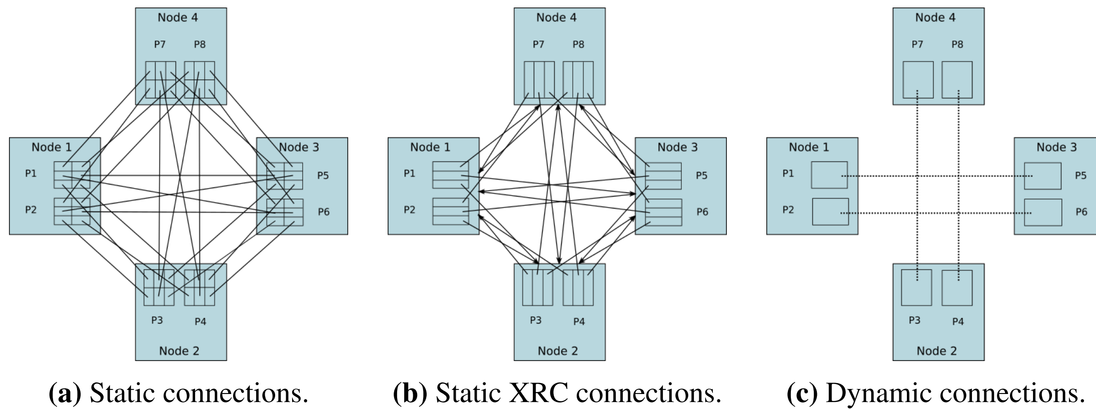
InfiniBand supports several different transport protocols. The most common one is the RC (Reliable Connection) transport. RC QPs are endpoints of a static communication channel. They are comparable to TCP sockets, although not quite the same. Every message sent on a reliable channel is followed by an acknowledgement message. A send-request can only be completed when the acknowledgement was returned. In a large network, as they are common in compute clusters, each process needs to maintain a large number of QPs to get full connectivity, which needs a lot of resources on each node. For example, a system with N nodes and P processes on each node needs $N \cdot P \cdot P$ QPs on each node to have full connectivity. RC transport has good performance but for a system with millions of processes, it consumes a lot of resources.

The second transport protocol is the UC (Unreliable Connection) transport. A UC QP is associated with exactly one other QP, just like RC. The only difference is that the connection is not reliable and packets can be lost. Figure 2.1a depicts the static RC or UC channels of a fully connected system of four nodes with 2 processes each. Intra-node communication is not shown. Each process has a channel to every other remote process.

The third type of supported transport protocol is UD (Unreliable Datagram) transport. This protocol is similar to UDP. A UD QP does not instantiate a connection to another QP unlike RC QPs. The destination of each individual message has to be stated. A single UD QP can be used to send messages to all other processes in the system. Therefore, the resource demands in a large system is constant per process. The drawback is that UD transport does not guarantee that the messages reach their targets. If a reliable transport is needed, it has to be implemented by the programmer, which degrades the performance of the network significantly. Figure 2.1c shows a system using UD transport. Channels are dynamic and can change its target at any time.

In addition to the three initial transport protocols, new types were introduced as InfiniBand evolved [16]. The XRC (extended Reliable Connected) transport protocol [11] tries to reduce the number of QP needed for all to all communication in a large cluster with many processes per node, while still being reliable. XRC significantly reduces the number of QPs from $N \cdot P \cdot P$ RC QPs to $N \cdot P$ XRC QPs per node. Each XRC QP connects to a XRC target. The target can be associated with all the QPs on that node. This means that a single QP can send packets to all processes of a single node, not only to a single process. Scalability of XRC transport is better than RC for clusters with a lot of cores but still a lot of resources are needed for large compute clusters with a lot of nodes. Figure 2.1b shows the channels of the XRC protocol. QPs are connected to target nodes instead of individual target QPs.

The last transport protocol introduced into InfiniBand is the DC (Dynamically Connected) transport. DC transport reduces the resource usage even further. It tries to combine the advantages of reliable connections and unreliable datagrams. Each process in the system needs at least one DC initiator QP for sending packets and a DC target to receive incoming packets. When a process needs to send a packet, it identifies the target of the packet and the DC QP dynamically



(a) Static connections. **(b)** Static XRC connections. **(c)** Dynamic connections.

Figure 2.1: Fully connected system with different connection modes. Illustrations inspired by Subramoni et al. [25].

connects to the corresponding DC target. The DC initiator and the DC target initiate a reliable connection for the time packets are sent from the initiator to the target. The connection remains active as long as the initiator sends packets to the same target. The connection gets torn down whenever the initiator changes its target or after a short timeout. The performance of the DC transport is comparable to the performance of RC transport but adds an overhead of roughly 100 nanoseconds to the latency for establishing the dynamic connections [25]. A single DC Initiator can dynamically connect to all targets in the system. Thus, each node in a compute cluster with P processes per node needs only P DC initiators and targets to achieve full connectivity. Therefore, the DC transport protocol scales well for large compute clusters with thousands or millions of nodes as the resource consumption of InfiniBand remains constant in the number of nodes. 2.1c depicts the dynamic connections of the DC transport protocol. Like UD transport, channels are dynamically created and destroyed whenever needed.

2.5 GPUDirect

GPUDirect is a family of technologies introduced by Nvidia for their general purpose GPUs [19]. First, it introduced a technology to support accelerated communication with third party PCI-E devices via shared pinned host memory. This accelerated communication by getting rid of a memory copy in host memory from the CUDA driver buffer into a third party driver buffer like the InfiniBand driver. Memory staging into shared host memory was still necessary. In 2011 Nvidia introduced GPUDirect peer to peer, which supports direct memory access between GPUs on the same PCI-E root complex. This allows memory transfers without passing the host main memory, removing another memory copy. In 2013 GPUDirect RDMA was introduced, which enabled third party drivers to directly access GPU memory. With GPUDirect RDMA a third party driver like the Mellanox InfiniBand driver can directly access GPU memory and bypasses host main memory altogether.

Mellanox supports GPUDirect RDMA in its drivers since the introduction of the Mellanox ConnectX-3 HCA [17]. This allows the InfiniBand HCA to directly access GPU memory and completely bypass host main memory. The system requires Mellanox OFED version 2.1 or

higher and the *nv_peer_mem* plugin to be enabled in order to work. The InfiniBand driver will automatically distinguish host memory and GPU memory and will automatically use GPUDirect RDMA in the latter case if the system is properly set-up.

Several MPI implementations support GPUDirect RDMA, which is often referred to as CUDA-aware MPI. OpenMPI support CUDA-awareness since version 1.7.0. To make use of the features of GPUDirect, one has to enable it by passing the appropriate parameters to *mpirun*, as it is disabled by default [24]. OpenMPI will use GPUDirect only for small packets smaller than 30'000 bytes by default. Larger packets are pipelined and staged through host main memory because it yields better throughput. The point where the protocol switch happens can be adjusted with another MPI parameter.

GPUDirect was designed for direct access of GPU memory from third party devices. The same GPUDirect API can be used to map the GPU memory to the CPU. Gdrcopy is an open-source library based on GPUDirect RDMA for low latency GPU memory copy [22]. It gives the user the ability to map GPU memory to user-space, which then can be used as if it were host memory. As a consequence, gdrcopy can be used to copy data from and to GPU memory with low overhead giving a low latency alternative to the CUDA memory copy routines. Initially, the GPU memory has to be pinned to host memory, which is expensive. When the memory is pinned, all memory copies have a low latency which is essential for a GPU messaging interface in a compute cluster controlled by the CPU as described in this thesis.

3

Programming Model

dCUDA defines and implements a programming model for multi-GPU applications with communication on the level of GPU thread blocks. We define the model here in more detail and we present a short program example to show how dCUDA can be used to program multi-GPU applications with fine-grained communication and synchronization patterns. Further, we discuss the interface and present all functions provided by the dCUDA library. We will also discuss in detail all changes in the library interface made since the original publication of dCUDA [8].

3.1 dCUDA Programming Model

dCUDA combines the programming models of CUDA and MPI. Both models are described in Sections 2.1 and 2.2 respectively. dCUDA kernels are launched with a user configurable number of thread blocks. The threads are organised in one dimension unlike CUDA kernels, which have up to three dimensions. Support for higher dimensional thread hierarchies might be added in future works on dCUDA. dCUDA does not allow to directly control the number of thread blocks a kernel launches. It only launches as many blocks as it is able to run in parallel on the streaming processors of the GPU or a maximum of 208 thread blocks. When CUDA launches more blocks than it could possibly run in parallel, some of them are postponed and are started when other thread blocks have finished execution and freed its resources. This would imply a major problem for dCUDA as inter-block synchronisations lead to potential deadlocks. The user can control the number of thread blocks indirectly by adjusting the number of threads per block or by limiting each thread to a maximum number of registers by setting the compiler flag *-maxrregcount*. Limiting register usage per block allows more blocks to run in parallel but execution might be a bit slower as some local variables are stored in global GPU memory instead of registers. Newer GPU devices that support preemption of thread blocks would relax the

restrictions on the number of spawned thread blocks. The current implementation of dCUDA is designed for non-preemptive GPUs.

In the MPI programming model, each MPI process can communicate with all other processes in a communicator and all processes are identified by a rank index. dCUDA uses fine-grained communication on the level of GPU thread blocks. Therefore, the dCUDA model identifies each thread block of its GPU device as a rank. In the remainder of this work we use the terms thread block and rank interchangeably in the context of dCUDA. dCUDA allows to send data and synchronise with any rank in the system, whether it is a local rank on the same device or a remote rank on another node.

MPI creates a global communicator on initialisation to communicate with other ranks in the system. The user can then create other communicators that consist only of a subset of all the ranks to form smaller communication groups. dCUDA provides two default communicators. First, the global communicator indicated by `DCUDA_COMM_WORLD`, which includes all ranks in the system and allows any two rank to communicate with each other. Additionally, there is a communicator denoted as `DCUDA_COMM_DEVICE`, which includes all local ranks of the same device. Currently, dCUDA does not support custom communicators. Support for it might be added in future versions of dCUDA.

dCUDA's communication model is based on notified RMA [2]. It is similar to one sided communication of MPI but with native support for target notification [7]. Communication requires creation of a memory window in one of the communicators. All ranks of the communicator have to collectively create a memory window that exposes the corresponding memory to remote accesses by other ranks in the same communicator. Any rank in that communicator can then access the memory windows of the other ranks and modify its content. Each put operation can be accompanied by a notification to let the owner of the memory window know that its memory was modified and to synchronise the ranks. Memory windows of ranks on the same device are allowed to overlap. dCUDA is optimised to perform no memory copy if the source and the target buffers point to the same memory location. Ranks that need data from another rank can wait for notifications with specific tags. When the data arrives, the receiver is notified and continues its work.

3.2 dCUDA Code Example

To illustrate the dCUDA programming model and its API, we present a small dCUDA code example. The example illustrates how to initialise dCUDA, how to load data to the GPU and back to the CPU and finally how to write a simple dCUDA kernel with fine grained communication and synchronisation. The simple example code provided here implements reduction. An array of random values is generated and all dCUDA ranks in the system calculate the sum of all values in parallel using reduction. The asymptotic runtime of this simple algorithm is $\mathcal{O}(\log n)$.

3.2.1 Host Code

Listing 3.1 shows the host code that is executed by the CPU. The code initializes dCUDA and generates the initial data that will be summed up by the dCUDA kernel. After launching the kernel, it will report the result and the execution time of the algorithm. Following, we will explain each step of the code in more detail.

First, the code includes all necessary packages and defines a custom struct *user_data* on line 1 to 12. The struct is later used to load data to the GPU. The user can add as many fields to the struct as necessary. In this example the field *data* will point to an array in GPU memory that will hold the initial data and the intermediate results of the algorithm. The *buffer* points to another array in GPU memory that is used as a receive buffer for incoming values from other ranks. Note that the buffer will be shared by all ranks on the same device and therefore has to be properly managed to avoid race conditions. The field *is_leader* indicates that this is the root device in the system. The double *result* will hold the result of the algorithm. Note that the struct can be used to load data to the GPU as well as copying the result back to the CPU after kernel execution.

On program start, the code initializes dCUDA with the function `dcuda_host_init()` on line 17. We pass the dCUDA kernel we want to execute later to the function and the number of threads per block, which is 128 in this example. dCUDA does not allow to directly control the number of blocks that are launched but it gives users the opportunity to query all relevant information so that they can design their algorithm appropriately. The function `dcuda_host_get_rank_info()` gives all relevant information about the configuration of the system. In the code example on lines 22 to 26 we read the number of ranks on the local device, which is equal to the number of thread blocks that will launch, and the total number of ranks in the system that includes all other dCUDA processes that are started in parallel.

Next, we allocate and initialize the arrays we need for the algorithm. The struct *user_data* does only hold pointers to the array but not the arrays themselves. Thus, we have to first allocate them in GPU memory. We decide that each thread of the device will have one value and set the size of the array appropriately. We use `cudaMalloc()` on line 30 to allocate memory on the GPU. Note that we want the two arrays to be in consecutive memory for our algorithm. Therefore, we allocate both arrays with only one call to `cudaMalloc`. In the next step, we initialize the data on lines 34 to 37. We allocate another array in main memory, initialize the array with random numbers, and copy the data to the GPU using `cudaMemcpy()` on line 40.

After having allocated the memory we need and copied the initial data to the GPU, we are ready to launch the actual dCUDA kernel on the GPU on line 44. The function `dcuda_host_run()` copies the userdata as defined in the struct *user_data* to the device, launches the kernel, and copies the userdata back to main memory. The struct holds any result that the kernel has set without the need to manually copy it. The function `dcuda_host_run()` only returns when the kernel finished execution.

After the kernel execution, the main process reports the result of the algorithm, which is stored in the *result* field of the *user_data* struct. On lines 47 to 51 we get the execution time of the algorithm (excluding any initialisation time) by calling `dcuda_host_get_timings()`, which returns all timing results done during kernel execution and we report the result. See the

```
#include <cstdlib>
#include <iostream>
#include "dcuda_host.h"
#include "dcuda_device.cu.h"
#define THREADS_PER_BLOCK 128
struct user_data
{
   double *data;
   double *buffer;
   bool is_leader;
   double result;
};

int main(int argc, char **argv){
   //dcuda initialization
   dcuda_host_context_handle dcuda_host;
   dcuda_host_init(argc, argv, &dcuda_host, kernel, THREADS_PER_BLOCK);
   user_data userdata;

   //rank information
   dcuda_rank_info rank_info;
   dcuda_host_get_rank_info(dcuda_host, &rank_info);
   int local_comm_size = rank_info.rank_responsible;
   int world_comm_size = rank_info.rank_count;
   userdata.is_leader = (rank_info.node_index == 0 &&
            rank_info.device_index == 0);

   //data allocation
   int size = local_comm_size * THREADS_PER_BLOCK;
   cudaMalloc(&userdata.data, 2 * size * sizeof(double));
   userdata.buffer = &userdata.data[size];

   //data initialization
   double *init_data = (double*) malloc(size * sizeof(double));
   for (int i = 0; i < size; i++){
      init_data[i] = std::rand();
   }

   //copy data to device
   cudaMemcpy(userdata.data, init_data, size * sizeof(double),
               cudaMemcpyHostToDevice);

   //run dcuda kernel
   dcuda_host_run(dcuda_host, &userdata, sizeof(userdata));

   //report result
   if(userdata.is_leader){
      double time = dcuda_host_get_timings(dcuda_host)[0];
      std::cout << "Sum = " << userdata.result << std::endl;
      std::cout << "Computation Time: " << time << " ms" << std::endl;
   }

   //cleanup
   free(init_data);
   cudaFree(userdata.data);
   dcuda_host_finish(dcuda_host);
   return 0;
}
```

Listing 3.1: Host Example Code: dCUDA initialisation and kernel invocation.

device code for an example on how to start/stop a timer.

We finish our code example by deallocating the memory used and clean up dCUDA by calling `dcuda_host_finish()` on lines 54 to 57.

3.2.2 Device Code

The device code running on the GPU does the actual work. The kernel code showed in Listing 3.2, executes when the function `dcuda_host_run()` is called. dCUDA kernel code is always simultaneously executed by many threads in parallel, like in the CUDA programming model. In the following, we describe the kernel example code step by step to see the important features of dCUDA in action.

At the beginning of every dCUDA kernel we have to initialize the dCUDA device library (line 6). First, we declare a `dcuda_state` in shared memory (indicated with the keyword `__shared__`). It is important to note that the `dcuda_state` must be in shared memory as it holds all dCUDA internal data private to each thread block and shared by the threads of the thread block. Next, we initialize the `dcuda_state` by calling the function `dcuda_gpu_init()`. This function must be called prior to any other dCUDA call.

After the initialisation, we query the size of the two default communicators and what rank index we have in the corresponding communicators on line 8 to 11. For now, the communicators `DCUDA_COMM_WORLD` and `DCUDA_COMM_DEVICE` are implemented, which contain all ranks in the whole system and all ranks of the local device respectively. The rank index in the device communicator is always equal to the block index of that thread block (`blockIdx.x`) and the size of the device communicator is equal to the CUDA grid size (`gridDim.x`). The query `dcuda_comm_rank()` returns a different result for each thread block. The rank in the world communicator is unique for each thread block across the whole system.

In the host code in the Listing 3.1 we stored the important information in the struct *user_data*, including a pointer to our data array.To access this data on the device, we call `dcuda_get_data()` on line 13, which gives us a pointer to the user data.

Before starting the algorithm, we have to create a memory window, which is used for communication with other ranks. `dcuda_win_create()` has to be called by all ranks in the communicator collectively like we do on lines 17,18. The window base in memory and window size can be different for each rank and ranks that do not participate in the communication can set the size to zero. Source and destination buffers of RMA operations have to reside inside of the created window. In our example, we create only one window in the world communicator. A window in the world communicator can also be used for local communication but a window in the device communicator can not be used for global communication. We create a window that spans both arrays: the *data* array holding our data, as well as the *buffer* array used as a receive buffer. Note that we allocated these arrays in consecutive memory in the host code.

Before starting the actual algorithm, we start a timer to measure the execution time of the algorithm on line 21 (`dcuda_start_cpu_timer()`). At the end of the algorithm on line 48 we stop the timer. The host can later query the timing result with a call to `dcuda_host_get_timings()`.

```
__global__ void kernel(dcuda_context_handle dcuda)
{

   int size, rank, local_size, local_rank;
   __shared__ dcuda_state gst;
   dcuda_gpu_init(dcuda, gst);
   dcuda_log log = dcuda_log_get(gst);
   dcuda_comm_size(gst, DCUDA_COMM_WORLD, &size);
   dcuda_comm_rank(gst, DCUDA_COMM_WORLD, &rank);
   dcuda_comm_size(gst, DCUDA_COMM_DEVICE, &local_size);
   dcuda_comm_rank(gst, DCUDA_COMM_DEVICE, &local_rank);
   user_data *data;
   dcuda_get_data(gst, (void**)&data);

   //create windows
   dcuda_win win;
   dcuda_win_create(gst, DCUDA_COMM_WORLD, data->data,
      2 * local_size * blockDim.x * sizeof(double), &win);

   //timing
   dcuda_start_cpu_timer(gst);

   //block reduction
   if (threadIdx.x == 0){
      for (int i = 1; i < blockDim.x; i++)
         data->data[local_rank * blockDim.x] +=
            data->data[local_rank * blockDim.x + i];
      }
   }

   //global reduction
   int offset = (local_size * blockDim.x + rank) * sizeof(double);
   for (int step = 1, round = 0; step < size; step *= 2, round++)
   {
      int range = 2 * step;
      bool send = rank % range == step;
      bool recv = (rank % range == 0) && (rank + step < size);
      if(send){
         dcuda_put_notify(gst, win, rank - step, offset, sizeof(double),
                 &data->data[local_rank*blockDim.x], round % 256);
      }
      if(recv){
         dcuda_wait_notifications(gst, round % 256, 1);
         data->data[local_rank * blockDim.x] += data->buffer[rank+step];
      }
   }

   dcuda_stop_cpu_timer(gst);
   if(rank == 0){
      data->result = data->data[0];
      log << "result = " << data->result << log.flush;
   }

   //cleanup
   dcuda_win_free(gst, win);
   dcuda_gpu_finish(gst);
}
```

Listing 3.2: Device Example Code: Distributed reduction

At the start of our algorithm each thread owns a single value located at `data->data[local_rank * blockDim.x + threadIdx.x]`. The algorithm sums all initial values of the data array. In the end, thread 0 of rank 0 will have the sum of all values stored in its memory location. We achieve this by doing a reduction on a binomial tree in two phases. In the first phase, each rank sums up the values of all its threads on lines 24 to 29. This phase does not need dCUDA specific functions and we use a simple but inefficient approach to do it. We use the threads in each block with index 0 to sum up all values of the block. A more efficient variant can be found in the Nvidia developer blogs [14]. At the end of this phase, the first thread of each rank has the sum of all the values in his block.

