Master Thesis

GPU Remote Memory Access Programming

Author[s]:
Bär, Jeremia

Publication Date:
2015

Permanent Link:
https://doi.org/10.3929/ethz-a-010571868

Rights / License:
In Copyright - Non-Commercial Use Permitted
GPU Remote Memory Access Programming

Jeremia Bär

Master Thesis
October 2015

Supervised by
Prof. Torsten Hoefler
Tobias Gysi

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

High performance computing studies the construction and programming of computing system with tremendous computational power playing a key role in scientific computing and research across disciplines. The graphics processing unit (GPU) developed for fast 2D and 3D visualizations has turned into a programmable general purpose accelerator device boosting today’s high performance clusters. Leveraging these computational resources requires good programming model abstractions to manage system complexity. Today’s state of the art employs separate cluster communication and GPU computation models such as MPI and CUDA. The bulk-synchronous nature of CUDA and the role of the GPU as a CPU-dependent co-processor limits cluster utilization. In this master thesis we devise, implement, and evaluate the novel GPU cluster programming model GPU RMA addressing three key areas. GPU RMA introduces a simplifying abstraction exposing the cluster as a set of interacting ranks. The ranks execute on the GPU removing the complexity of explicit GPU management from the host. GPU RMA introduces communication amongst ranks on local and remote devices allowing more fine-grained communication and synchronization compared to conventional models increasing concurrency and resource usage. GPU RMA implements one-sided notified access communication leveraging the recently introduced RDMA support for GPUs providing richer communication semantics than current solutions. We implemented GPU RMA on a cluster with Intel CPUs, Nvidia GPUs, and an InfiniBand interconnect, studied the impact of system components to communication latency, and derived an empirical performance model. We performed a case study to assess GPU RMA performance compared to the established MPI and CUDA approach and validated our performance model. Our experiments show GPU RMA improves performance up to 25% compared to the state of the art and encourages further study of GPU RMA performance and scalability.
Acknowledgements

I would like to thank my advisor Tobias Gysi for the numerous discussion putting my work into a broader perspective as well as his readiness to respond to development challenges. I appreciate your relaxed and humorous mentoring.

I would like to thank Hussein N. Harake from the Swiss National Supercomputing Centre and administrator of the greina cluster for his excellent support and flexibility regarding the cluster installation. Quick to respond to any requests investigating hardware issues, updating broken software and deploying custom kernel modules your work allowed me to exploit the most recent technology available with great support on my side.
Contents

List of Figures vii
List of Tables ix
List of Listings xi

1 Introduction 1

2 GPU Cluster Programming 5
  2.1 State of the Art ........................................ 5
  2.2 The GPU RMA Programming Model ...................... 8
    2.2.1 Motivation ........................................ 8
    2.2.2 Design Questions .................................. 10
  2.3 Specification .......................................... 14
    2.3.1 Terminology ........................................ 14
    2.3.2 Memory Model ..................................... 15
    2.3.3 Library Interface .................................. 16
  2.4 Examples ................................................ 23

3 Implementation 29
  3.1 GPU Programming Fundamentals ........................ 29
    3.1.1 Device-Level Scheduling and Synchronization ...... 29
    3.1.2 Memory Model ...................................... 30
    3.1.3 GPUDirect and CUDA-aware MPI ..................... 32
    3.1.4 Host and Device Memory Access ..................... 34
  3.2 System Architecture ................................... 36
    3.2.1 Network Communication ............................... 36
    3.2.2 Device and Host Interaction ......................... 38
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3</td>
<td>Notified Access</td>
<td>39</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Queue Design and Notification Matching</td>
<td>45</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Memory Organization</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Deployment and Portability</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>Evaluation</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>System Latency Analysis</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>GPU RMA Performance Model</td>
<td>60</td>
</tr>
<tr>
<td>4.3</td>
<td>Case Study: Horizontal Diffusion</td>
<td>62</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Algorithm</td>
<td>62</td>
</tr>
<tr>
<td>4.3.2</td>
<td>HD Performance Model</td>
<td>63</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Implementation</td>
<td>69</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Results</td>
<td>71</td>
</tr>
<tr>
<td>5</td>
<td>Related Work</td>
<td>79</td>
</tr>
<tr>
<td>5.1</td>
<td>GPU Cluster Programming</td>
<td>79</td>
</tr>
<tr>
<td>5.2</td>
<td>GPU-controlled Communication</td>
<td>81</td>
</tr>
<tr>
<td>5.3</td>
<td>Inter-GPU Communication</td>
<td>82</td>
</tr>
<tr>
<td>5.4</td>
<td>Single-GPU Programming</td>
<td>83</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>89</td>
</tr>
</tbody>
</table>
List of Figures

3.1 Multi-GPU Communication Scenarios ........................................ 32
3.2 GPU RMA System Architecture ................................................. 38
3.3 Notified Access Sequence Diagram ............................................. 41

4.1 greina System Configuration .................................................. 54
4.2 Phases of a HD-OVERLAP Iteration .......................................... 67
4.3 HD Local Performance .......................................................... 73
4.4 HD Remote Performance ......................................................... 75
4.5 HD-OVERLAP Remote Performance ........................................... 76
4.6 HD-OVERLAP and HD-N00VERLAP Comparison ............................... 77
List of Tables

4.1 System Latency Overview (Roundtrip Time) . . . . . . . . . . . . . . . . 55
4.2 GPU RMA Performance Model Parameters . . . . . . . . . . . . . . . . 61
4.3 Overview of the Horizontal Diffusion Model Parameters . . . . . . . . . 65
4.4 GPU Resource Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
List of Listings