The lines 31 to 46 show the second phase of the algorithm. The phase has a logarithmic number of steps. In the first step, all ranks with odd rank index send their value to their neighbour with smaller rank index. The ranks with even index wait for the value and add it to their own value. In the next round, the ranks that have sent a value remain inactive. Half of the ranks that received a value in the previous round will now become senders and the other half remain receivers. After $\log n$ rounds rank 0 is the only active rank left and it will hold the result of the reduction.

Now, we look into how data can be sent with dCUDA. With `dcuda_put_notify()` we can take memory from our window and put it into the window of a remote rank. The remote rank is notified once the data has arrived. On line 39 we pass the following arguments to the function: the state of the rank, the memory window to use, the target rank index, the offset from the target window base where the data will be written, the size of the data, the pointer to the local data buffer (which has to be inside the specified window), and a notification tag. The offset in the example equals the location `data->buffer[source_rank]` in the target window. Note that the tag value has to be positive and less than 256. Section 3.3.3 discusses the design decision to limit the tag size to 8 bits. dCUDA automatically detects if the target rank is located on the same device or on a remote device and acts accordingly.

dCUDA uses RMA to send data, which means the sender can modify remote memory directly. Thanks to the dCUDAs notification system, the receiver is notified whenever its memory was modified and new data is ready to use. If a rank needs data from another rank, it can wait for incoming notifications by using `dcuda_wait_notifications()`. In our example we wait at line 43 for a single notification with the tag `round % 256`. Note that the function can also be used to wait for multiple notifications with the same tag, e.g. a rank waits for notifications from multiple sources. The function blocks until enough notifications have arrived or returns immediately if enough notifications have been received previously. We know that after receiving the notification, the corresponding data is ready. Therefore, the receiver rank can add the received value to its own value on line 44.

Finally, we print the result directly to the console on line 51 with the help of the *dcuda_log* object. Then, we free the dCUDA window and finish the kernel by calling `dcuda_gpu_finish()` to clean up dCUDA resources on lines 55 and 56.

It has to be mentioned that the code in Listing 3.2 does not yield optimal performance but it shows the basic building blocks of the dCUDA programming model. The code has more remote accesses and memory copies than necessary. An optimised version would divide the last phase into two phases: one that reduces the result on the local device without the use of memory copies (dCUDA provides the function `dcuda_notify()` for that case). In the last phase,

only one rank per device is active and a minimal number of remote accesses are performed to get the final result.

3.3 dCUDA Interface

The host API is mostly used to initialize dCUDA and launch the dCUDA kernel. The device API provides functions to create memory windows and several functions to perform RMA operations and synchronisation mechanisms.

3.3.1 Host API

dCUDA simultaneously launches multiple processes on multiple nodes in the system. One process manages exactly one GPU device. Nodes with more than one GPU device can launch up to as many processes as they own. The initialisation of the dCUDA host sets up and prepares all communication channels needed by dCUDA. Also, the dCUDA kernel is prepared for launch.

Listing 3.3 shows the interface of the host. The type *dcuda_host_context_handle* points to the dCUDA context, which holds all resources needed by the host to perform any dCUDA related operation. All dCUDA host functions require a context handle to be passed as an argument. The type *dcuda_context_handle* is a pointer to the dCUDA resources needed by the device. The type *dcuda_kernel_t* defines the signature of the dCUDA kernels that the users implement. Any dCUDA kernel must have a single parameter, which is a dCUDA device context.

dcuda_host_init initialises dCUDA. The program parameters, which include dCUDA run information, are passed down to dCUDA with the first two parameters of *dcuda_host_init*. The function outputs a *dcuda_host_context_handle* in the third parameter, which has to be passed to any subsequent dCUDA call. Additionally, the function requires a pointer to the dCUDA kernel and the number of threads per rank to properly prepare the launch of the kernel.

dcuda_host_get_rank_info can be used to query the launch configurations of dCUDA. It returns a *dcuda_rank_info* struct that holds all relevant information that the user needs to prepare the data for the kernel launch. Lines 7 to 15 in the Listing 3.3 show all fields of the rank info struct.

dcuda_host_run launches the dCUDA kernel previously passed to dCUDA. Optionally, one can pass a pointer to some user data to the function. The data will be copied to the device where it can be accessed with the device function *dcuda_get_data*. The function will copy the data back from GPU memory to the same host memory after the kernel execution. Therefore, all the changes made by the device will be visible to the host. The function blocks during kernel execution and it returns the total execution time in milliseconds of the dCUDA kernel.

dcuda_host_get_timings returns a vector containing the timing information collected using the device functions *dcuda_start_cpu_timer* and *dcuda_stop_cpu timer*.

dcuda_host_finish has to be called at the end of the program to clean up all resources and tear down open connections. A dCUDA host context can not be used for any dCUDA calls after a call to *dcuda_host_finish*.

```
typedef struct dcuda_host_context* dcuda_host_context_handle;
typedef struct dcuda_context* dcuda_context_handle;
typedef void (*dcuda_kernel_t)(dcuda_context_handle);

struct dcuda_rank_info
{
    int rank_count;        ///< total number of ranks
    int rank_responsible; ///< number of ranks I own
    int rank_start;        ///< index of first rank I own
    int device_count;      ///< number of active devices on this compute node
    int device_index;      ///< my device index on the node
    int node_count;        ///< number of compute nodes
    int node_index;        ///< index of my node
    int process_count;     ///< number of host processes
    int process_index;     ///< index of my process
};


void dcuda_host_init(int argc, char *argv[],
                      dcuda_host_context_handle *ctx,
                      dcuda_kernel_t kernel,
                      int threads_per_rank);

void dcuda_host_get_rank_info(dcuda_host_context_handle ctx,
                               dcuda_rank_info *rank_info);

double dcuda_host_run(dcuda_host_context_handle ctx,
                     void    *gpu_data = NULL,
                     size_t gpu_size  = 0);

vector<double> dcuda_host_get_timings(dcuda_host_context_handle ctx);

void dcuda_host_finish(dcuda_host_context_handle ctx);
```

Listing 3.3: DCuda Host API

3.3.2 Device API

The device API can only be used within dCUDA kernel code. Most of the functions must be called collectively by all threads in the same rank with some exceptions. Not following the dCUDA calling conventions can lead to deadlocks or undefined behaviour.

Listing 3.4 shows the dCUDA library function to initialize the dCUDA ranks as well as some timing functions. dCUDA passes a *dcuda_context_handle* as an argument to the kernel it

```
//Types
typedef struct dcuda_context* dcuda_context_handle;
struct dcuda_state;

//Constants
using dcuda_comm = int;
#define DCUDA_COMM_WORLD 0
#define DCUDA_COMM_DEVICE 1

//Functions
__device__ void dcuda_gpu_init(dcuda_context_handle dcuda,
                                dcuda_state& gst);

__device__ void dcuda_gpu_finish(dcuda_state& gst);

__device__ void dcuda_get_data(dcuda_state& gst,
                                void **gpu_data);

__device__ void dcuda_comm_size(dcuda_state& gst,
                                 dcuda_comm comm, int *size);

__device__ void dcuda_comm_rank(dcuda_state& gst,
                                 dcuda_comm comm, int *rank);

__device__ void dcuda_start_cpu_timer(dcuda_state& gst);

__device__ void dcuda_stop_cpu_timer(dcuda_state& gst);
```

Listing 3.4: dCUDA device API, initialisation and timing

launches. The dCUDA context stores all necessary information to initialise the individual ranks on the device. The struct *dcuda_state* stores the state of the individual ranks and has to be stored in rank local CUDA shared memory. All dCUDA device functions require a *dcuda_state* as a library handle. The constants *DCUDA_COMM_WORLD* and *DCUDA_COMM_DEVICE* define the two exclusive dCUDA communicators.

dcuda_gpu_init must be called before any other call to the dCUDA device library. It initialises the rank local *dcuda_state*, which must be stored in CUDA shared memory. The initialisation requires the *dcuda_context_handle* that is passed to the kernel as the only argument. This function has to be called by all dCUDA ranks and all threads within a rank. It synchronises all dCUDA ranks in the system with a global barrier.

dcuda_gpu_finish frees all dCUDA resources on the device. After calling *dcuda_gpu_finish* no library functions can be called. This function has to be called by all ranks in the system. It performs a global barrier.

dcuda_get_data returns a pointer to the user data in the second parameter. It holds custom data passed to dCUDA with the call to the host function *dcuda_host_run*. After the execution

of the kernel, dCUDA copies the user data back to the host, making all the changes made to the user data visible on the host.

dcuda_comm_size takes a communicator in the second parameter and returns the size of the communicator in the last parameter. Calling this function and passing the communicator *DCUDA_COMM_DEVICE* will return the number of thread blocks on this device, while it will return the total number of ranks in the system if the world communicator is passed. This function can be called by individual threads without synchronisation.

dcuda_comm_rank returns the rank index at the given communicator. A rank in the device communicator returns the block index (`blockIdx.x`),while a rank in the world communicator returns `rank_start + blockIdx.x`. This function can be called by individual threads without synchronisation.

```
__device__ void dcuda_comm_barrier(dcuda_state& gst,
                                    dcuda_comm comm);

__device__ void dcuda_win_create(dcuda_state& gst,
                                  dcuda_comm comm, void *winbase,
                                  int size, dcuda_win *win);

__device__ void dcuda_win_free(dcuda_state& gst,
                                dcuda_win win);

__device__ void dcuda_notify(dcuda_state& gst,
                              int rank, dcuda_comm comm,
                              int tag);

__device__ void dcuda_put(dcuda_state& gst,
                           dcuda_win win, int rank,
                           int offset, int size,
                           void *buffer);

__device__ void dcuda_put_notify(dcuda_state& gst,
                                  dcuda_win win, int rank,
                                  int offset, int size,
                                  void *buffer, int tag);

__device__ void dcuda_win_flush(dcuda_state& gst,
                                 dcuda_win win);

__device__ int dcuda_test_notifications(dcuda_state& gst,
                                         int tag, int count);

__device__ void dcuda_wait_notifications(dcuda_state& gst,
                                          int tag, int count);
```

Listing 3.5: dCUDA Device API: Notified RMA

dcuda_start_cpu_timer, dcuda_stop_cpu_timer starts, respectively stops a timer. The actual measurements are done by the CPU, making it independent from GPU clock rate changes.

Both functions send a message to the host causing it to take the current CPU time. Whenever the function *dcuda_stop_cpu_timer* is called, the host stores the difference in time since the last call to *dcuda_start_cpu_timer*. The host function *dcuda_host_get_timings* returns all timing results. Both functions need to be called collectively by all ranks of the device.

Listing 3.5 shows relevant functions for communication and notified remote access. Before ranks can send data to each other, they have to create a memory window in a communicator, exposing memory to remote access. The window object can then be used to put data to remote memory regions.

dcuda_comm_barrier synchronises all ranks of that communicator. The function has to be called by all ranks in the communicator. Otherwise, some ranks might deadlock and never return from the function. The function guarantees that no thread will leave the barrier before all threads in the communicator have entered the corresponding barrier call.

dcuda_win_create creates a memory window for RMA in a communicator and returns a *dcuda_win* object as the last parameter. The function has to be called by all ranks in the communicator that is given to the function and synchronises them. Each window is associated with a communicator and can only be used for communication within this communicator. The *winbase* points to the start of the memory region in GPU global memory that a rank wants to expose to remote accesses and *size* is the number of bytes of that memory region. Windows of ranks on the same device may or may not overlap. If memory windows do overlap and a remote rank writes to that region, only the target rank is notified. If window creation fails, this function returns `DCUDA_WIN_NONE`. Note that windows are only used for memory accesses but not by the notification system. Notifications are not associated with a window but rather with a tag value.

dcuda_win_free destroys the window object created earlier. It has to be called by all ranks in the communicator associated with this window object. It synchronises all the calling ranks.

dcuda_notify is called by a single rank and sends a notification to some other rank with a tag. The tag has to have a value between 0 and 255. The parameter *rank* defines the target rank index of the notification in the communicator specified with the parameter *comm*. Note that ranks have different rank indices in different communicators.

dcuda_put accesses and writes memory to the memory window of another rank. The parameter *win* takes a window object created earlier with *dcuda_win_create* and the parameter *rank* takes the rank index of the target rank in the communicator associated with the window object. The memory will be written to the memory starting at `winbase + offset`, whereas winbase corresponds to the memory location that was given at window creation by the target rank. The parameter *size* says how many bytes will be copied and *buffer* points to the send buffer of the calling rank. Note that the target rank will not be notified by any memory accesses

by calling this function. The target rank has to be notified eventually with a call to *dcuda_notify* or *dcuda_put_notify*.

dcuda_put_notify combines the functions *dcuda_put* and *dcuda_notify* in a single message. It includes the parameters of both functions and the target rank will be notified as soon as the data is ready to use.

dcuda_win_flush blocks the calling rank until all previous outgoing messages associated with a window object are completed. The *dcuda_put* functions return immediately after issuing a send request. Modifying the send buffer after a put is unsafe and can alter the data that is sent to the target. Therefore, a rank can call *dcuda_win_flush* before modifying the send buffers, which guarantees that all put request are completed. The function *dcuda_notify* does not affect this function as no send buffers are involved.

dcuda_test_notifications verifies if one or more notifications with a specific tag have been received. The function tries to find *count* many notifications with the tag *tag*. The function consumes the notification and returns the value 1 if there are enough notifications ready with the right tag. If there are not enough notifications, it returns with the value 0 without consuming any notifications. Note that we cannot test for notifications from a specific source rank. The programmer can encode the source information into the tag if needed.

dcuda_wait_notifications is a blocking variant of the function *dcuda_test_notifications*. If the rank does not have enough notifications with the right tag, it blocks until at least *count* notifications with the right tag are ready for consumption. This function has the same behaviour as a call to *dcuda_test_notificcations* that returns 1.

3.3.3 Interface Changes

The interface of dCUDA as described here in the thesis went through several changes in respect to the initial interface [8]. We extended the interface to have a broader range of functions. This allows the user to create programs with more fain grained and tuned communication patterns. On the other hand, some functions were simplified to improve the performance of dCUDA by reducing library overhead and reducing the communication latency.

CPU timing

Support for CPU timing was introduced in this thesis with the functions *dcuda_start_cpu_timing* and *dcuda_stop_cpu_timing*. The functions use the infrastructure provided by dCUDA to send a message to the CPU, starting or stopping a timer on the CPU.

A programmer who wants to measure the runtime of his applications can either use the CUDA functions *clock* and *clock64* to get the current 32 bit or 64 bit clock counter respectively, or he

can use the dCUDA CPU timer functions. The result of the CUDA clock functions depend on the clock rate of the GPU shader clock, which may change during execution due to the Nvidia GPU boost technology. Furthermore, the programmer has to handle counter overflows to get an accurate timing result and measurements over a long period of time are difficult due to the counter overflows. The dCUDA CPU timing functions provide an alternative way to measure time periods, which is easy to use. To start and stop the timers, a message to the CPU is required, which needs around $2\mu s$. Therefore, we state that the precision of the function is in the range of less than $5\mu s$.

Put and Notify

The functions *dcuda_put* and *dcuda_notify* are new in the interface of dCUDA. Previously, the only way to communicate with other ranks were the *dcuda_put_notify* and *dcuda_get_notify* functions, which access remote memory and notify the target rank. The new functions allow to access memory and notify a target rank individually, giving the programmer more freedom on how to design his applications. The notify function allows for fine-grained synchronisations of communication ranks without sending actual data and therefore causing less overhead. Additionally, a rank can notify other ranks without first creating a dCUDA window.

The put function does not notify the target rank. It is useful in case one wants to send multiple non-contiguous memory blocks. The sender might want to notify the target only when all the memory blocks have been transferred. By using the put function and only at the end calling put_notify, the overhead of delivering a notification is reduced to a minimum. Note that a put_notify is conceptually the same as a put followed by a notify to the same rank, but put_notify yields less overhead as only one message is sent instead of two. All put and notify messages to the same remote rank are guaranteed to be received in the same order they were sent.

Get

Support for remote memory read was removed from the interface of dCUDA. We found that reading memory from remote memory is rarely useful in HPC applications. It seems more natural to send data to all remote ranks that need the data when it is ready instead of waiting for remote ranks to get it themselves. Performance critical applications should prefer using *put* over *get* functions, as reading remote memory adds an additional half of the round trip time to the latency. Additionally, the function *get_notify* does not suit the new implementation design of dCUDA with InfiniBand as there is no function that reads remote data and notifies the target in the InfiniBand Verbs API.

Wait and Test Notifications

The notification system was overhauled in the new implementation to reduce library overhead. Previously notification matching was a computationally expensive operation. Each rank owned a notification queue for each window. The user could search and wait for specific notifications by matching a tag, the source rank, and the window. One could provide wildcards for tag and

source rank.

The dCUDA interface of the current implementation is much simpler. First of all, notifications have no relation with dCUDA windows any more. In fact, by using the new interface, one can send notifications without creating a single window. The only thing that windows do represent is the memory. Therefore, windows are only relevant for remote memory accesses but not for the notification logic. Furthermore, the notification system does not distinguish notifications from different source ranks. In consequence, we can not match notifications from a specific source anymore. If the user wants to wait for a notification from a specific source, he can encode this information into the tag value and wait for a specific tag.

The tag value of a notification is the only parameter for notification matching. We simplify notification matching even further by limiting the size of the tag value to 8 bits. This means that the tag value has to be between 0 and 255. These interface simplifications open up room for performance optimisations that we will discuss in detail in Chapter 4 where we discuss the implementation of dCUDA.

The new notification matching is less powerful since we can only distinguish 256 different notification tags which also have to encode the source rank. We argue here that most, if not all applications, can be designed in such a way, so that 256 tags are sufficient to guarantee correct synchronisation. Suppose we have more than 256 ranks, then an application that uses all-to-one communication patterns can not distinguish between all the notifications from the different ranks. This however, is not optimal as the receiver can not wait for a message from a specific source rank but rather has to wait for groups of notifications. In such a case, more tag values would be beneficial, but we argue that such an application design is not optimal in the first place. All-to-one communication patterns do not distribute the work evenly on all ranks and one single rank has more work to do than the rest. It is better to use a reduction tree in order to collect and reduce data at a single rank. Reduction trees better distribute the work and less than $\log n$ communication partners per rank are needed. This means that with 256 tags we can efficiently serve up to 2^{256} ranks without using the same tag twice. The same can be done with all-to-all communication patterns.

We showed that a rank in a carefully designed application does only communicate with $\log n$ other ranks to perform a collective operation involving all ranks. But successive collective operations that can overlap in time can involve different communication partners needing another set of $\log n$ tag values. We argue that a carefully designed application synchronises its ranks such that only a constant number of collective operations do overlap in any point in time, thus, tags can be reused. Usually, applications always need to be loosely synchronised because of data dependencies in the applications. Therefore, $c \log n$ tag values are sufficient to design most applications, as in most cases c is sufficiently small.

4

Implementation

The dCUDA framework can be separated into different modules that interact with each other. The device library interacts with the host runtime system, which uses a network manager to communicate with remote nodes over the network. We will present an overview over all parts of the dCUDA library including host and device code. Then we will cover device and host interaction, the MPI interface, the InfiniBand verbs interface, and the notifications logic in more detail, focusing on the specific implementation details added in the course of this work. Finally, we will present prerequisites and how to deploy dCUDA applications and discuss portability.

4.1 Architecture Overview

The architecture of dCUDA is divided into two parts. The device-side library and the host-side runtime system that manages remote communication and kernel setup among other things. We create a device-side instance for each rank on a device. The instances have to be stored in rank local CUDA *shared* memory and provide all the device library functions to communicate and synchronise with other ranks in the system. Device local communication works without any host interaction, but window creation and communication with remote ranks located on other devices requires the help of the host runtime system.

The system instantiates one host runtime instance per device. Computing nodes that manage multiple GPU devices can launch multiple dCUDA processes, each having a single host runtime instance that manages one GPU device. The host runtime initialises the kernels and manages any non-device-local communication. dCUDA implements two network communication managers. The first communication manager uses the InfiniBand Verbs interface for fast and low latency communication among all processes. InfiniBand benefits from Nvidia's GPUDirect RDMA technology to directly send data from and to GPU memory without stating

memory through the host. The second communication implementation is MPI based and uses a CUDA-aware MPI implementation and employs GPUDirect RDMA technology as well. Before compiling a dCUDA application, one can choose which network communication manager to use. The InfiniBand requires the system to have an InfiniBand network, while the MPI variant has no restrictions on the type of network.

Figure 4.1 shows an overview of the dCUDA architecture. A single host process and a single device can be seen. Multiple processes communicate over the network not depicted in the diagram. The upper part of the diagram illustrates the device library instances for each thread block, while the lower part shows the host runtime system that manages remote communication with other devices. The user application code runs on multiple CUDA thread blocks, each having an instance of the device library that is used to communicate with other local and remote ranks. Ranks on the same device can communicate with each other directly through device global memory without any host interaction.

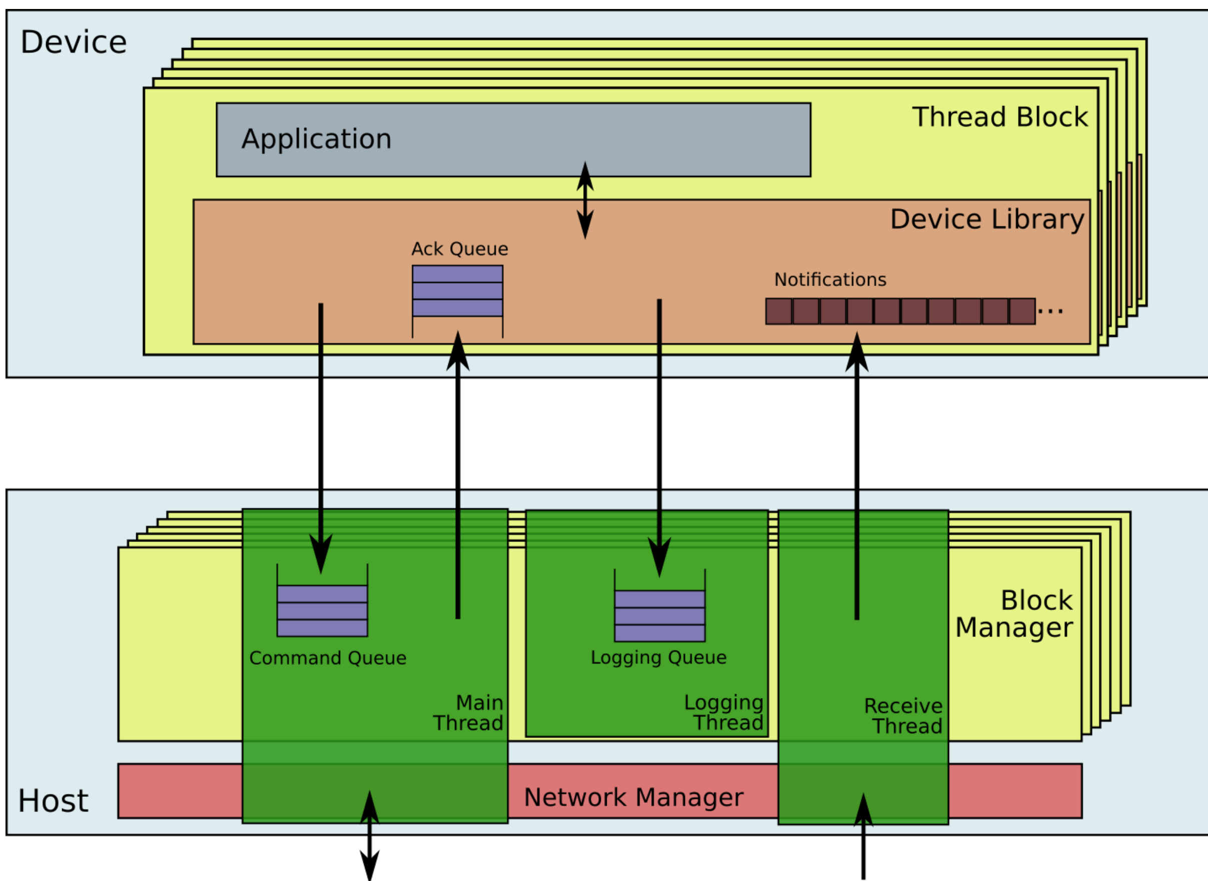


Figure 4.1: dCUDA architecture overview: Device library instances with corresponding block managers in the host runtime system.

Communication with remote ranks needs the help of the host runtime system. As can be seen on the left of the Figure 4.1, the device library sends commands to its block manager on the host. The host runtime system maintains an instance of the block manager for each thread block running on the device and holds all communication queues and configuration information of its corresponding thread block. The main thread of the runtime system manages all the command queues from all block managers and handles the incoming commands. Some commands like

window creation or barriers require an acknowledgement, which is sent back to the device into the Ack queue. Most of the requests require the host to send messages to remote ranks using the network manager. Depending on the dCUDA settings, the network manager is a CUDA-aware MPI implementation or a network interface based on InfiniBand verbs. RMA command requests do not contain the actual data. Only the relevant meta data is sent to the host such that the host can issue the send process, whereas the actual data is directly copied to the network adapter using GPUDirect RDMA technology [19].

Incoming RMA commands from remote processes are received and processed by the receive thread of the runtime system as depicted on the right side of Figure 4.1. The data is written directly to device memory with GPUDirect RDMA. The host receiver thread gets the metadata including the notification tag required to notify the destination rank. Each rank on the device owns a simple data structure holding counters for the received notifications. The receiver thread updates this data structure whenever a notification arrives. A GPU rank waiting for a specific notification can peek on the notification data structure and gets informed by any update that the host issues when notifications have arrived.

Native CUDA comes with very limited logging support. It implements the *printf* function in kernel code to write to the console. The CUDA runtime buffers the strings until the kernel finished execution. Only then the printed data gets visible in the console. This is a viable solution for most CUDA programs, as usually multiple short kernels are started one after another. dCUDA starts only a single large kernel that only terminates at the end of the program. In consequence, it is very hard to debug a dCUDA kernel with printf statements, especially if some ranks get stuck because of bugged synchronisation in the application. Therefore, dCUDA includes its own logging infrastructure. Similarly to the command queue, each block manager has a logging queue. By using the *dcuda_log* object, messages are sent to the host, which prints it directly to the console, while the kernel is still running. This is depicted in the center of Figure 4.1. The logging thread on the host manages all the logs from all the ranks.

A new optimisation of dCUDA is the introduction of a multi-threaded host runtime system that distributes the work on multiple threads as illustrated in Figure 4.1. When the host launches a kernel on the device, it also launches two additional host threads to divide the work on more CPU cores for better performance. We call the first spawned thread *receive-thread*. Its job is to manage any incoming RMA requests from other processes and to update the notification data structures of the device ranks. Since receiving messages is the only duty of the thread, no other tasks can hold the thread back from processing incoming messages, which lowers the latency of remote communication compared to a single-threaded approach. The lifetime of this thread is as long as the lifetime of the kernel. Without a running kernel, no RMA requests will arrive at the host and therefore, the thread has no work to do.

The second thread is called the *logging thread*. Its only job is to manage device logging. Whenever a device rank uses the dCUDA logging system, a message is sent to the host over a logging queue. The logging thread will poll that queue and print the logs to standard output. The purpose of the logging thread is to reduce the overhead of extensive logging by taking some work from the main thread.

The tasks of the main thread are to wait for commands from the devices, including window creation, outgoing RMA commands and global barriers. Communication with the device ranks is achieved by using two queues per rank, one for each direction.

4.2 Network Manager

The network manager is used to communicate with other dCUDA processes in the system. Processes are distributed over multiple nodes connected by some interconnect. If a node has multiple GPU devices installed, multiple dCUDA processes need to be started to manage all the devices. Communication between local processes uses the same network interface as inter-node communication. The performance of node local communication could be improved by copying data directly from device to device using GPUDirect peer to peer technology. In this thesis we focus more on inter-node communication performance instead and leave intra-node communication for future works.

dCUDA provides two different network manager versions. First, there is the MPI version where all networking is done by MPI calls [1, 8]. The MPI version has the advantage that it can be employed on all systems that have a CUDA-aware MPI implementation installed and has no further hardware requirements on the network. Secondly, there is a version using a network manager based on the InfiniBand Verbs API. This network manager was implemented as part of this work and stands out for its low communication overhead. Because InfiniBand Verbs are used, this network manager can only be used in a system connected by an InfiniBand network. The MPI network manager can be enabled or disabled with the *cmake* option flag *USE_MPI*. The InfiniBand network manager is enabled by default if MPI is disabled. See Section 4.6 on how to build dCUDA projects. Following, we will describe each network manager in more detail.

InfiniBand Verbs Communicator

InfiniBand does not provide a high level interface like MPI does, but instead provides a low level interface, which gives the programmer a lot of freedom. Because of the low level nature of the interface, we implemented a network communicator based on the InfiniBand Verbs interface, which has a similar functionality to MPI. The interface can be found in the dCUDA source directories in the file *ib_communicator.h*. Following, we expect the reader to have read Section 2.4 for some background information about the InfiniBand Verbs interface.

Creating an *ib_communicator* object and calling *init* on it will initialise all InfiniBand resources like QPs and CQs. It also allocates and registers a buffer in host memory for RDMA over InfiniBand. Each process has to know the leader process at start-up. The network manager will then connect to the leader process using TCP sockets. The leader gathers all relevant information that is needed for communication using RDMA. This includes the *lid* number that uniquely identifies an InfiniBand HCA in a network, a QP number, a buffer address and the corresponding remote key. When the leader process has received this information from all the remote processes, it scatters this information to all of the other processes, such that every process has the ability to set-up a channel to any other process. Then the TCP connections are closed as they are not needed anymore. Any further communication will use low latency InfiniBand RDMA.

We chose to use the dynamic connection (DC) transport protocol for the implementation of our network manager. DC transport scales well on large compute clusters, as a single DC QP is

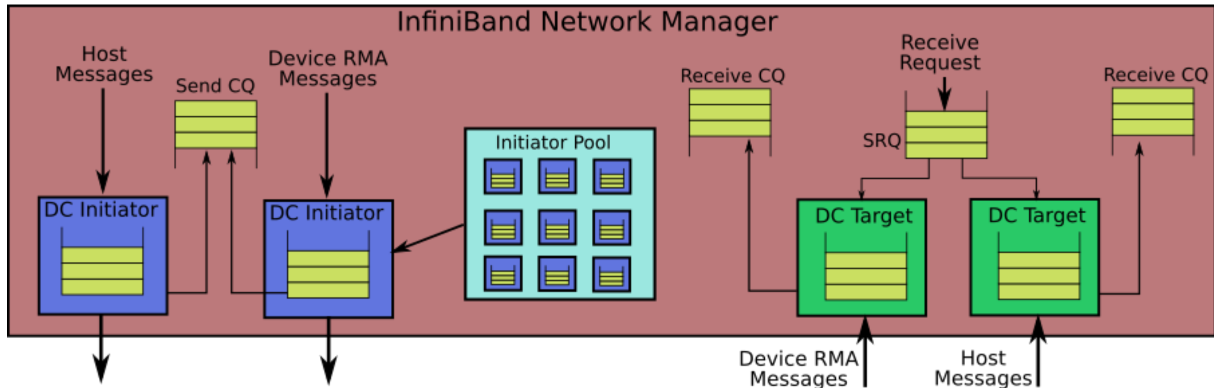


Figure 4.2: InfiniBand Network Manager

able to communicate with any number of remote nodes while providing reliable connections and performance comparable to RC transport [25].

In the initialisation phase, we create two DC targets acting as receive queues. One is exclusively used for RDMA from and into host memory, while the second one is used for RDMA on device memory. The former is used to exchange information for window creation or to implement a global barrier, which is done by the main thread. The second DC target is only operated by the receive thread to receive dCUDA RMA notifications. If only one DC target was used, the system would suffer from race conditions as two threads would poll the same queue concurrently. Figure 4.2 illustrates the InfiniBand network manager and its resources. On the right side we see the two DC targets. In order to receive messages, the network manager posts receive requests to a SRQ (Shared Receive Queue), which are forwarded to the corresponding DC targets. Whenever a message arrives at a target, a receive request is consumed and a completion entry with the notification tag is created once the operation has finished. One can poll the receive CQ to get notified by completed receive operations. Note that *put* operations without notification do not consume a receive request and do not generate a completion entry.

The network manager creates multiple DC initiator objects used to initiate memory transfer to a remote DC target. The first DC initiator is operated by the main thread and used for RMDA from and to host memory only. The main thread uses it for window creation and global barriers. Additionally, up to 10 initiators are created and added to a pool (this can be changed in the *dcuda_constants.h* file). These initiators are all used for dCUDA RMA operations on device memory. We decided to use a pool instead of a single initiator as H. Subramoni et al. [25] showed that in a one-to-all communication pattern, multiple initiators outperform a single initiator and proposed the usage of a pool similar to the one we implemented. They also showed that in an all-to-one communication pattern, the performance does not benefit from multiple DC targets. Therefore, we use a single DC target for all incoming dCUDA RMA operations.

We recap here in short the arguments for picking a pool of DC initiators, as proposed by H. Subramoni et al. [25], instead of a single initiator. Suppose process A wants to send 2 packets to process B, followed by a packet to process C using a single DC initiator. The HCA of process A will first send a connect message to HCA B immediately followed by the two actual packets. Then HCA A has to wait for an Ack message for each packet before it can close the connection because DC transport is a reliable transport type. When the Ack messages finally arrive, a disconnect message is sent to HCA B followed by a connect to HCA C and the actual data for

process C. We see that the HCA has to wait a round trip time each time it wants to change the connection to another HCA.

Having the communication model of DC transport in mind, we designed a pool with the following behaviour. If a process wants to send a message to destination D, the pool will return the DC initiator that was used last time to send a message to destination D. If this is the first time we send something to destination D or if that initiator was used in the meantime to send a message to another destination, the pool will return a random initiator in a round-robin fashion. This pool will make sure that messages to the same destination use the same initiator, making it more likely that an open connection is reused. Listing 4.1 shows the algorithm that implements this behaviour. In the best case, we can reuse a connected initiator QP which requires two map lookups. If we can not reuse an open connection, the algorithm does two additional map lookups. Since the pool size is set to be constant, the asymptotic runtime of the algorithm remains constant.

```
1  vector<ibv_qp *> pool;
2  int iterator, pool_size;
3  map<ibv_qp *, int> last_connection; //maps QP to last connected destination
4  map<int, ibv_qp *> last_qp; //maps destinations to the QP used last time
5
6
7  ibv_qp* qp_pool::get_qp(int dest_process)
8  {
9     ibv_qp* qp = last_qp[dest]
10    if(qp == NULL || last_connection[qp] != dest_process){
11       qp = pool[iterator];
12       iterator = (iterator + 1) % pool_size;
13       last_connection[qp] = dest_process;
14       last_qp[dest_process] = qp;
15    }
16    return qp;
17 }
```

Listing 4.1: Initiator QP pool algorithm

The left side of Figure 4.2 illustrates the process of sending a RMA message. We pick a DC initiator from the pool and post a send request on the send queue of the initiator. The HCA processes all requests in the queues of the initiators and whenever a request has completed, a completion entry is posted to the send CQ. All completion entries from the initiator are posted on the same CQ. Therefore, we can poll the CQ to check whether the data has been sent and update the corresponding flush counters, which keep track of the oldest pending send requests. Host messages that do not involve any device memory are sent over a separate DC initiator, which makes use of the same CQ as the other initiators.

MPI Communication

dCUDA delivers an alternative to the InfiniBand Verbs network manager by using MPI. To switch from the default InfiniBand network manager to MPI, one can toggle a *cmake* option switch called `USE_MPI` using *ccmake*. Networking in dCUDA using MPI was well described in dCUDA's original paper [8]. Therefore we will recap only the most important aspects here.

There are two kinds of commands from GPU ranks that are processed by the host runtime system. First, there are notifications and remote memory accesses that are sent from one rank to another single remote rank. Then, there are commands involving all ranks in the system like window creation and destruction and global barriers. In case of a collective operation like the latter kind of commands are, the host waits until all the local ranks issued the request before doing anything. Then it calls MPI collective functions like *MPI_Allgather* to exchange window informations or *MPI_Barrier* for global barriers.

RMA requests are processed differently and require two MPI messages. The RMA command messages coming from the local source rank include all meta data like a pointer to the source buffer in device memory, target rank, target window and window offset. The host first sends a message containing meta information needed by the remote host to initiate the actual data transfer using the non-blocking function *MPI_Isend*. Essentially, it contains the target rank, window index, and window offset. Immediately after the first message, the host sends a second message containing the actual data using *MPI_Isend*. MPI detects by itself that the data is located in device memory and reads it directly from there with GPUDirect RDMA. Both this messages are non-blocking and the host main thread can proceed and process other requests while the network is sending the data. As long as the send operation is not complete, it is not safe to modify the data buffer in GPU memory. Therefore, the main thread is regularly checking whether the MPI requests have finished using *MPI_Test*. If so, it updates a *flush counter* in GPU memory to inform the rank that issued the RMA request that is has completed and the buffer is safe to be modified again.

The reception of RMA requests is done by the host receive thread. When the receive thread is started, it issues a *MPI_Irecv* request from any source to receive the meta information for the actual RMA operation. The receive thread regularly tests the MPI request and whenever a message has arrived, it calls *MPI_Irecv* again to receive the meta information of the next RMA request. When meta information arrived at the destination host, the memory transfer of the actual data has to be issued. The meta information includes enough information for the host to know where the data has to be written to in device memory. Therefore, the receive thread issues another *MPI_Irecv* and passes the pointer to the device buffer. Additionally, the receiver stores the meta information such that when the receive operation is completed, a notification can be sent to the destination rank to inform it of the memory access if it was a notified RMA request.

The dCUDA implementation is muti-threaded and two threads call MPI functions concurrently. Even through MPI is thread save if configured correctly, the MPI implementations are not performance optimised and a single threaded approach can yield better performance. Therefore, we added a switch to enable and disable multi-threading for dCUDA using MPI. To dissable multi-threading, one can set the macro `MPI_MULTITHREADED 0` in the file *dcuda_constants.h*.

4.3 Host-Device Communication

Low latency communication between the host runtime system and the GPU device is essential for good performance of the dCUDA library. For data transfers from host to device we use the low latency memory copy library *gdrcopy*, which uses GPUDirect RDMA for fast memory transfers (Details in Section 2.5). For transfers in the other direction, we map page-locked host

memory into the GPU address space such that the device can directly write to host memory.

Host and device communication mostly uses queues as shown in Figure 4.1 with the exception of notification updates. We briefly recap the queue design used as presented in the original paper [8]. The queues are located in the memory of the receiver to minimize poll overhead. The tail pointers are updated lazily such that optimally only the actual data is sent over PCI-e for each request.

As part of the dCUDA performance optimisation process, we removed the notification queues from dCUDA. The notification matching process was overhauled and the host can notify the ranks by directly updating a tag counter in the device memory using *gdrcopy*.

4.4 Notification Management

The dCUDA notification interface was simplified as described in Section 3.3, opening up opportunities to optimise the notification system and the notification matching process. Notifications are not stored in a queue anymore. Instead we keep track of the received notification using a set of counters. Each rank has two integer arrays of length 256 in global memory and one in shared memory. Each element of the arrays is a counter that keeps track of incoming and consumed notifications. One array in global memory counts all notifications coming from ranks on the local device, while the second array counts all notifications from remote ranks. The array in shared memory holds counters that are incremented whenever notifications are consumed.

```
1  __device__ void dcuda_wait_notifications(dcuda_state &gst,
2                                              int tag,int count)
3  {
4     __syncthreads();
5     if(threadIdx.x == 0){
6        uint32_t consumed = gst.consumed_notifs[tag] + count;
7        uint32_t lnotifs, rnotifs, total, diff;
8        do {
9           lnotifs = readNoCache(&gst.local_notifs[tag]);
10          rnotifs = readNoCache(&gst.remote_notifs[tag]);
11          total = lnotifs + rnotifs;
12          diff = total - consumed;
13       }while(diff > UINT_MAX / 2)
14
15       gst.consumed_notifs[tag] = consumed;
16    }
17    __syncthreads()
18 }
```

Listing 4.2: dCUDA wait notifications library function

Suppose a rank wants to wait for a notification with a given tag. The value `local_notifs[tag]` gives the number of notifications received from ranks of the local device since the start of the application. The value `remote_notifs[tag]` gives us the number of notifications coming from remote ranks. The sum of both values gives the total number of notifications that this rank received with the given tag since application start. The value `consumed_notifs[tag]`, which is stored in shared memory, gives the number of notifica-

tions we already consumed previously. Therefore, the expression `local_notifs[tag] + remote_notifs[tag] - comsumed_notifs[tag]` returns the number of notifications that are ready to consume. Note that overflowing counters in long runs of dCUDA are properly handled by the device library.

Listing 4.2 shows the procedure to check if enough notifications have arrived to consume `count` notifications. The function waits in a loop until enough notifications are ready and accounts for overflowing counters. To load the notification counters, we use the function `readNoCache`, which uses a volatile pointer to load from memory without using the cache.

The `local_notifs` array is only updated by other ranks of the local device. They use CUDA atomic operations to prevent race conditions and to maintain a consistent memory state. The array `remote_notifs` holds the notifications from remote ranks. This array is updated by the host runtime system using *gdrcopy* whenever it receives a new notification. The array `consumed_notifs` is updated only by the owning rank and therefore resides in shared memory for fast access. We chose to use multiple counter arrays instead of a single one because the GPU architecture of our test setup does not support system-wide atomic operations, which would result in race conditions when the host and local ranks try to write to the same memory location. The most recent GPUs architectures support system-wide atomic operations, which could boost the performance of dCUDA, since the notification system could be implemented with a single array per rank, thus reducing memory accesses.

The notification process has significantly less overhead compared to the older variant. Now two reads of global memory, one read of shared memory and a write to shared memory to consume the notifications are all it needs to process some notifications. Before the optimisation, a notification queue stored all the notifications. The entire queue had to be traversed if a matching notification could not be found. Additionally, notifications matching used the tag, source and window instead of only a tag value. When some notifications were consumed, it required a costly memory copy of the remaining notifications entries to keep the queue organised.

Using the new notification system, notifications to local ranks are delivered directly and do not have to loop back over the host runtime system anymore. This reduces the latency of local notifications to a single memory write and a memory read to global GPU memory and reduces the demand on the CPU and the PCI-e bandwidth.

4.5 Notified Access

The dCUDA function *dcuda_put_notify* performs notified access of local or remote memory and notifies the target rank. The functions *dcuda_put* and *dcuda_notify* only access remote memory or notify a target rank respectively. Most effort was put into optimising this functions as they are the most important operations in dCUDA because they are used for data transfers of any kind.

Notified access to memory of a rank on the local device is now performed by the device only. The data is directly copied from the source buffer to the target buffer in global memory of the local device. In case of overlapping windows in the global memory and if the source and target buffers are equal, no copy is performed. The copy operation is performed by all threads

in the thread block of the source rank in parallel with coalesced access for optimal memory throughput. In a next step, the source rank updates the notification counter of the target rank using the CUDA atomic add function. Before updating the notification counter, a memory fence instruction guarantees that the target rank observes the memory copy operation and the notification update in the same order. Without a fence instruction, CUDAs weak memory model would not guarantee any ordering and a rank might see the notification before the memory copy is complete.

Notified access of remote memory involves the host runtime system, which initiates the data transfer and sends the notification to the target host. The source rank uses the command queue to communicate with the corresponding block manager on the host runtime system. The rank sends two messages to the host for a single remote access request as the queue entries are of limited size and one message can not hold all necessary information. The first message includes the type of remote access, e.g. notified put, put or notify, the window id, the target rank index, the notification tag, and the operation id, which is used to keep track of completed remote access operations. The second message, sent to the host immediately after the first one, holds the pointer to the source buffer holding the actual data, the size of memory transfer, and the offset to the target window base-pointer where the data will be written to. This operation does not require any acknowledgement from the host and the device can continue its work while the host runtime system processes the request.

The main thread of the host runtime system periodically polls all the command queues of all block managers. When the thread receives a notified remote access command from the device, it will issue the actual memory transfer to the target memory. How this is done depends on what network manager is used.

InfiniBands RDMA with immediate

Processing RMA requests using the InfiniBand network manager is better optimised than the MPI version and uses a single RDMA message. The host runtime system of the receiving side is less involved in the memory transfer than the MPI version. Figure 4.3 shows the sequence diagram of a notified put request when the InfiniBand network manager is used. Again, the main thread of the source host receives two messages from the device. RDMA requires the sender to know the exact memory location of the target buffer. Therefore, the sender looks up the base pointer of the target window. All the base pointers of every rank and window are stored on each process in an array in host memory. Knowing the target memory location, the main thread issues a non-blocking RDMA write with immediate operation to write the actual data directly into the remote memory. In addition to the actual data transfer, the operation sends a 32 bit immediate value to the target host. The first 8 bits of the immediate value hold the tag value of the notification, while the rest hold the rank index of the target. The main thread regularly polls the send CQ that holds all completed InfiniBand RDMA send operations. When a RDMA request has completed, the host updates a counter with the operation id in device memory to let the device know that the send operation is completed and the source buffer can be reused.

The RDMA operation transfers the data directly to the target buffer on the destination device. The host runtime system of the receiver does not contribute to the data transfer in any way. The only job of the host on the receive side is to update the notification counters. When the Infini-

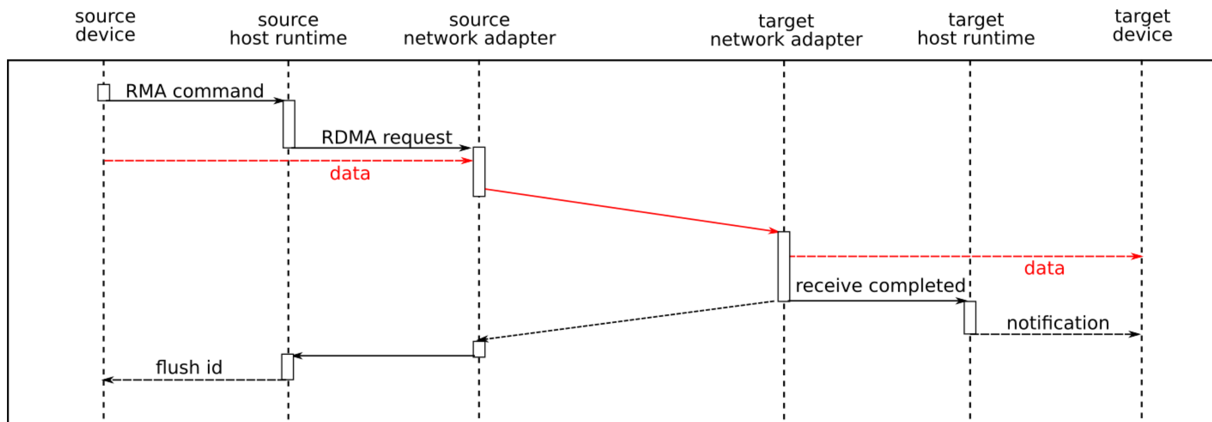


Figure 4.3: Sequence diagram of notified memory put using InfiniBand Verbs

Band HCA of the receiver has completed an incoming RDMA request, it will post a completion entry to the receive CQ located in host memory. The host receive thread will regularly poll said queue and reads the completion entry which includes the immediate value of the RDMA operation. The target rank and tag value of the notifications encoded in the immediate value are enough information to increment the corresponding notification counter in device memory.

Remote put requests without notification use RDMA write without an immediate value. This operation directly writes to the remote device memory but does not generate a completion entry at the receiving host. Therefore, the host runtime system is not involved at all on the receiver side. Notify requests without memory access work the same as put notify requests, except that the payload of the RMDA write with immediate operation is empty and only the immediate value is sent to the target host.

MPI ISend

We recap the process of sending RMA messages using MPI, which was not changed by the optimisations done in this thesis [8]. Figure 4.4 shows a sequence diagram of the whole process. When the main thread on the source host receives the two messages from the device, it first sends a MPI message to the host of the remote rank including the meta information of the actual data transfer. The meta information includes the access type, e.g. notified put, the source rank, the target rank, the window id, the offset, the message size and the notification tag. Right after the first message, the main thread sends a second MPI message to the target host including the actual data. The network driver uses GPUDirect RDMA to directly read the data from device memory without staging it through host memory. Both MPI messages are sent using the non-blocking *MPI_Isend* function, such that the main thread can process other RMA requests while the data is transferred. The main thread periodically tests whether the MPI operations have completed. Then the host updates a counter called flush id in device memory, telling the source rank that the operation has completed and that the source buffer can be used again.

On the target host, the receive thread processes the incoming MPI messages from the source host. The thread makes sure that a non-blocking *MPI_Irecv* operation is always active to receive incoming RMA requests from any source to a temporary receive buffer in host memory. The thread starts a new receive operation whenever the previous operation completes. The messages

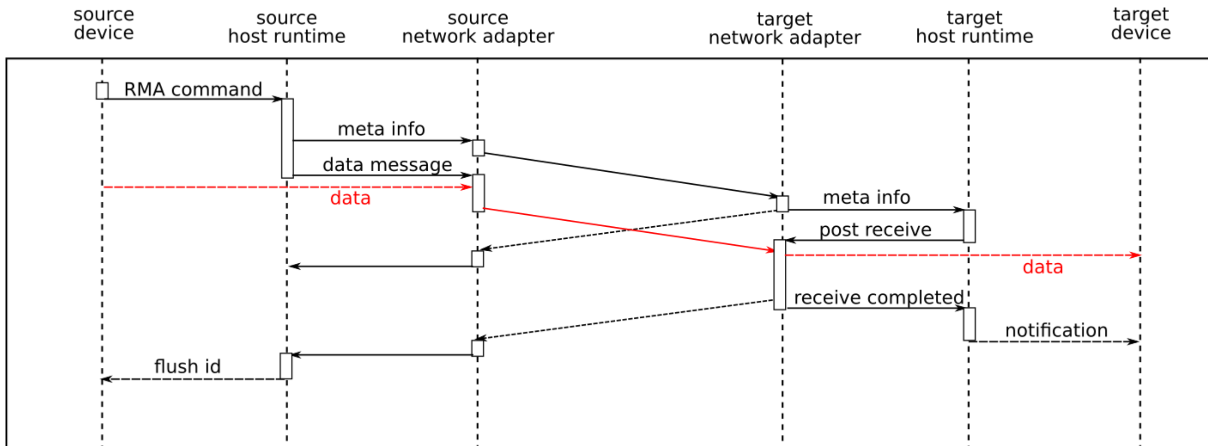


Figure 4.4: Sequence diagram of notified memory put using MPI

that the thread receives include the meta information of the actual data transfer, which is then used to look up the base-pointer of the target window of the target rank. Then, the receive thread starts another non-blocking receive operation for the actual data transfer from the source host. The receive buffer pointer points to the target memory on the device. MPI makes sure that the data is directly copied from the network card to the device memory without traversing host memory. The receive thread regularly tests if the memory transfer has completed. Once it has completed, the thread updates the corresponding notification counter of the target rank using *gdrcopy*.

Put operation without a notification follows the same process except that it does not include the last part where the receive thread updates the notification counter. Notify operations without a remote memory access involve only a single MPI message and the receive thread updates the notification counter without waiting for a data transfers.

4.6 Deployment

We tested the current dCUDA implementation on a compute cluster connected by an InfiniBand network and with Nvidia Tesla K80 GPUs using CUDA 8.0. We used openMPI version 2.0.1 for the MPI based dCUDA. Other CUDA-aware MPI implementations might but are not guaranteed to be compatible with dCUDA. We also tested dCUDA with MVAPICH2 version 2.2, but due to some bugs in the eager messaging protocol, MVAPICH2 deadlocks in combination with dCUDA. The source directory of this project includes a custom patch to make MVAPICH2 compatible to dCUDA. In the remainder of this thesis we use openMPI because it is compatible with dCUDA without source code modifications.

The MPI version of dCUDA relies on MPI only for communication. Therefore, dCUDA should run on any kind of network with a driver supporting GPUDirect RDMA. Prerequisites are a CUDA-aware MPI version and on an InfiniBand system the Nvidia peer memory driver plugin that enables GPUDirect RDMA for InfiniBand. To build a CUDA-aware openMPI from the sources, one can use the configuration argument `--with-cuda=/path/to/cuda`. Additionally, openMPI has to be configured to enable multi-threading. The configuration argument

`--enable-mpi-thread-multiple` enables support for a multi-threaded host runtime system. Optionally, multi-threading can be disabled for the MPI version in the *dcuda_constants.h* file by setting the macro `MPI_MULTITHREADED` to 0. Disabling multi-threading for MPI can also improve the performance of dCUDA as the performance of concurrent MPI calls are not performance optimised in the current implementations of openMPI. The InfiniBand version does not need an installation of MPI but requires the system to be part of an InfiniBand network. The Nvidia peer memory driver module has to be loaded as well. Both versions require the *gdrcopy* driver module installed in the system as well.

GPUs different from the Tesla K80 can run dCUDA as well as long as it is supported by CUDA. Note that some older GPU can not map all available GPU memory to the host and to GPUDirect RDMA because the size of the BAR1 memory is smaller than the device global memory space. The K80 has a BAR1 of 16GB which is larger than the 12GB GPU frame buffer. Therefore, all of the device memory can be mapped on the PCI-e. The older K20c has 5GB frame buffer but only 256MB BAR1. This means that dCUDA can effectively only use a fraction of the device memory.

Building dCUDA

We use *cmake* to manage the building process of dCUDA. The *CMakeLists.txt* file in the main folder configures dCUDA and generates a makefile to compile several test programs and benchmark applications. The configuration of dCUDA can be changed with *ccmake*. Setting the cmake option `CMAKE_BUILD_TYPE` to `Release` enables compiler optimisations. The cmake switch `USE_MPI` is used to switch between the MPI version and the InfiniBand Verbs version of dCUDA. The switch `BMAN_LOGGING` enables additional dCUDA logs for debug purposes.

The file *dcuda_constants.h* defines some dCUDA constants that can be adjusted to tune the application's performance. E.g. queue sizes of InfiniBand queues or internal dCUDA queues.

To compile a custom dCUDA application and link it with the dCUDA library, one can add the following lines to the *CMakeLists.txt*.

```
CUDA_ADD_EXECUTABLE(example_app apps/example/src1.cu
                                apps/example/src2.cu)
TARGET_LINK_LIBRARIES(example_app dcuda)
```

Running dCUDA

When a dCUDA application was successfully compiled, we can run it either using *mpirun* for dCUDA applications using MPI or we can run it using the bash script *dCudaRun.sh* located in the dCUDA main directory if dCUDA was compiled with InfiniBand Verbs.

A dCUDA process should be started for each device we want to run the application on. When using *mpirun*, multiple runtime configuration parameters are available, some of which are important for dCUDA. In the following we list some important parameters of openMPI.

-mca btl_openib_want_cuda_gdr 1 tells openMPI to use GPUDirect to access GPU memory. By default, openMPI will stage the memory through host memory for all messages.

-mca btl_openib_cuda_rdma_limit X tells openMPI to use GPUDirect for all message sizes up to X bytes. Larger messages will be staged through host memory for better throughput. Default is 30'000 bytes.

-mca mpi_common_cuda_cumemcpy_async 1 tells openMPI to use non-blocking async memory copies when staging device memory through the host. Non-async memory copies would cause a deadlock with dCUDA, as they can not be executed while a kernel is running.

Applications that are compiled with the InfiniBand Verbs version of dCUDA can not be executed with the help of mpirun. Each process can be started individually but the following parameters have to be passed to the program:

```
-d, --dev_id       device id of the GPU device
-p, --proc_id      unique process id < n
-n, --n_procs      number of total processes
-r, --root_name    node name of the leader process (e.g. localhost)
-l, --is_root      sets process to be leader
```

The device id tells dCUDA, which GPU device to use. Note that no more than one process on the same node should have the same device id. One process has to be the leader process, which is indicated by adding the parameter `--is_root`. All other nodes have to pass the name of the node where the leader process is to the program. dCUDA will then connect to the leader and set up all InfiniBand connections for the rest of the execution. The processes start the actual application only when all processes have connected to the leader. The leader process has to be started and ready for incoming connections before any other process can be started.

Instead of starting each process individually, the bash script *dCudaRun.sh* can be used, which automatically spawns all processes on all hosts using ssh. The executable has to be present on all hosts at the same location in the file system. Before starting the script, one has to edit the file and change the variable `build=/path/to/build_folder`, such that build points to the folder where the executable is located on each host.

The *dCudaRun.sh* scripts uses the following parameters:

```
1 -f    name of the executable - optional application parameters
2 -n    total number of dCUDA processes spawned
3 -p    add a hostname to the host list
```

`'/dcuda/dCudaRun.sh -f "example_app -x 1" -n 4 -p host0 -p host1'` runs four processes of an application example_app on 2 nodes and passes the custom parameter `-x 1` to the application. The parameter `-p` can be used multiple times to add more hosts to the list. The script only spawns multiple processes per node if the total number of processes is

larger than the number of hosts given to the script. The script has to be started on the first host specified with the parameter `-p`. It saves the output of each process in files called *proc_x.log* in the build folder of the application with *x* being the id of the process.

5

Evaluation

The dCUDA programming model is designed to simplify the process of programming GPU accelerated applications on large compute clusters. Apart from the usability of the framework, performance is critical in this field. An easy-to-use programming model that has a poor performance will not be used by performance aware developers. We show that the implementation of dCUDA as shown in Chapter 4 performs as good as or better than the traditional approaches like MPI-CUDA, depending on the communication patterns and application design. dCUDA suffers from more library overhead but makes up for it with fine-grained synchronisation and overlap of communication and computation yielding higher GPU utilisation than state of the art approaches. We will study the performance of dCUDA by conducting several micro-benchmarks to find the overhead of each component of the dCUDA system. Then, we use the micro-benchmarks to derive the throughput of dCUDA. Finally, we will compare the performance of dCUDA to the state of the art approach using four different case studies. The case studies include a particle simulation, a stencil code, sparse matrix vector multiplication, and power iteration.

5.1 Test Environment and Timing

All tests and benchmarks presented in this chapter were executed and measured on up to five nodes of the *greina* compute cluster in CSCS (Swiss National Sumpercomputing Center) [4]. Three of the nodes have an Intel Xeon E5-2630 v4 CPU (Broadwell), one has an Intel Xeon E5-2680 v3 and one has an Intel Xeon E5-2699 v3 CPU (Both Haswell). All run with Scientific Linux 7.2. Each node has a single Nvidia Tesla K80 connected over PCI-e. A Tesla K80 has two independent GPUs that share the same PCI-e slot. The nodes are connected by an InfiniBand network with a Mellanox MT4115 ConnectX-4 adapter using the OFED 3.4 software stack.

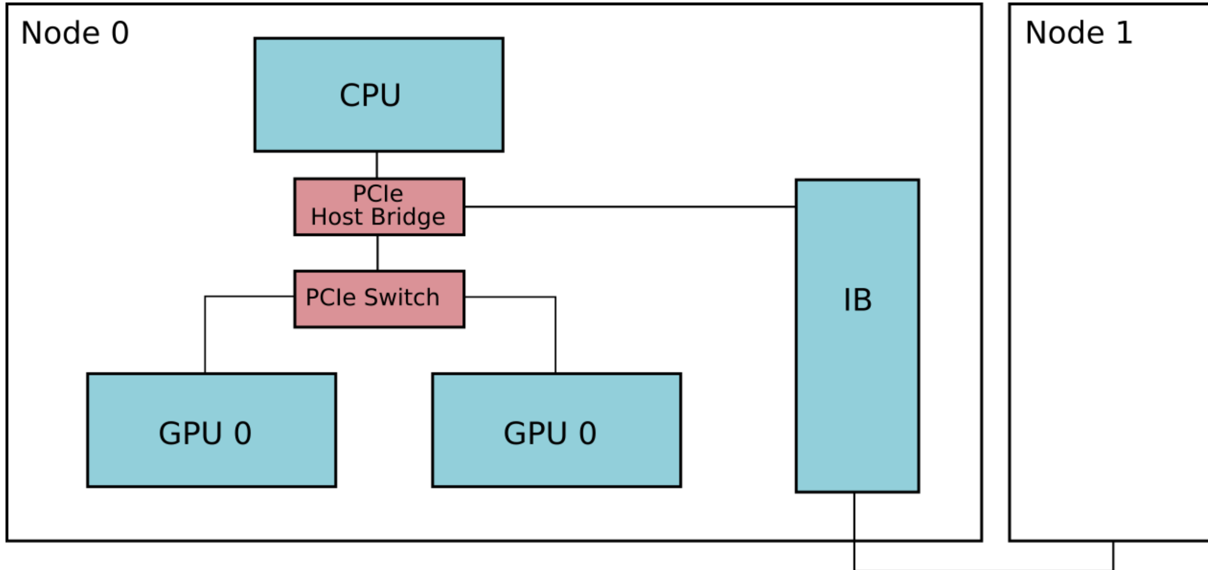


Figure 5.1: Test system configuration

The two GPU devices can communicate with the InfiniBand network adapter through a PCIe host bridge. Figure 5.1 shows the system configuration determined using the Nvidia topology tool with the command *nvidia-smi topo -m*.

```
         GPU0  GPU1  mlx5_0   CPU Affinity
GPU0     X     PIX   PHB      0-9
GPU1     PIX   X     PHB      0-9
mlx5_0   PHB   PHB   X

Legend:

   X   = Self
   PHB = Connection traversing PCIe as well as a PCIe Host Bridge
   PIX = COnnection traversing a single PCIe switch
```

The dCUDA code was built using CUDA 8.0 toolkit and *cmake* version 4.3.1 with GCC version 4.9.2. The Nvidia drivers version 367.48 was installed and the dCUDA variant with MPI used openMPI version 2.0.1.

If not otherwise specified, we use the following method to measure the execution times of the benchmarks in this chapter: The execution time dCUDA applications use the dCUDA timing infrastructure as described in Section 3.3. We start a CPU timer from GPU code at the start of an algorithm but after the initialisation of dCUDA and after the window creation. We stop the timer when the algorithm has finished. Starting and stopping the timer requires a message to the CPU, which has an overhead of less than $2\mu s$. The CPU uses the `std::chrono::high_resolution_clock` to get the current time. Baseline applications that do not use dCUDA, are timed with the same high resolution CPU clock. The algorithm is again timed at the start and at the end of the algorithm code. Additionally, we time each network communication step individually and sum it up to get the communication overhead of the algorithm. We do not measure the communication overhead in the dCUDA applications as

communication phases are not distinct from computation phases.

5.2 System Latency Benchmarks

We measure the latencies of each component of the system individually to understand the capacity of the system and its limitations. The 95% confidence interval of all reported latencies lie within +/- 0.1 μs of the reported mean values.

Notified Access Latency

We start off by measuring the latency of notified accesses. We measure the following scenarios: Notified access to a rank on the local device, notified remote access to a rank on a peer device on the same node, and notified remote access to a remote device on a different node. Additionally, we test notify without memory access to a peer device on the same node and to a remote device on a remote node. We measure the latency of the operations by sending ping-pong messages between two ranks. We start the timer and then start to send 500'000 ping-pong messages. After the last iteration we stop the timer. We get the mean round trip time of the operation by dividing the measured time by the number of iterations. The latency of a single notified access is half the round trip time, which we report in Table 5.1. The notified access tests, writes 4 bytes of data to remote memory.

	InfiniBand	**MPI**	**Original**
Not. Access Local Device	2.4	2.4	6.4
Not. Access Local Node	6.7	23.7	25.4
Not. Access Remote Node	6.9	12.2	14.7
Notify Local Device	1.9	1.9	6.4
Notify Local Node	3.4	5.0	n/a
Notify Remote Node	3.6	5.4	n/a

Table 5.1: Notified access and notify latency in μs.

Table 5.1 shows the test results. We compare the results from dCUDA with the InfiniBand network manager, dCUDA with MPI, and dCUDA without the optimisations of this thesis as described in the original paper [8]. This thesis introduces the notify operation, which is why the original implementation does not support it. We simulated notify to a local rank by setting the message size to 0 bytes.

The latency of communication with local ranks is reduced by a factor of 2.6 and even more for notify. The reason is that the new dCUDA implementation does not need host interaction anymore and notification matching is simplified. All dCUDA versions use loopback on the network adapter for communication with a peer device on the local node. The openMPI implementation can not handle loopback well when device memory is involved, leading to degraded performance, which is worse than communication to a remote node. We do not observe this

behaviour with the notify operation as no data is transferred from GPU memory. The InfiniBand version of dCUDA does not suffer from the same effects as MPI and therefore achieves a speedup of over 3.5 compared to both MPI version. Memory access on a remote device is 1.75 times faster when using InfiniBand over MPI or even 2.1 times faster compared to the unoptimised version. Notification without device memory access can be achieved 1.5 times faster with InfiniBand.

We analyse device local communication in more detail, as the only component involved is the GPU in the optimised implementation. Notified access involves two memory reads on global memory to lookup window buffer and the notification array. Then a memory copy in volatile global memory is followed by a memory fence and an atomic add of the notification counter. After this, the receiver rank has to wait until the notification is visible in memory by reading the notification counter. All these memory operations sum up to an overhead of 2.4 μs. Removing the memory copy reduces the overhead to 1.9 μs. The unoptimised original implementation required GPU-host communication for handling notifications, leading to much longer latencies.

Host-Device Latency

dCUDA operations that are not local to a single device require communication with the host runtime system. To measure the delay of a message sent from host to device and back, we used different benchmarks on our system. The first benchmark is a ping-pong test using a single flag in memory. The host sets it, while the device waits and resets it as soon as it notices the change. We measured the round trip time of 100'000 ping-pong iterations. Table 5.2 show the round trip times of this test. *Host mem access* used a single flag in host memory, while the flag is located in device memory in the *device mem access* test. The device uses CUDA's host memory mapping to access host memory and the host uses *gdrcopy* to access device memory. The memory access time of device memory is lower because the device requires a lot of time to access host memory, while the host can access device memory relatively fast with *gdrcopy*.

Host mem access	2.2
Device mem access	1.8
Queue H-H	2.4
Queue D-D	2.0
Queue H-D	2.4
Queue D-H	1.9

Table 5.2: Host-Device ping-pong on flag in host memory, flag in device memory and using dCUDAs flagged queues. Results are round trip times in μs.

The next tests shown in Table 5.2 show the ping-pong round trip time using special queues as used in dCUDA[8]. We use one queue for each direction, which we call ping-queue and pong-queue. The host writes to the ping-queue while the device polls it. The other direction uses the pong-queue. We tested four different configurations. In each test the queues are in different memory locations, e.g. the configuration H-D says that the ping-queue is in host memory and the pong-queue in device memory. The results from the configurations H-H and D-D show

that the queues have an overhead of about 0.2 μs in addition to a the memory access when we compare the result with single flags.

The configuration H-D performs equally to H-H. This tells us that the location of the pong-queue is not as important as the location of the ping-queue because the host can access both host and device memory relatively fast compared to the access delays of the device. The performance of the configuration D-H is the best as both, host and device, poll on local memory and the memory write operations are the only memory access that pass the PCI-e bus. dCUDA uses the configuration D-H for the any host-device communication. The communication delays are most likely not the same for both directions but as communication is usually symmetric, we approximate the message delay for each direction to be $0.95\mu s$.

5.2.1 Network Communication Latency

The dCUDA version using MPI sends MPI messages to perform RMA access on remote nodes and on peer devices on the same node. We measured the MPI overhead in dCUDA for sending RMA requests using a micro benchmark. dCUDA uses two MPI messages to perform one RMA operation. First, is sends the meta data from host memory to the host memory of the remote process. Immediately after that, the actual data from device memory is sent to the remote device memory of the receiver using a second MPI message. We measured the ping-pong delay of this communication pattern by first sending two MPI messages to a remote process, which send the data back after the reception using again two MPI messages.

The first column of Table 5.3 shows the results of this benchmark. The first MPI message always uses host memory to send the meta-data and the second MPI message sends the actual data. First, we used a host buffer for the actual data and in a second test we used device memory, which is accessed using GPUDirect RDMA like in dCUDA. We tested communication to a peer device on the same node and communication to a remote node. The actual data buffer has a size of 4 bytes. The latencies reported in Table 5.3 are half the round trip times of a single ping-pong.

The test shows that RMA of host memory on the same node or a remote node has a latency of $1.3\mu s$ and $1.8\mu s$ respectively. RMA to device memory on a remote node is three times slower because of the additional access of device memory at the sender and the receiver. This is a reasonable result, as the round trip time of device memory access is around $1.8\mu s$ as seen in Section 5.2. The device memory latency has to be added two times for both accesses to device memory at the sender and at the receiver. RMA access of peer device memory on the same node results in more latency than we would expect. The reason is that the openMPI implementation does not make use of the GPUDirect RDMA technology for messages to a device on the same node, even though it is enabled. MPI copies the data to the host and sends it to the peer process, which then copies it on its device memory. This requires much more time than GPUDirect would need.

The dCUDA version using InfiniBand does not use MPI but uses the low level InfiniBand Verbs library instead. We performed the same benchmark using only the InfiniBand network manager. We send ping-pong messages using a single InfiniBand RDMA write with immediate operation from one process to another to simulate the dCUDA behaviour. The data buffer has again a size of 4 bytes and is located in either host or device memory. Again, GPUDirect RDMA is used by

	MPI	InfiniBand
Peer put host mem	1.3	1.2
Peer put device mem	53.2	2.0
Remote put host mem	1.8	1.4
Remote put device mem	5.4	2.2

Table 5.3: RMA ping-pong mean latencies in μs

the InfiniBand driver to read and write to device memory.

The second column in Table 5.3 shows the result of this benchmark. RMA accesses of host memory are slightly faster than MPI because InfiniBand has less library overhead and only a single message is sent instead of two as in the MPI implementation. Remote device memory access is 2.5 times faster using InfiniBand compared to MPI. An additional overhead of $0.8\mu s$ is required to read and write device memory. We suspect that the InfiniBand driver can better pipeline device memory accesses and network transaction than MPI or that MPI does an additional memory copy at some place. Peer device memory access is as fast as we would expect from the other measurements. It does not suffer from the same performance degrading effects as MPI does.

5.3 Bandwidth

We measured the bandwidth of the system with notified access. One rank sends a notified RMA message to a second rank, which waits for the notification and answers with another notification without any data. When the sender receives the response from the receiver, the next iteration starts. We use the Hockney model to derive the bandwidth of the system. The model requires latency L, execution time t, and message size s to derive the bandwidth B. The latency was measured by using a message size of 4 bytes. The message size s of the bandwidth test is 1MB.

$$B = \frac{s}{t - L}$$

Table 5.4 shows the result of the bandwidth test. The 95% confidence intervals of the reported average bandwidth lies within +/- 0.1 GB/s. The first column shows the results when GPUDirect RDMA is used. The dCUDA version that uses InfiniBand Verbs uses GPUDirect for any kind of message, while MPI uses GPUDirect only for small messages. MPI does not use GPUDirect for large messages but rather stages the memory through host memory, which yields better bandwidth.

Device local communication does not involve the host and any network facility. Thus, the performance does not depend on whether GPUDirect is enabled or not. The bandwidth of device local memory copies for communication to device local ranks depends on kernel configuration parameters. All the blocks of a kernel share the bandwidth of the memory bus and a single rank can only use a part of the whole bandwidth. Kernels with few thread blocks achieve better memory bandwidth per rank than kernels with a lot of thread blocks. To achieve the best total

	GPUDirect	Host staging
device local (1/52 ranks)	6.9	
device local (26/52 ranks)	79.3	
peer device (1 rank)	2.7	12.6
peer device (all ranks)	2.2	10.3
remote device (1 rank)	2.7	6.0
remote device (all ranks)	2.2	7.2

Table 5.4: Bandwidth of one-directional notified accesses in GB/s.

memory bandwidth, all ranks have to send data simultaneously. A single rank in a kernel with 52 ranks achieves a bandwidth of 6.9 GB/s. When more ranks send data simultaneously to other ranks, the total bandwidth grows rapidly while bandwidth per rank drops slightly.

Communication to a peer device or to remote devices has a rather low bandwidth due to GPUDirect RDMA. Maximum observed bandwidth is 2.7 GB/s and if multiple memory transfers happen simultaneously, the total bandwidth drops to 2.2 GB/s. The dCUDA version using InfiniBand Verbs uses only GPUDirect, which makes it not a good choice for applications that send large data packets.

The dCUDA version with MPI uses GPUDirect only for small messages and stages large messages through host memory. This protocol has a larger latency but achieves much better bandwidth. Communication to a peer device on the same node is limited by the PCI-e bandwidth to 12.6 GB/s. We achieve 80% of the theoretical peak bandwidth of the PCI-e x16 switch. Communication to remote nodes is limited by the bandwidth of the InfiniBand network. We achieve 60% of the theoretical peak bandwith of 12 GB/s when all ranks of one device simultaneously send messages to a remote device. Our bandwidth test is one-directional and we would expect slightly better network utilisation in a bidirectional test.

The results of this test motivate to extend the dCUDA InfiniBand network manager in future work with an alternative protocol for large messages that stages the memory through host memory to mitigate the low bandwidth of GPUDirect RDMA.

5.4 Case Study: Particles

The first case study is a particle simulation in two dimensional space as first described and implemented in the original dCUDA paper [8]. Particles interact only with nearby particles within a cutoff distance. In each iteration step we compute the forces between all particles within the cutoff distance. In a next step, we adjust the velocity of each particle and compute the new position. We decompose the rectangular domain in the y-direction into small segments. We have to consider only particles in the same segment or in neighbouring segments to compute the forces on a specific particle because the width of a segment is equal the cutoff distance. All particle are stored in a single array and are sorted according to the segment they belong to. The segments have a variable size as every segment holds a different number of particles. Whenever a particle moves to another segment, we copy the particle's information to the new segment to

maintain the order.

We implemented multiple versions of this algorithm. One uses the traditional two model approach using CUDA and MPI individually for host-device interaction and for networking respectively. A second implementation uses the InfiniBand network manager as described in Section 4.2 instead of MPI but uses the traditional two model approach as well. Finally, we implemented the algorithm using dCUDA and we tried both networking variants, InfiniBand network manager as well as MPI.

In the traditional MPI-CUDA and IB-CUDA approaches, we divided the particles array and distribute it over multiple communication ranks, such that each node handles the same number of segments. A single node simulates only the particles in its segments. In addition to the segments a node owns, it stores two additional segments of variable length at the edges that are duplicates of the boundary segments of the neighbouring nodes. These duplications are necessary to compute the forces on particles in segments at the boundary of the local domain. After each iteration step, these duplicates are updated by the neighbouring nodes. Figure 5.2 illustrates inter-node communication of the algorithm. Each column represents the particles in a single segment. After each iteration, a node sends the state of the segment at the boundary to its neighbour and receives the same information from its neighbour in return. The forces and movements of all the local particles are computed by running several CUDA kernels. The local segments are distributed among the different thread blocks to compute the forces and movements in parallel.

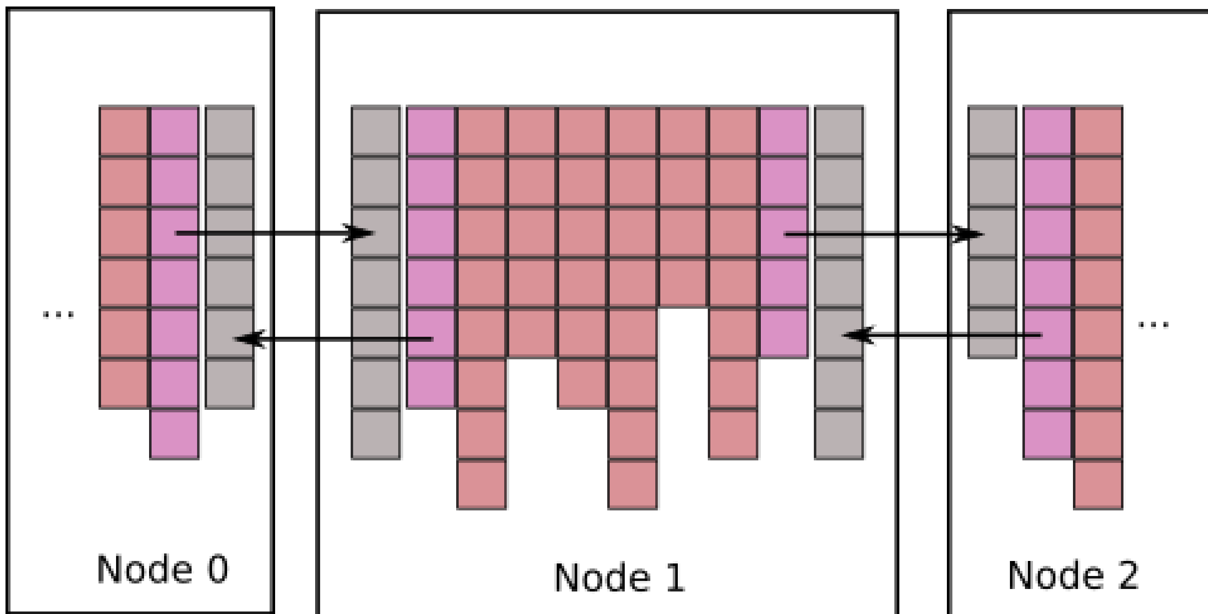


Figure 5.2: Particle algorithm communication pattern

The dCUDA variant of the algorithm runs a single dCUDA kernel and uses dCUDAs notified access to communicate with other ranks. In the dCUDA variant we again decompose the domain into smaller segments of variable size and distribute them among the dCUDA ranks. In the program logic, each rank exchanges its boundary segment with its neighbouring ranks similar as shows in Figure 5.2, except on the level of dCUDA ranks instead of nodes. The memory windows of ranks on the same device physically overlap such that no actual memory trans-

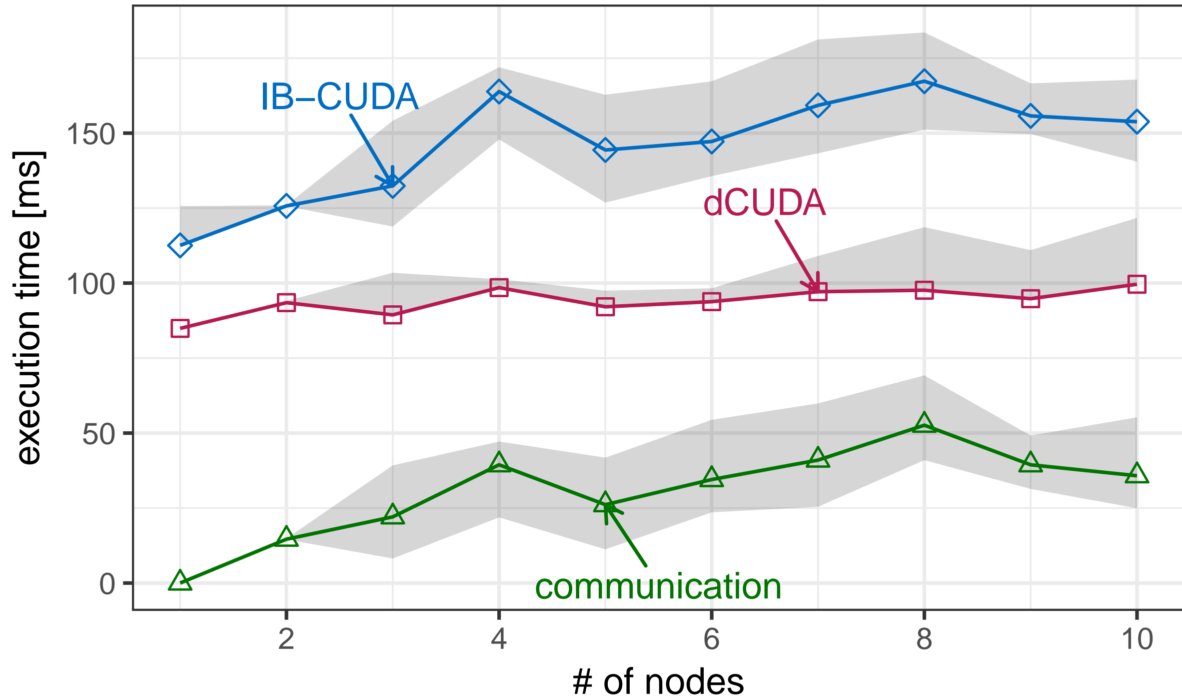


Figure 5.3: Particles weak scaling with InfiniBand network manager.

fer is necessary. Only the ranks at the boundary send data to the neighbouring devices. The dCUDA version has the advantage that communication and computation can overlap because synchronisation is more fine-grained than with the two model approach.

To compare the performance of the different algorithm variations, we performed a weak scaling benchmark. We ran the same problem on 1 to 10 devices on 5 different nodes and scaled the problem size with np, the number of processes. We set the domain size to $416 \times 416 \cdot np$ with a cutoff distance of 1. We initialised $41'600 \cdot np$ particles and ran the simulation for 100 iterations. We collected the runtime of 20 runs and report median runtime as well as the 95% confidence intervals. We measured only the actual runtime of the algorithm excluding the initialisation phase. In the two model variants, we measured the communication overhead separately for each network operation and summed it up to get the total communication overhead for both InfiniBand and for MPI.

First, we analyse the results of the applications using the InfiniBand Verbs network manager. Figrue 5.3 shows the runtime of the two model approach InfiniBand-CUDA and dCUDA using the InfiniBand network manager. Additionally, we report the communication overhead of the InfiniBand network manager.

The algorithm has no communication overhead when only one process is started. Still, dCUDA achieves a speedup of 1.3 on a single node. This shows that we profit from fine-grained synchronisation mechanisms. While IB-CUDA synchronises all blocks on its device after each step, dCUDA synchronises only neighbouring ranks. The communication overhead of InfiniBand grows with the number of processes. The communication intensive benchmark has a communication overhead of almost up to a third of the total execution time with the IB-CUDA algorithm. The dCUDA algorithm sends the same amount of data over the network but the

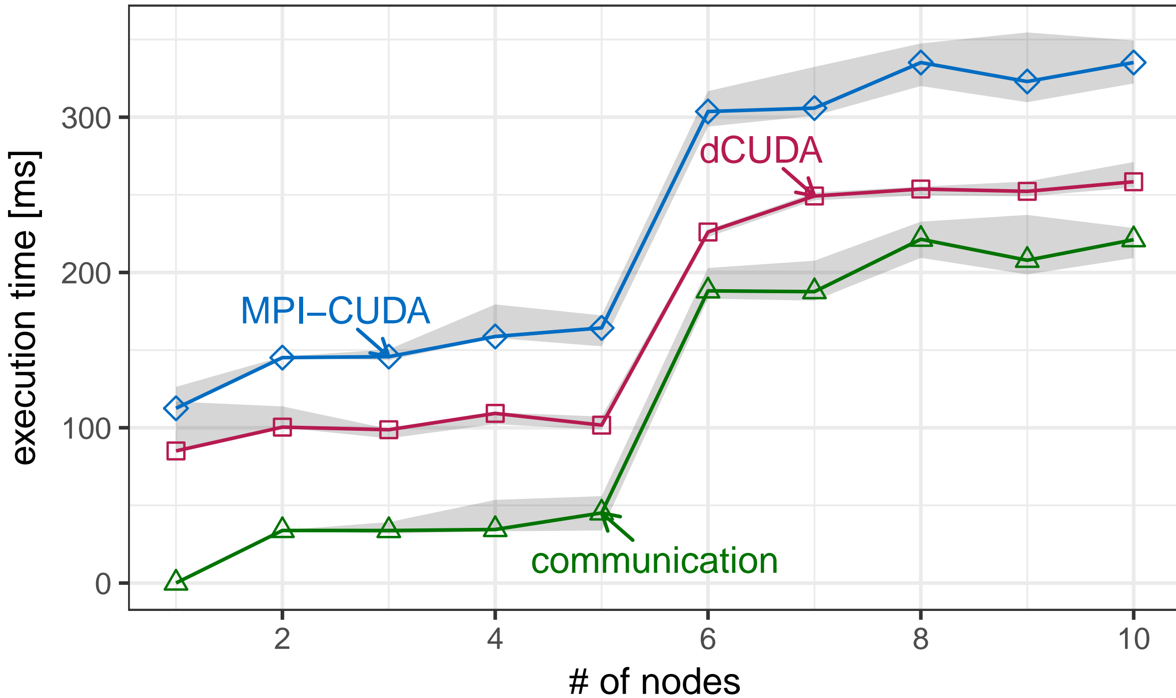


Figure 5.4: Particles weak scaling with MPI.

execution time increases only slightly. This is because in dCUDA communication and computation can overlap. We achieved a speedup of up to 1.7 using dCUDA compared to the two model approach with InfiniBand Verbs.

Figure 5.4 shows the same results when MPI is used instead of the InfiniBand Verbs network manager. Again, we compare the traditional two model approach MPI-CUDA with dCUDA, which uses MPI this time. We see the same result for a single process without communication as before. When we scale the problem to more processes, the communication overhead increases and the execution time of MPI-CUDA increases by the same amount. The execution time of dCUDA increases only slightly thanks to computation and communication overlap. With 6 and more processes we experience a performance drop caused by increased MPI overhead. As stated above, we used five nodes for this benchmark. When we start more than five processes, some of them run on the same node. This is no problem for the InfiniBand Verbs network manager, but openMPI can not properly manage multiple processes on the same node. The communication overhead of MPI is almost four times higher as soon as we make use of 6 or more processes. In consequence, the execution time of both MPI-CUDA and dCUDA increases. While the execution time of MPI-CUDA is the sum of the communication overhead and the computation time, dCUDA can mitigate a part of the communication overhead thanks to overlapping computation.

Figure 5.5 compares dCUDA with MPI and dCUDA with the InfiniBand Verbs network manager. The InfiniBand Verbs version is slightly superior thanks to better communication latencies but the difference is only little because dCUDA can hide most of the latency overhead. MPI can not make use of the superior throughput, because dCUDA mitigates low bandwidth by overlapping the communication with computation in this particular example. As soon as we spawn multiple processes on a single node, MPI can not compete with InfiniBand Verbs anymore.

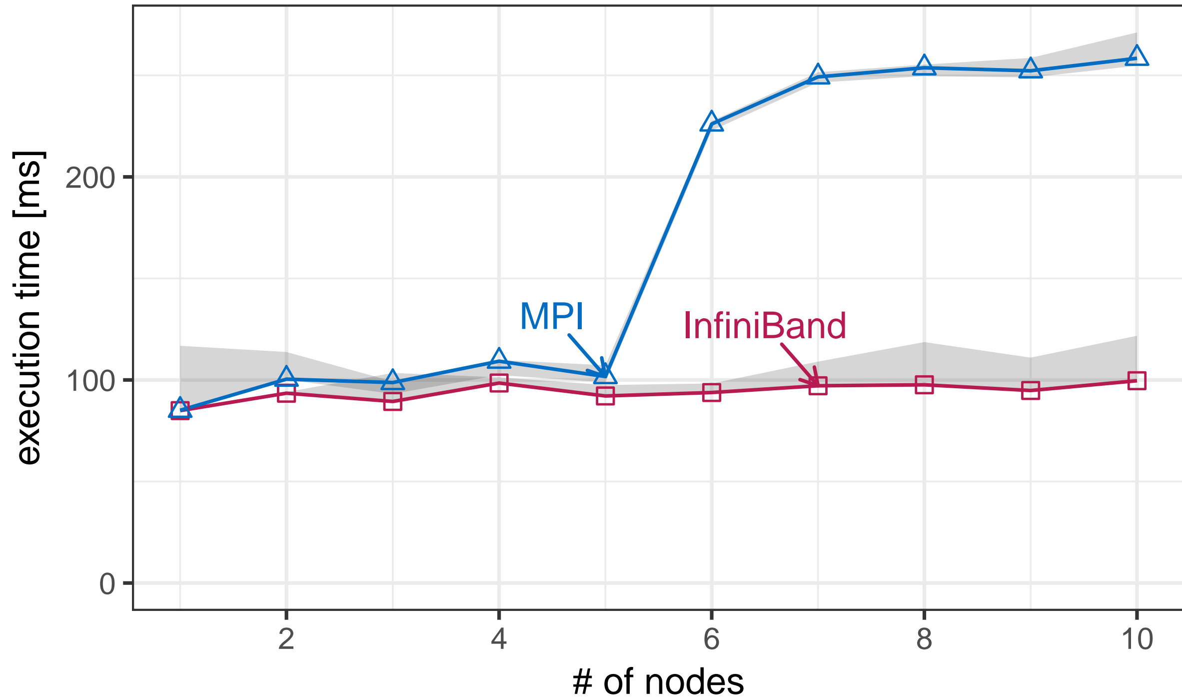


Figure 5.5: Particles weak scaling with two versions of dCUDA.

5.5 Case Study: Stencil Program

The second application we tested on our system was a stencil program, which iteratively executes a simplified horizontal diffusion kernel. A more detailed description of the algorithm can be found in the original paper of dCUDA [8]. We have a three-dimensional grid and apply three dependent stencils to the grid. The stencils consume between two and four neighbouring points in the horizontal xy-plane. We perform a one-dimensional domain composition in the y-dimension and distribute the sub-domains on the different nodes.

We implemented three variants of the same stencil program: First, the two-model approaches MPI-CUDA and IB-CUDA. The former uses MPI for communication and the latter uses the InfiniBand Verbs network manager for communication. Finally, we implemented a dCUDA variant, which we tested using MPI and using the InfiniBand Verbs network manager. The two-model approaches computed the stencils with CUDA kernels and after each kernel execution the two-dimensional border planes are exchanged with the neighbouring nodes. The difference to dCUDA is that dCUDA ranks compute slices in the xy-plane individually and exchange the information with their neighbouring nodes individually. Instead of sending the border plane at once, dCUDA sends it in smaller pieces, while other ranks continue with the computation.

We set the domain size of the problem to $128 \times 320 \times 26$ grid points per device, giving us border planes of size 128×26. The two model approach sends the border grid points with one message to the neighbouring node, while dCUDA sends 26 smaller messages. We measured 20 samples of 200 iterations for every application variant. We scaled the problem weakly to up to eight processes on four different nodes. Test-runs with more than four processes spawned up to two processes per node, each managing a single GPU device. We report the median of the 20

samples and show the 95% confidence intervals.

Figure 5.6 shows the result of the traditional two model approach with the InfiniBand Verbs network manager for the communication and compares it to dCUDA, which uses the InfiniBand Verbs network manager as well. The figure also shows the communication time of the InfiniBand network manager. The computation time without any communication is almost the same for both programs when we start only one process. dCUDA profits from its fine-grained synchronisation mechanisms, but because the work is evenly distributed, IB-CUDA can utilise the GPU resources almost as good as dCUDA. The application sends a lot of small messages and the communication overhead has a radical impact on both variants. dCUDA can mitigate some of the communication overhead with its ability to overlap communication and computation. We achieve a speedup of up to 1.2 using dCUDA over the more traditional IB-CUDA approach.

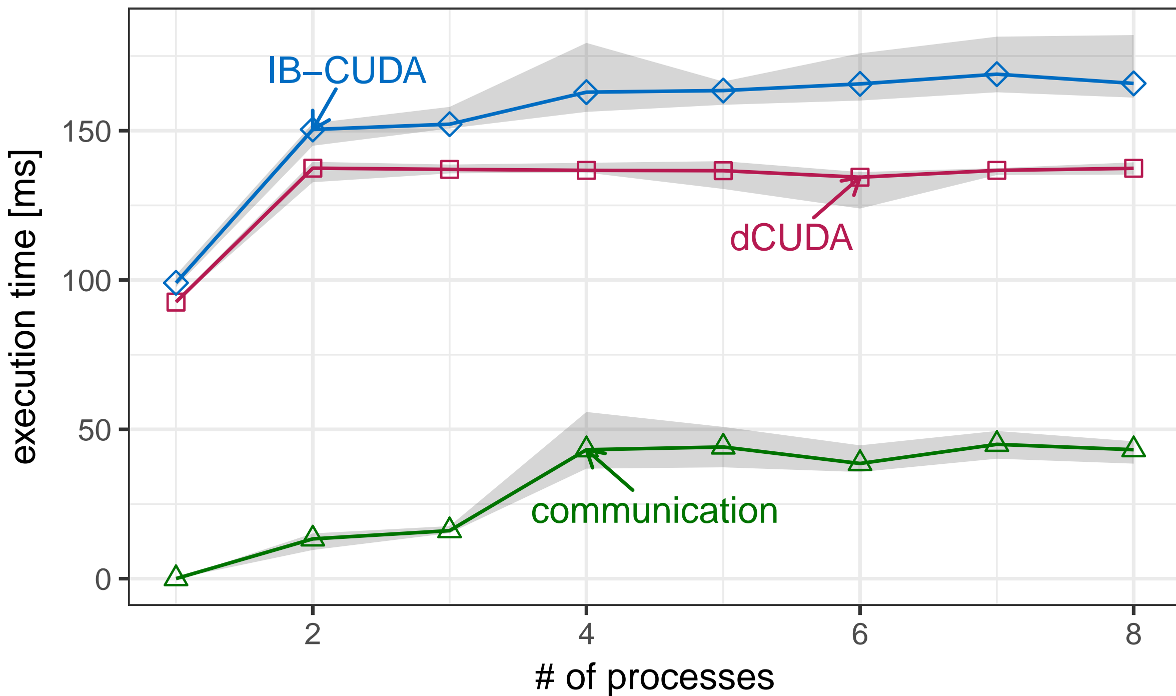


Figure 5.6: stencils weak scaling with InfiniBand network manager.

Figure 5.7 shows the results of weak scaling using MPI instead of the InfiniBand Verbs network manager. We show the execution time of the traditional MPI-CUDA approach and the communication overhead of MPI for the algorithm. The communication overhead increases more than expected when we start multiple processes on the same node, which is the case for all tests with more than four processes. The communication overhead is two times larger because MPI has some problems managing messages from two processes from the same node. The execution time of the MPI-CUDA algorithm is the sum of the computation time and the communication time because communication and computation can not overlap in the two model approach.

We expected the execution time of dCUDA to be somewhere in the range in-between the MPI communication overhead and the execution time of MPI-CUDA, but dCUDA performed worse than we expected. It is up to two times slower than the traditional MPI-CUDA approach. The communication pattern of this algorithms seems to cause degraded communication perfor-

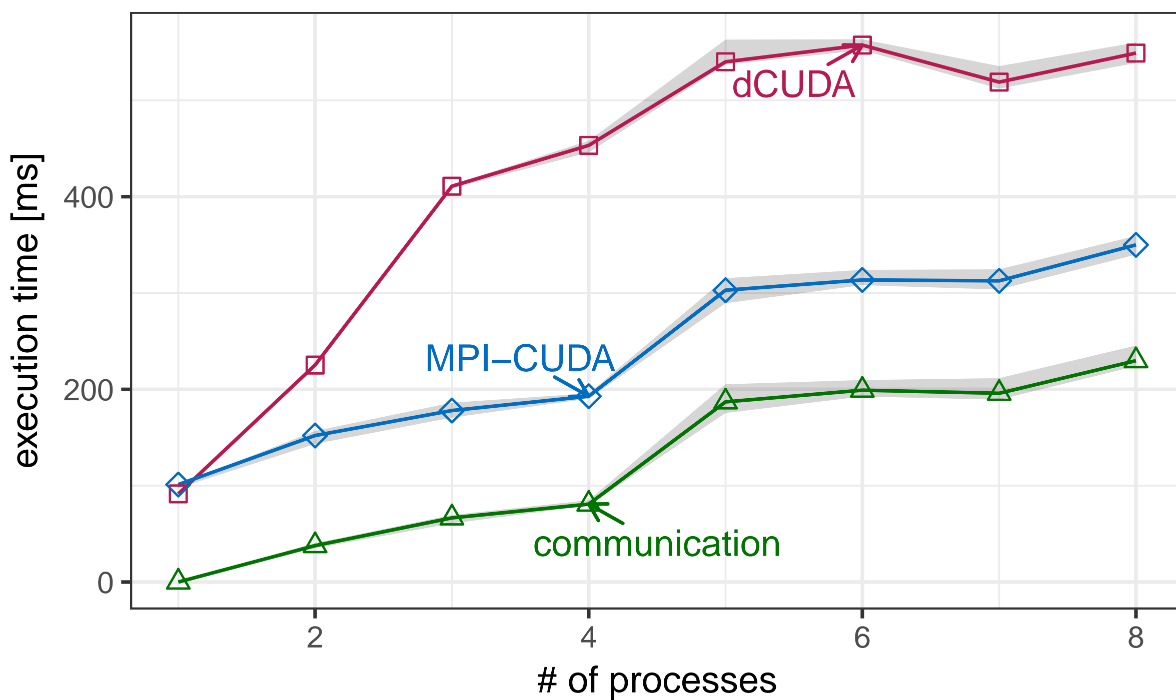


Figure 5.7: Stencils weak scaling with MPI.

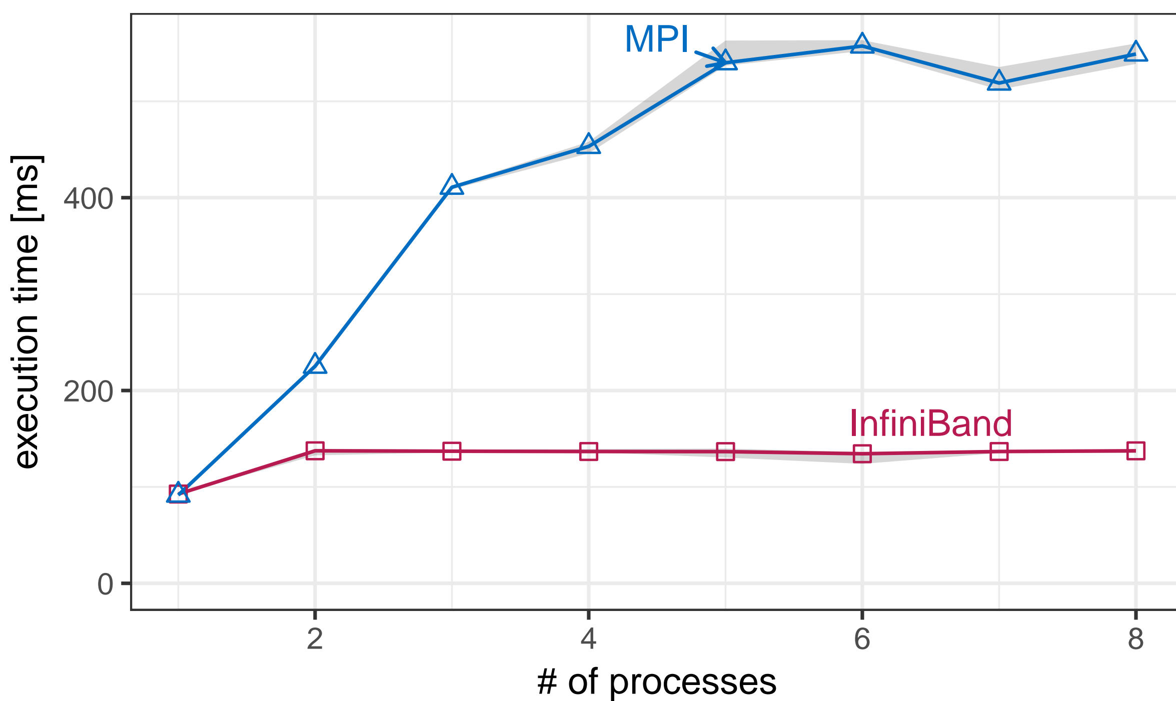


Figure 5.8: Stencils weak scaling with two versions of dCUDA.

mance. We could not reproduce the results from the original paper with our optimized dCUDA version using MPI, nor with the unoptimized version from the original paper [8]. The original paper showed that unoptimized dCUDA ourperforms MPI-CUDA. We assume that the configuration of our new test system causes openMPI to perform poorly when many small messages are sent simultaneously. We could not determine the exact cause of this behaviour. We did not observe this effect in the particles simulation application because data is sent with a single message to its neighbours. The stencil dCUDA program is designed to send 26 small messages to each neighbouring node in each iteration, which seems to reach a turning point, where MPI starts to lose performance.

Figure 5.8 compares the execution times of dCUDA with InfiniBand Verbs and dCUDA with MPI. Because dCUDA with MPI performed poorly, InfiniBand outperforms it by a factor of up to almost four times.

5.6 Case Study: Sparse Matrix Vector Multiplication

In the third case study we compute a matrix vector multiplication of a square sparse matrix with a dense vector. We implemented three variants of the algorithm again. Two algorithms using the two-model approach, one using the InfiniBand Verbs network manager and one using MPI. The last algorithm uses dCUDA and we tested it with the InfiniBand Verbs network manager and with MPI. Again, we perform a weak scaling benchmark with increasing matrix size and device numbers [8]. We store the sparse matrix with the compressed row storage (CSR) format and perform a two dimensional domain decomposition.

Figrue 5.9 shows an example of a $16\ times 16$ domain decomposition and shows all the resources that each device stores. The devices themselves are arranged in a 2×2 matrix and each device has his own 8 by 8 square block of the matrix A in CSR format. We decompose and distribute the blocks in the initialisation phase before the start of the algorithm. The dense input vector b is decomposed as well and distributed to the devices on the first row, e.g. device 0 and device 2. The first device row will send their sub vector b down the columns during the algorithm. When the algorithm starts, each device has its matrix block and each device in the first row has a part of the vector b.

In the first phase of the algorithm, we send the vector b to all devices. In the beginning, only the first row has the part of b that it needs for the multiplication. We distribute the vector down the column using a binomial tree. In our example in Figure 5.9, device 0 sends the vector to device 1 and device 2 sends it to device 3. After this phase, each device has the input data and is ready to perform the multiplication.

In the next phase, each device computes locally $x' = Ab$ and gets the intermediate result vector x'. The computation is performed on the GPU device and each element of x' is computed by a single thread block to prevent write conflicts.

The vectors x' have to be summed up to get the final result of the multiplication. In a last step we perform a reduction on each device row. We use a binomial reduction tree. In each reduction step, the vector x' is sent to next device, which calculates the sum of both vectors. In our example in Figure 5.9, device 2 sends its intermediate result to device 0 and device 3 sends

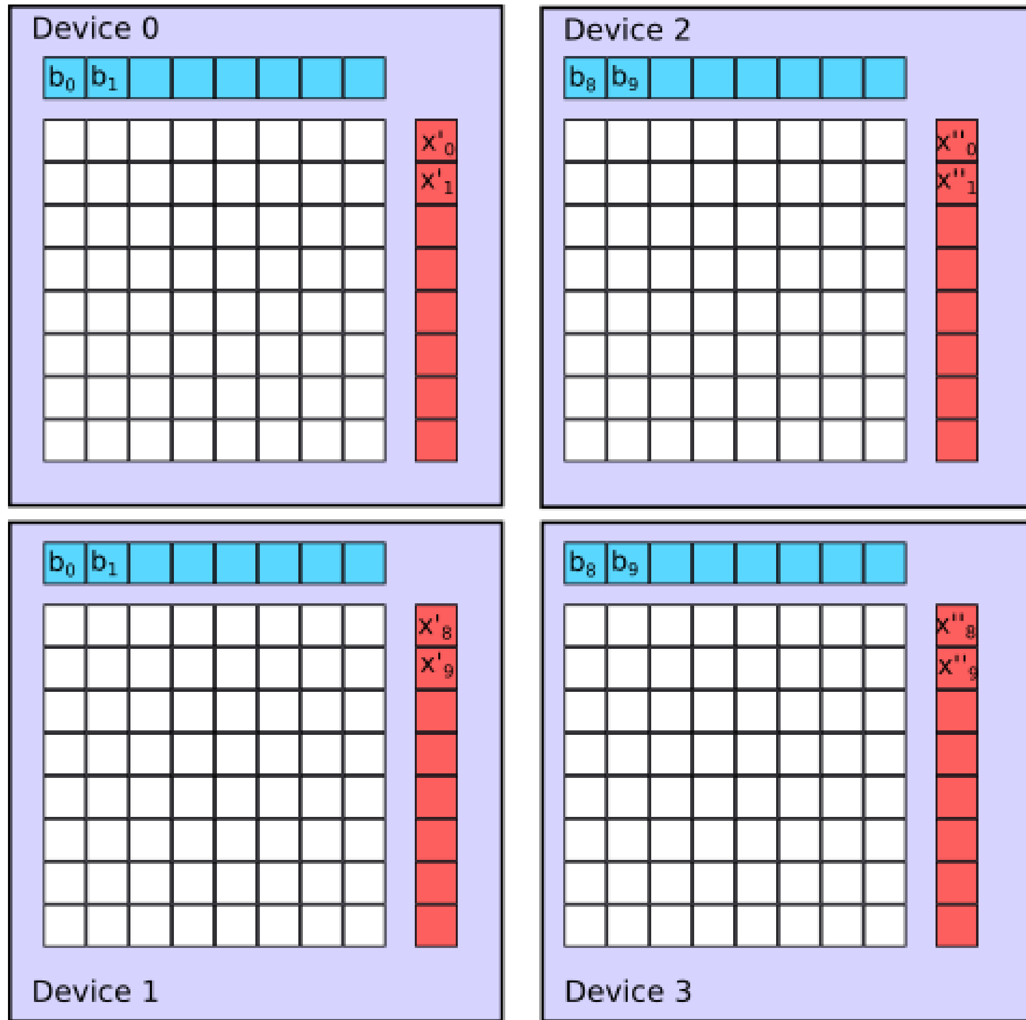


Figure 5.9: Sparse Matrix Vector Multiplication domain decomposition.

it to device 1. In the end, the resulting vector x is located on the devices in the first column, e.g. devices 0 and 1.

In our benchmark we perform the same matrix vector multiplication one after another separated by a global barrier synchronisation to emulate a follow-up operation that synchronises all devices.

The difference of the traditional two-model approach and dCUDA is the following: While the traditional algorithm reduces the resulting vector x' on a per device level, dCUDA can reduce it on a per rank level. dCUDA ranks can start reducing part of the vector x' while other ranks are still performing the multiplication thanks to the fine-grained synchronisation. We expect a small speedup of dCUDA over the traditional two-model approach thanks to overlap of communication and computation. Because of the global barrier at the end of each multiplication, we can not use the full potential of dCUDA as multiple iterations can not overlap in time.

We performed weak scaling with 1, 4 and 9 devices on 5 different nodes. Each device owns a sparse matrix block of size 10816×10816 with 0.3% non-zero matrix elements. We took 20 samples of the execution time of 100 iterations of the algorithm excluding initialisation time.

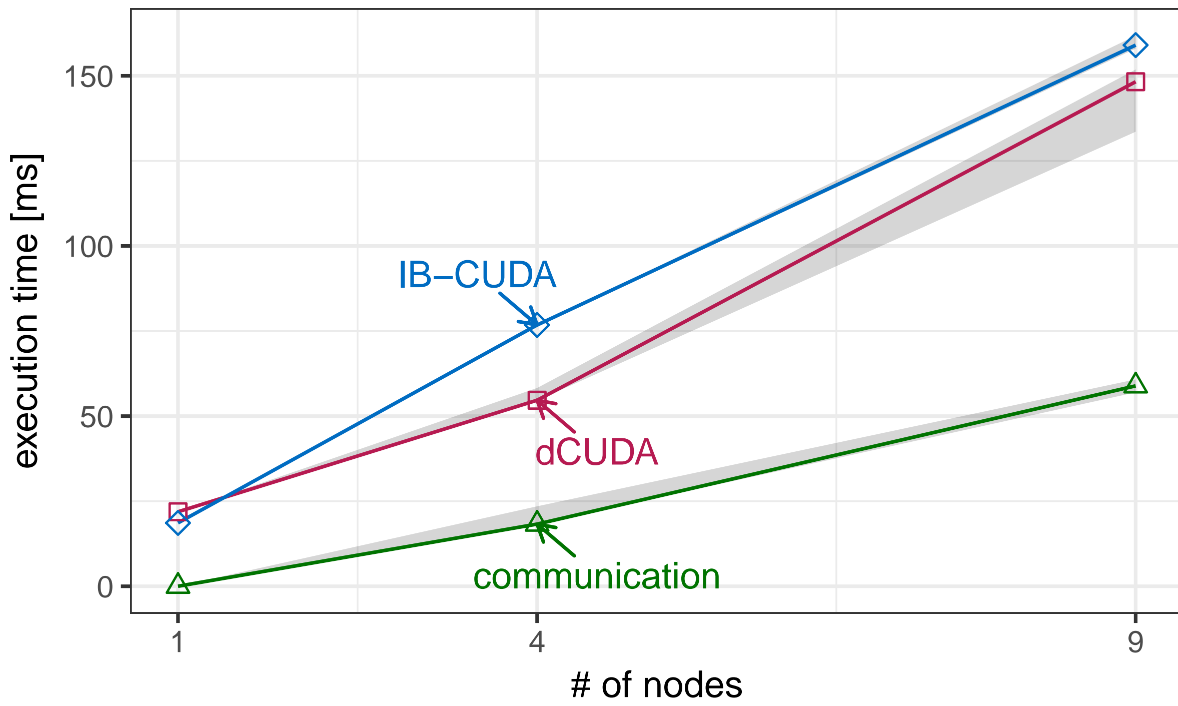


Figure 5.10: Matrix vector multiplication weak scaling with InfiniBand network manager.

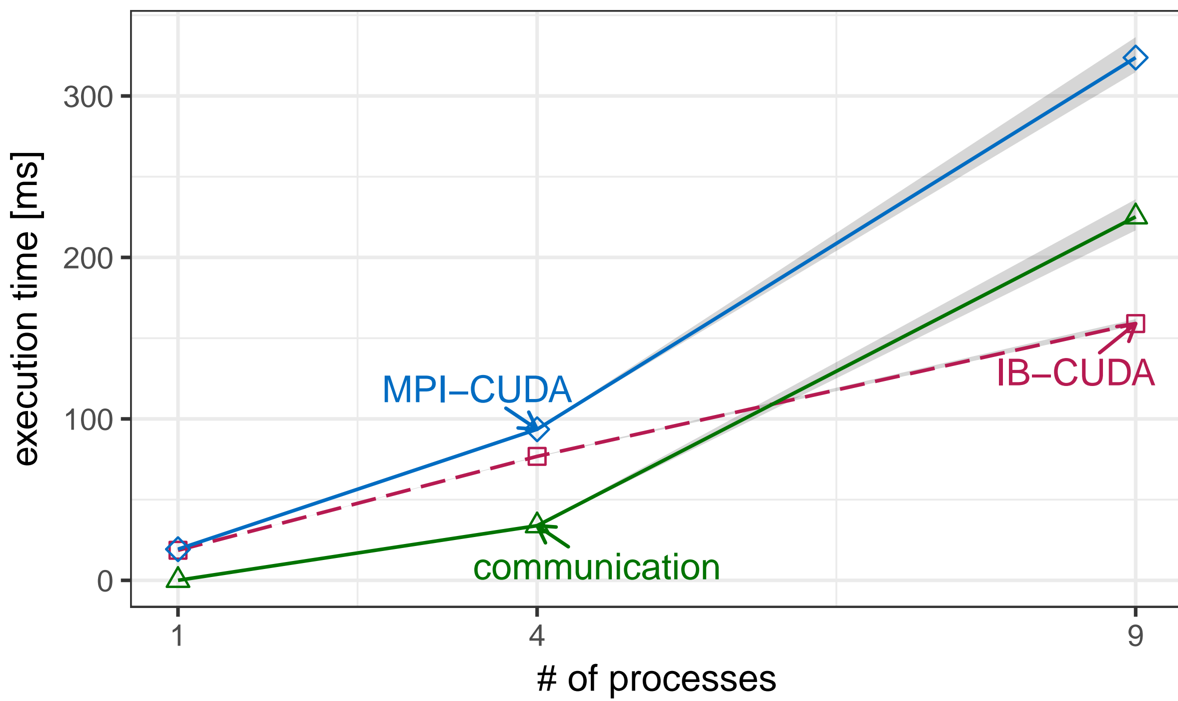


Figure 5.11: Matrix vector multiplication weak scaling with MPI and IB-CUDA as a reference.

We report the median of the 20 samples and the 95% confidence intervals.

Figure 5.10 shows the resulting execution times using the InfiniBand network manager and compares the two model approach IB-CUDA with dCUDA. The communication overhead of InfiniBand was measured using the IB-CUDA application. The difference in execution time between IB-CUDA and dCUDA is small when running the algorithm on a single device. The algorithm distributed the work evenly on all thread blocks, which yields good GPU usage even without the fine-grained synchronisation mechanisms of dCUDA. IB-CUDA is slightly faster because dCUDA has some additional library overhead.

By scaling the problem to multiple devices, we add not only communication overhead but increase the computation time as well, due to the additional reduction of the device local multiplication results. dCUDA can mitigate a part of the communication overhead thanks to overlap of computation and communication. Because of the global barrier at the end of each iteration that synchronises all ranks on all devices, dCUDA can not reach its full potential using communication-computation overlap. Despite the global barrier, which is a worst case scenario for the dCUDA framework, dCUDA achieves a speedup of 1.4 with 4 nodes.

Figure 5.11 shows the execution time of MPI-CUDA and compares it to IB-CUDA. The performance of MPI drops using nine processes as some of them reside on the same nodes, which causes openMPI to perform poorly. Overall IB-CUDA performs better thanks to lower communication latencies. MPI might perform better for larger problem sizes, when its superior throughput comes into play.

5.7 Case Study: Power Iteration

The last case study was fully designed and implemented during this thesis. The algorithm is an extension of the sparse matrix vector multiplication algorithm. The power iteration or Von Mises iteration algorithm computes the greatest eigenvalue of a matrix A and the corresponding eigenvector [27].

The algorithm starts with a matrix A and a random vector b_0. In each iteration, the vector b_k is multiplied by the matrix A and then normalized to get a new b_{k+1}.

$$b_{k+1} = \frac{Ab_k}{||Ab_k||}$$

When the algorithm runs enough iterations, the vector b will converge to an eigenvector associated with the dominant eigenvalue if the matrix A has a dominant eigenvalue and the starting vector b_0 has a non-zero component in the direction of the eigenvector.

In the initialisation phase, the matrix A is decomposed in two dimensions and distributed among all devices. Then, we generate the initial vector b_0 and we decompose the vector into smaller sub-vectors and distribute them among the devices in the first row. Figure 5.12 shows an example decomposition of a 16x16 matrix on four devices. Device 0 and device 2 represent the first row and have the vector b_0 stored initially. Figure 5.12 indicates each step of the algorithm with numbers, which we explain in the following paragraphs. Each device stores the its own matrix

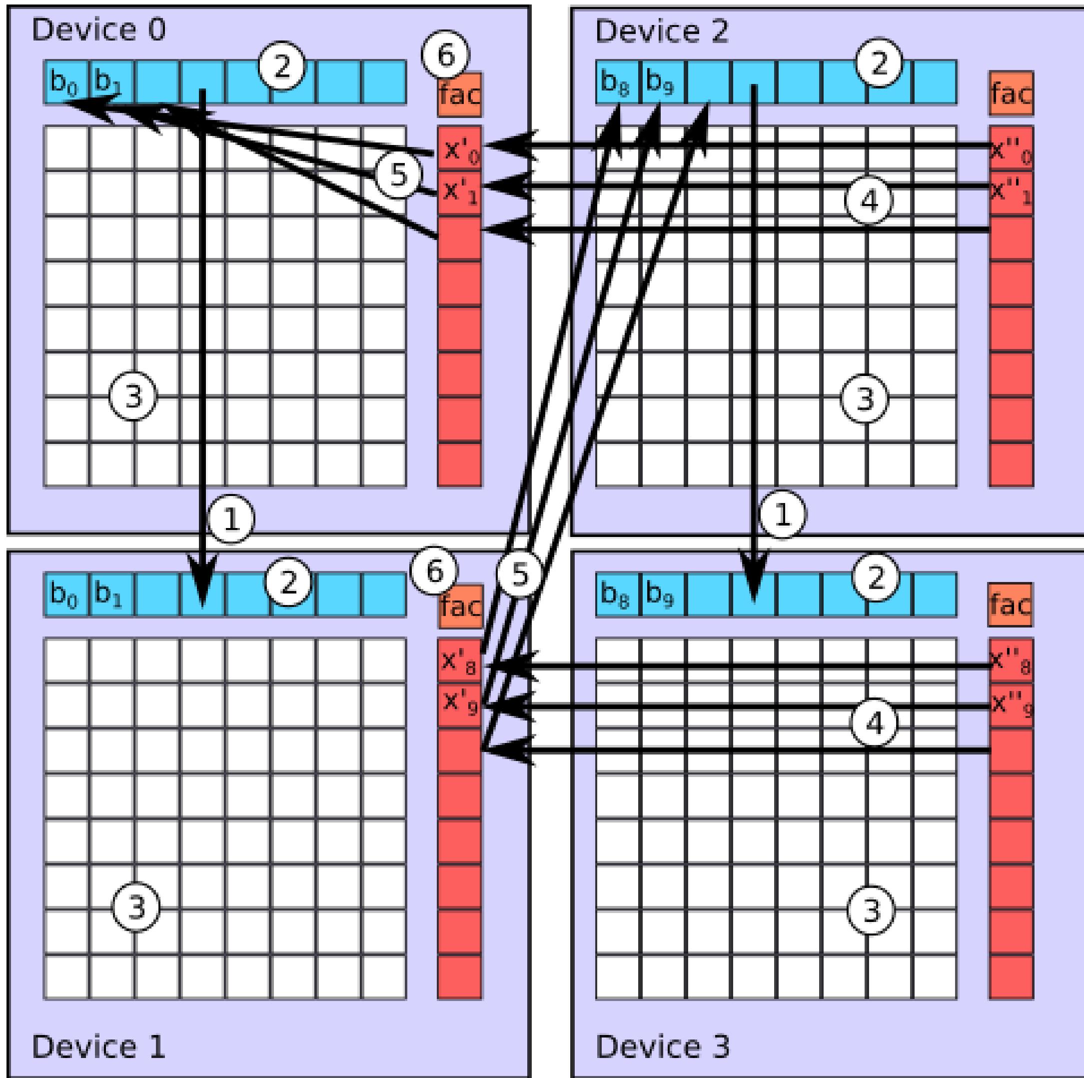


Figure 5.12: Power Iteration work steps

block and the vector b_k of the current iteration. Additionally, each device has a vector x, which holds the result of the matrix vector multiplication ($x_k = Ab_k$) and a single element called fac, which holds the factor to normalize the vector b for the next iteration.

$$fac_{k+1} = \frac{1}{||x_k||} = \frac{1}{||Ab_k||}$$

1. First, we send the vector b to the devices in the same column using a binomial tree. In the same step we also distribute the value of fac to all the devices using the same binomial tree. In Figure 5.12 only the transmission of the vector b is indicated.
2. In the beginning of this step, each device has the current vector b and the factor fac. We normalise the vector b by multiplying each element with the factor fac using all thread blocks of the device in parallel.
3. Now that we have a normalised vector b_k, we compute the matrix vector multiplication and store the result in the vector x. Each element of the vector x is accessed only by a single thread block to prevent write conflicts or additional synchronisation mechanisms.

4. To get the final result of the matrix vector multiplication of the entire matrix, we have to add up the intermediate results stored in vector x on each row. Therefore, we reduce vector x along the rows using binomial reduction trees. Finally, we will have the result of the multiplication stored in the devices in the first column, e.g. device 0 and 1 in our example in Figure 5.12.
5. We prepare the next iteration by transposing vector x of the devices in the first column to the vector b in the devices of the first row. Device 0 can just copy its vector x to vector b, but all other devices on the first column send the data diagonally to the devices on the first row.
6. Finally, we have to compute the length of the vector to derive the factor fac to normalize b. The devices in the first column compute the length by summing up all squared elements of the vector x and then reduce the result to device 0. Device 0 takes the square root of this sum and computes the division to get fac. At this point we are ready to go to step 1 again and distribute the new vector b and the value fac for the next iteration.

The implementations of the two model approach, MPI-CUDA and IB-CUDA, perform each step one after another and at the end of each step, the thread blocks on the local device are synchronised because CUDA synchronises the device after each kernel invocation. In each communication step, the processes that communicate with each other are synchronised as well. dCUDA allows synchronisation to be more fine-grained. dCUDA does not synchronise the local device at the end of each step because everything runs in a single dCUDA kernel. Communication on the level of dCUDA ranks allows fine-grained synchronisation of single ranks of the device.

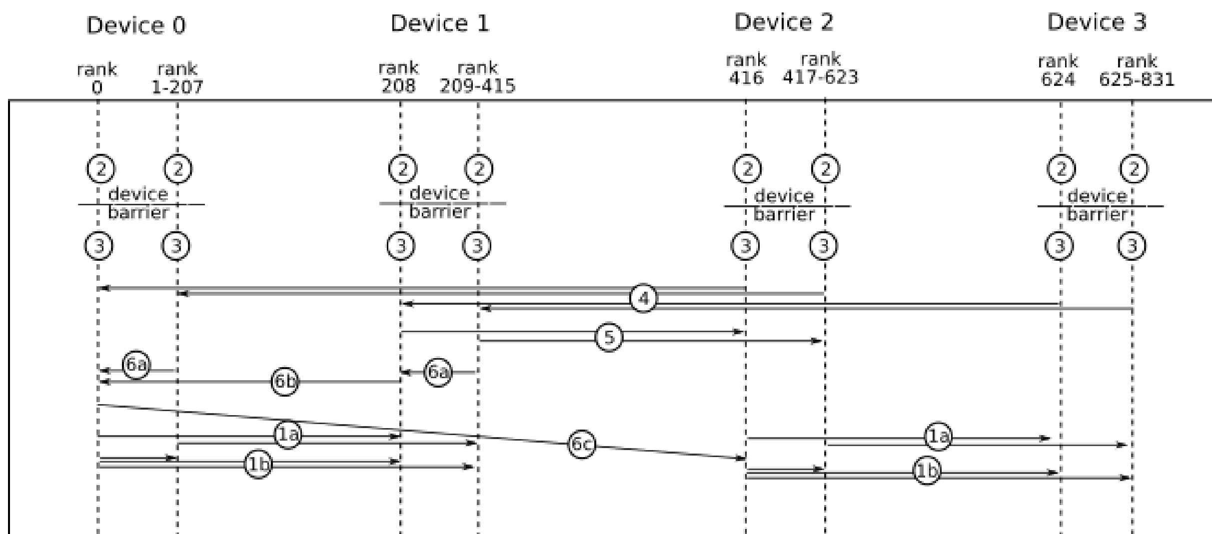


Figure 5.13: Sequence diagram of the power iteration algorithm.

Figure 5.13 shows a sequence diagram that shows the communication and synchronisation pattern of the algorithm with dCUDA. The diagram starts with step 2 to show the overlap of step 1 and step 6. The diagram shows four devices that are organised as in Figure 5.12. Each device has two time lines, one for the main rank of each device and a second time line representing all remaining ranks on the corresponding device. The dCUDA algorithm creates 208 ranks per

device. The individual ranks are not synchronised, which means that one rank can be in a different step than another rank. The only time a rank synchronises is when it waits for a notification from another rank.

At the top of the sequence diagram, we start in a state where every device has its sub domain of A stored locally, as well as the not yet normalised vector b and the factor fac, which is used to normalise the vector. In step 2, each rank individually normalises a part of the vector b in parallel. In the multiplication step, each rank needs to read the whole vector b. For that reason, we have to issue a device barrier such that in the next step every rank reads the updated normalised input.

When the ranks leave the barrier, they knows that vector b was updated and each rank individually computes a part of the matrix vector multiplication. Without any synchronisation, the ranks start to reduce their part of the result with the corresponding rank on the neighbouring device in step 4. Note that there is no device synchronisation in this step nor in the next step 5, where we transpose the resulting vector. Vector transposing on device 0 is done locally but the ranks of device 1 send the reduced result from the last step back to device 2.

In step 6 we compute the length of the resulting vector and send the factor fac to the devices in the first row. This requires multiple sub-steps. First in step 6a, the local ranks of device 0 and device 1 take the squared sum of the resulting vector in parallel and reduce the result to the local leader rank. In step 6b, the leader rank of the devices reduces the result globally to rank 0. In step 6c, rank 0 sends the factor fac to device 2. Note that all of this is done in parallel to the data transmissions of step 5, which takes some time as a large, dense vector is sent over the network.

In step 1, we distribute the new vector b and the factor fac to the devices in the same column. In step 1a, the vector b is distributed. Each rank individually sends a part of the vector to its partner rank on the other device. In step 1b, the leader rank distributes the factor fac to the leader rank of the other device. It also has to inform all other ranks of its own device and one in the same column that the new factor is ready. This could be done by sending a notification to each rank, but we used another method that uses less messages. The leader rank writes to a global flag on its device and the partner device using the function `dcuda_put`. When the ranks observe the write to the flag, they know that the factor fac is ready to read. Step 6c and step 1a can overlap because the two messages involve different data, the vector b and the factor fac.

The fine-grained synchronisation patterns of this algorithm allow a lot of overlap of communication and communication. For example, lets assume that rank 416 in Figure 5.13 is slow and needs a lot of time to perform its part of the matrix vector multiplication. Before the rank finishes the multiplication in step 3, rank 417 could already have sent its result to device 0, received its part of the new b for the next iteration from device 1 and forwards the data to device 3 in step 1a without being blocked by teh slower rank 416 and without violating any data dependencies. Therefore, a lot of computation overlaps with communication.

We performed a weak scaling benchmark with every variant of the power iteration algorithm. We used 1, 4 and 9 devices on 5 different nodes. We tested only these configurations, because they allow a two dimensional domain decomposition into a square of sub-blocks. Each device has a sparse matrix of size 10816×10816 with 0.3% non zero entries stored in CSR format. We initialised every device and measured the execution time of 100 power iterations. We report the

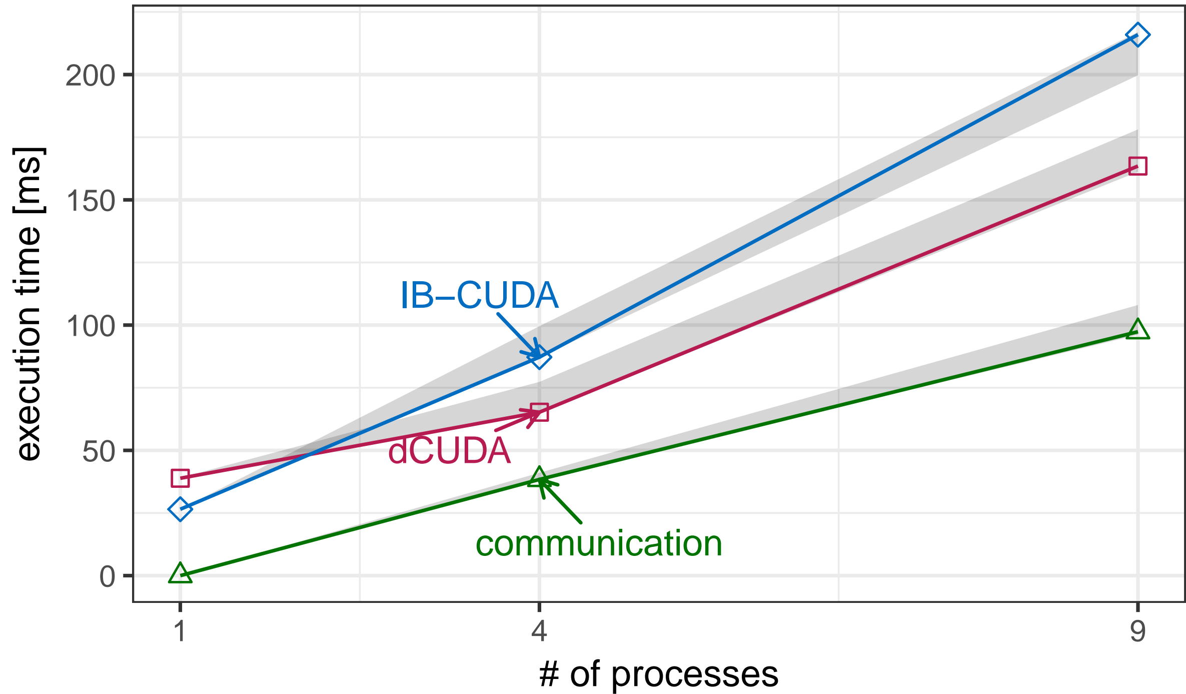


Figure 5.14: Power iteration weak scaling with InfiniBand network manager.

median of 20 samples of 100 iterations and show the 95% confidence interval in grey.

Figure 5.14 shows the result of the two model approach IB-CUDA that uses the InfiniBand Verbs network manager and the result of dCUDA with InfiniBand Verbs. Additionally, we report the communication overhead of InfiniBand measured using the IB-CUDA implementation. Note that the communication overhead is only an approximation as the sender does not wait for an acknowledgement on every send.

The algorithm has a good GPU utilisation on a single device, because the work can be evenly distributed among the thread blocks. Therefore, dCUDA can not profit from more fine-grained synchronisation patterns and additional library overhead for the more complex synchronisation pattern leads to longer execution times.

dCUDA performs much better compared to IB-CUDA when we scale the problem to more devices. dCUDA can mitigate a lot of the communication overhead thanks to overlap of communication and computation. On the other hand, the execution time of IB-CUDA is the sum of the computation time and the communication time. Note that as more devices are involved, the computation time increases as well as additional reduction steps are required.

We use a binomial trees to distribute the data and reduce the results. The height of the trees is $\lceil \log_2 \sqrt{n} \rceil$, which equals to one when using four device and two using 9 devices. This means that the communication overhead is doubled as we see in Figure 5.14 because the communication paths are twice the length. We expect that if we scaled the problem to 16 processes, the communication overhead would have increased only slightly compared to 9 processes, because the depth of the binomial trees would be two as well. With 9 devices, the communication overhead dominates the execution time and dCUDA can not mitigate all of it by overlapping it with

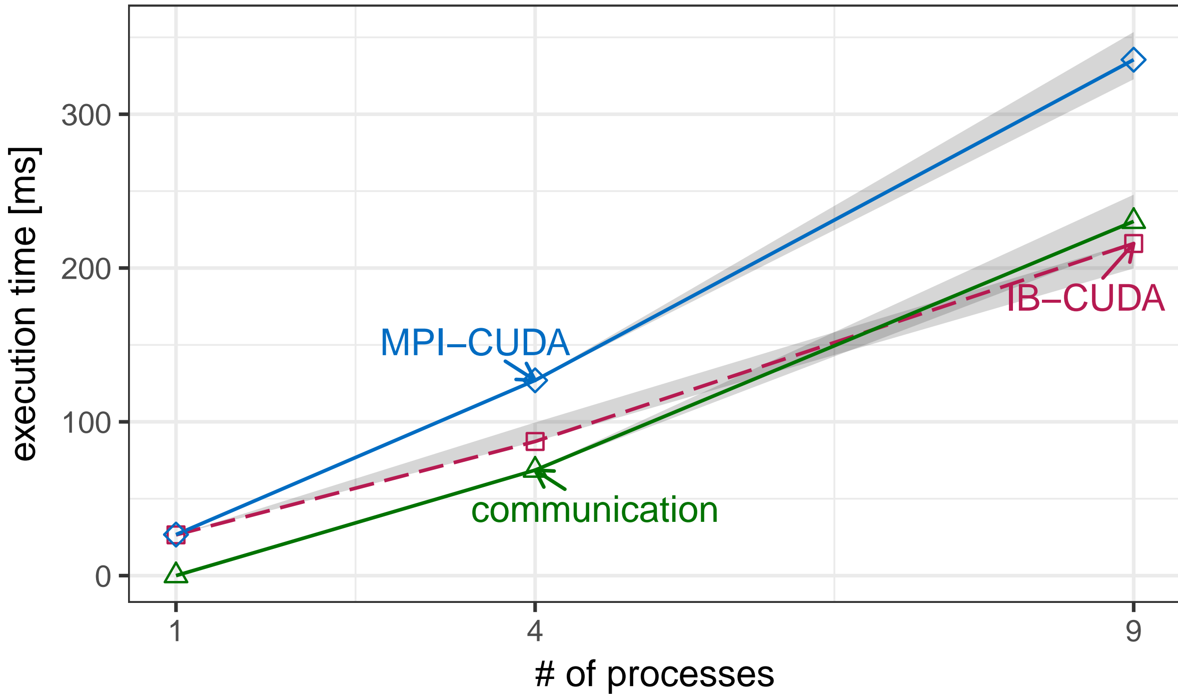


Figure 5.15: Power iteration weak scaling with MPI and IB-CUDA as reference.

computation. Nevertheless, dCUDA achieves a speedup of about 1.33 compared to IB-CUDA.

Figure 5.15 shows the result of the MPI-CUDA implementation and the communication overhead of MPI as well as the result of the IB-CUDA approach as a reference. We could not run dCUDA using MPI, as dCUDA sends a lot of small messages which MPI can not handle in our test setup.

The communication overhead on 9 processes has more than doubled compared to the overhead on 4 processes because we only used 5 nodes. The test that uses 9 processes spawns two processes per node, which leads to poor MPI performance as we showed in Section 5.2.1 about network latencies. By using the InfiniBand network manager instead of MPI we achieve a speedup of up to 1.57 with IB-CUDA and a speedup of 2.1 using dCUDA.

6

Related Work

Over the past years, several approaches and programming models for GPU cluster programming, which try to build a unified view on the distributed system, have been presented. For example, Kim et al. [12] introduced an OpenCL framework for GPU clusters. OpenCL is a unified programming model and abstracts different kinds of computation units including GPUs on a single node. The OpenCL framework extends the OpenCL model to clusters and creates the illusion of a single system. Inter-node communication is hidden from the user and uses some communication interface like MPI. The rCUDA virtualisation framework [6] tries a similar approach by introducing a GPU device virtualisation middleware that makes remote GPUs available to all nodes in the cluster. The virtualisation runtime uses the socket interface for communication with remote nodes.

An extension to OpenSHMEM [9] that supports one sided put and get remote memory operations adds support for GPUDirect RDMA features. OpenSHMEM is a partitioned global address space programming model that provides put and get operations to access remote memory. The extension adds support to allocate GPU memory inside OpenSHMEM, which uses GPUDirect RDMA to remotely access the GPU memory for low latency communication. Like CUDA-aware MPI, they used different transfer protocols for different message sizes. In contrast to dCUDA, they do not provide a device-side communication library and do not support notified access.

Other works introduce some sort of device-side communication library. For example, GPUNet [13] implements a high-level socket interface on the device. GPUNet builds on the open source rsockets library that implements sockets using RDMA over InfiniBand. GPUNet extends it with device-side sockets in GPU memory. GPUNet uses InfiniBand RDMA operations and the GPUDirect RDMA technology to send the data from GPU memory to a remote GPU device. Sending and receiving requires the help of the host, which uses ring buffers to communicate with the local device. These CPU-GPU ring buffers inspired the implementation of the command and

logging queues of dCUDA.

GGAS [23] implements RMA in a device library using a custom built network adapter that supports device to device communication without host interaction. In contrast to dCUDA, GGAS's programming model synchronises all local thread blocks of a device before any remote access, which prevents overlap of communication and computation.

GPUrdma [5] is a device-side library for RDMA without any host interaction. Modifications to the GPU driver and the InfiniBand Verbs library allow them to allocate and access the InfiniBand resources (QPs and CQs) directly in GPU memory. Because GPUrdma can issue send and receive requests directly from GPU code, communication does not need any host interaction. The advantage of GPUrdma is that the user can use the CPU for different tasks while the GPU manages all its communication by itself and the authors expect lower communication latencies as no GPU-CPU interaction is required. Unfortunately, GPUrdma uses more GPU resources and the memory access time of the GPU is higher than the memory access time of the CPU. Therefore, they achieve a latency of $5\mu s$ for device to remote host communication, which is comparable to the performance of dCUDA (6.9 μs for device to remote device).

7

Conclusions

In this thesis we addressed the design and performance of dCUDA, a unified programming model for GPU cluster programming, and we investigated traditional state of the art GPU cluster programming models and their performance. Traditional GPU cluster programming approaches use two models, the CUDA programming model for computation on local GPUs and a network communication interface like MPI. dCUDA provides a single model to control multiple GPU devices that interact with each other through shared memory windows using RMA put operations. dCUDA provides a device-side library for fine-grained communication and synchronisation on the level of CUDA thread blocks. Thanks to fine-grained synchronisation and loosely coupled thread blocks, dCUDA hides communication latencies by overlapping communication with computation and better utilises the GPU resources.

We extended the dCUDA runtime by implementing a network manager as an alternative to MPI on the basis of the InfiniBand Verbs library. The InfiniBand network manager shines with low latency RDMA functionality and low overhead. The network manager can write data to GPU memory of remote devices without interaction of the remote host and at the same time notifies the receiver using a single message. This can be achieved through the GPUDirect RMDA technology and the InfiniBand RDMA write with immediate message type.

The optimised dCUDA implementation with the InfiniBand network manager reduces the message latencies by a factor of 2 to 3 compared to the original dCUDA implementation and achieves a latency of 6.9 μs instead of 14.7 μs for a notified remote access to a remote device. Message latency for local messages on the same device is reduced to 2.4 μs from 6.4 μs.

We used the GPUDirect RDMA technology for low latency memory access. But low PCI-e bandwidth limits the performance of the InfiniBand network manager with large messages. MPI avoids this problem by using a pipelined host staging protocol for large messages instead of

GPUDirect, which yields higher bandwidth. Adding the same feature to our InfiniBand network manager would further improve the performance of dCUDA.

Communication between ranks on different devices on the same node uses the loopback over the InfiniBand network adapter and has only slightly lower latencies than messages to remote devices. By extending the dCUDA device library to use the GPUDirect peer to peer technology to access memory of a peer device, the latency could be decreased even further.

The new dCUDA implementation is designed to scale well to large compute clusters because the InfiniBand network manager uses dynamically connected communication channels. The usage of network resources is constant for each dCUDA process independent of the size of the system and each process is able to communicate with every other dCUDA process. Thanks to a sophisticated connection pool, the network manager can maintain multiple connections with low protocol overhead.

In addition to the new InfiniBand network manager, we optimised the general dCUDA design and implementation by introducing multi-threading to the host runtime system and redesigning the whole notification system. The new notification system enabled optimised low latency communication between ranks on the local device and overall decreased library overhead. New remote put and notify functions give the users more possibilities to design their applications and a new timing mechanism enables improved benchmarking of GPU applications. For example, the new notify function provides a fast synchronisation mechanism with a latency of 3.6 μs from device to remote device.

We performed multiple case studies and implemented a new dCUDA application to test and benchmark the usability and performance of dCUDA compared to the traditional two model GPU cluster programming approaches. Depending on the application and its communication pattern, dCUDA achives a speedup of 2 to 3 compared to the two model approach MPI-CUDA. The new low latency InfiniBand network manager contributes to most of the performance increase. We also tested the two model approach IB-CUDA that uses our InfiniBand network manager instead of MPI. dCUDA is 1.2 to 1.7 times faster than IB-CUDA using the same network manager. The speedup of dCUDA is larger in applications that do not require tight synchronisations of the computation units, as dCUDA enables overlap of computation and communication time.

As part of this thesis, we implemented a power iteration algorithm that calculates the eigenvector of the dominant eigenvalue of a sparse square matrix [27]. The implementation of the algorithm using the dCUDA programming model showed the advantages of device initiated communication and fine-grained synchronisation. The implementation proved the usability and correctness of the programming model and benchmarking the algorithm showed superior performance of dCUDA compared to traditional programming models.

The dCUDA framework lacks collective functions. Even though the InfiniBand network manager implements collective gather and scatter functions for host to host communication, dCUDA supports only one sided remote memory put operations. In distributed application often the same communication patterns are used, e.g. parallel reduction as we used in some of our benchmark applications. An efficient parallel global reduction is not difficult to implement using the dCUDA programming model but requires some programming effort and knowledge of the programming model. An extension to dCUDA including support for collective communication

patterns would improve the experience for dCUDA users and would reduce the development expenses of dCUDA applications. Additionally, this would open up new performance optimisation opportunities. For example, intermediate reduction steps could optionally be computed directly on the CPU instead of the GPU to avoid long device memory access delays.

Advances in GPU hardware technology and introduction of new features could enable new dCUDA optimisations. For example, system-wide atomic operations, which were introduced with the Nvidia Pascal GPU architecture, would enable optimisations on dCUDA's notifications system and would reduce the device memory footprint of dCUDA.

An interesting alternative for the host controlled InfiniBand network manager would be the integration of GPUrdma [5] into dCUDA. GPUrdma enables direct access of InfiniBand resources from device code and allows network communication without host interaction. Using GPUrdma on dCUDA devices allows us to relocate many components of the dCUDA host runtime system to the GPU, which leads to a shorter control path. The performance benefit of the integration of GPUrdma into dCUDA remains to be tested but we expect no performance increase because a CPU is generally better suited for networking tasks due to the high device memory latency over PCI-e. The low latency of the current dCUDA implementation strengthens this claim.

In this thesis we redesigned and optimised the dCUDA programming model and made it competitive to other unified GPU cluster programming models in terms of performance and showed its advantages over traditional GPU cluster programming models like MPI-CUDA. dCUDA takes GPU devices a step further away from the GPU as co-processor design and makes them first class communication endpoints.

Bibliography

[1] Jeremia Bär. GPU Remote Memory Access Programming. Master's thesis, ETH-Zürich, Zürich, 2015.

[2] R. Belli and T. Hoefler. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS'15)*. IEEE, May 2015.

[3] COSMO: Consortium for Small-scale Modeling. `http://www.cosmo-model.org`.

[4] CSCS: Centro Svizzero di Calcolo Scientifico. Swiss National Supercomputing Centre. `http://www.cscs.ch`.

[5] 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.

[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*, pages 385–394. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[7] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, Nov. 2014.

[8] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. dCUDA: Hardware Supported Overlap of Computation and Communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 52:1–52:12, Piscataway, NJ, USA, 2016. IEEE Press.

[9] Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, Hari Subramoni, Ching-Hsiang Chu, and Dhabaleswar K. Panda. CUDA-Aware OpenSHMEM: Extensions and Designs for High Performance OpenSHMEM on GPU Clusters . *Parallel Computing*, 58:27 – 36, 2016.

[10] Evatt R Hawkes, Ramanan Sankaran, James C Sutherland, and Jacqueline H Chen. Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. *Journal of Physics: Conference Series*, 16(1):65, 2005.

[11] InfiniBand Trade Association. *InfiniBand Architecure Specification Annex: Extended Reliable Connected Transport Service*, 1.2.1 edition, 2009. available at: https://cw.infinibandta.org/document/dl/7146.

[12] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. *OpenCL as a Programming Model for GPU Clusters*, pages 76–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[13] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, Broomfield, CO, 2014. USENIX Association.

[14] Justin Luitjens. Faster parallel reductions on kepler. In *Parallel Forall*. NVIDIA Corporation. Available at: https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/.

[15] Mellanox. *Intro to InfiniBand for End Users*, June 2010. Available at: http://www.mellanox.com/pdf/whitepapers/Intro_to_IB_for_End_Users.pdf.

[16] Mellanox. *Mellanox OFED for Linux User Manual*, 3.1 edition, 2015. available at: http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v3.10.pdf.

[17] Mellanox. *Mellanox GPUDirect RDMA User Manual*, 2016. available at: http://www.mellanox.com/related-docs/prod_software/Mellanox_GPUDirect_User_Manual_v1.3.pdf.

[18] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, September 2012. Available at: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[19] NVIDIA Corporation. NVIDIA GPUDirect. In *NVIDIA CUDA Zone*. Available at: https://developer.nvidia.com/gpudirect.

[20] NVIDIA Corporation. *CUDA C Best Practices Guide*, 7.0 edition, 2015. Available at http://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[21] NVIDIA Corporation. *CUDA C Programming Guide*, 7.0 edition, 2015. Available at http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[22] NVIDIA Corporation. *gdrcopy: A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology* , 2015. Available at `https://github.com/NVIDIA/gdrcopy`.

[23] L. Oden and H. Froning. 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.

[24] OpenMPI. Running CUDA-aware Open MPI. Available at: `https://www.open-mpi.org/faq/?category=runcuda`.

[25] Hari Subramoni, Khaled Hamidouche, Akshey Venkatesh, Sourav Chakraborty, and Dhabaleswar K. Panda. *Designing MPI Library with Dynamic Connected Transport (DCT) of InfiniBand: Early Experiences*, pages 278–295. Springer International Publishing, Cham, 2014.

[26] The CP2K Developers Group. CP2K, 2012. Available at: `http://www.cp2k.org/`.

[27] Richard von Mises and H. Pollaczek-Geiringer. Praktische Verfahren der Gleichungsauflösung. *ZAMM - Zeitschrift für Angewandte Mathematik und Mechanik*, (9):152–164, 1929.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

dCUDA: GPU Cluster Programming using IB Verbs

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Kuster	Lukas

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
17.03.2017	[signature]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.