2.1 GPU RMA Library Initialization: Host API .......................... 17
2.2 GPU RMA Library Initialization: Device API ...................... 19
2.3 GPU RMA Memory and Window Management API ............... 21
2.4 GPU RMA Notified Access API .................................. 22
2.5 Remaining GPU RMA API ....................................... 23
2.6 Hello World in GPU RMA ....................................... 24
2.7 Rank Exchange Host Code ...................................... 25
2.8 Rank Exchange Device Code .................................... 26

3.1 Host Event Handler ............................................. 40
3.2 Device-side Queue Implementation .......................... 47
3.3 PSG Node Topology ........................................ 52

4.1 Horizontal Diffusion Variations ................................. 64
Introduction

High performance computing examines how to build supercomputers with a tremendous computational power and investigates models that allow skilled programmers to implement computational tasks in a way that fully leverages the computational resources of the system. High performance computing systems are built from individual computers called compute nodes connected via a high-performance network to form a compute cluster. To make use of the computational resources an application programmer must understand how to divide a computational tasks into independent subtasks that can be handled by the individual compute nodes and how these nodes must communicate in order to complete the overall computational goal. A programming model presents a view of the compute system to the programmer that he employs to describe how the computation should be performed. In this master thesis we investigate a novel programming model for specialized compute clusters equipped with graphics processing units.

Graphics processing units (GPUs) have emerged as specialized hardware accelerator devices in the 1990s to enable fast 2D and 3D visualizations in personal computers. The regular nature of visual computations are also found in many scientific computations and soon researchers started to use the GPU functionality for their own non-graphic related applications which became known as general purpose GPU programming (GPGPU). Nvidia, one of the leading manufacturers of GPUs noticed this trend and promoted these new applications by introducing a general purpose programming model and development infrastructure for their GPUs called CUDA in 2007. CUDA exposes the GPU as a data-parallel compute device where a set of threads performs the same set of operations collectively but on different data, hence data-parallel. While this massive parallelism is a different way of expressing computation than on traditional computers, it applies well to computations scientific research faces and we find graphics processing units deliberately optimized for high performance computing as accelerator devices in todays supercomputers. The name ceases to remind of the historic origin of
1 Introduction

Programming GPU-enabled compute clusters is a complex task, starting from managing a single GPU on a compute node, understanding the performance impact of different network interconnects in communication performance, to maintaining the cluster software stack. The GPU and its host have separate physical memory with limited mutual access, requiring manual memory management by the programmer. Both the GPU and host employ their own virtual address space and while different ways of mapping each others physical memory exist, memory must be referenced careful with respect to the correct virtual memory address and memory access incurs varying latency depending on the access method chosen. The resulting partitioning of application state amongst the different physical memory location in favor of system performance imposes additional complexity on the application programmer. Inter-GPU communication to exchange computational results with other GPUs in the compute cluster is heavily dependent on the cluster topology and determined by several interconnects. A GPU is typically connected to its host over PCI-E. For communication amongst cluster nodes, a high-efficiency network such as InfiniBand is installed. Multi-socket compute nodes may require forwarding of communication between the sockets via a CPU interconnect such as QuickPath for Intel CPUs. All these interconnects have their own performance properties dependent on the actual product installed and their interoperability varies. For the programmer there is no one size fits all solution to optimize an application as characteristics are different amongst clusters and building parameterizable systems requires advanced domain knowledge. Finally, compute clusters are a scarce and expensive resource constantly evolving and shared by many scientists that doesn’t usually allow for a customized software stack requiring developers to compromise on implementation flexibility. We faced several of these typical challenges in cluster configuration ourselves during the development of our novel programming model. Initial tests on our development system returned terrible system performance. Contacting the system administrator revealed the GPUs were installed with outdated parts limiting system performance. Later in the process, hardware-supported inter-GPU communication refused to operate and contacting Nvidia eventually revealed they had released their driver with the feature broken, rendering our development system unusable. Thankfully, we had a very supportive system administrator reverting the driver installation. Our work also required experimental kernel modules by Nvidia not officially part of CUDA to enable lower latency device interaction. While not possible on common compute clusters our system administrator quickly deployed the requested modules and we appreciate his ongoing support. After all, GPU-enabled compute clusters are a highly complex system to manage and requires good programming model abstractions for an application programmer to manage the complexity. Furthermore, such systems require a sound mathematical performance model to allow the application programmer to predict application performance and enable application optimization.

A popular communication model for compute clusters is message passing as defined by the message passing interface MPI. Individual actors called ranks can communicate by exchanging messages. The ranks are mapped onto the compute nodes of the compute cluster. This communication model resembles well the actual compute cluster architecture as individual compute nodes share no common memory and require explicit mes-
sage transfers over the network. In MPI, one rank sends a message and waits until its communication peer explicitly states to receive the message. As both peers are involved this communication is known as two-sided communication. Another popular communication model leverages the networking hardware feature remote direct memory access (RDMA) that allows to directly access memory of a remote compute node without active involvement of its operating system or user application. RDMA avoids the synchronization between the two communicating peers known from two-sided communication and programming models based on it are known as one-sided communication. Various realizations of one-sided communication models yield different semantical richness. We will cover the differences in this report.

In this master thesis we focus on compute clusters equipped with graphics processing units. Compute clusters have a long tradition while general purpose GPU programming emerged during the last decade. GPUs were introduced to improve the computational power of compute node and accelerate overall computation time. They are installed as a coprocessor that cannot function on its own but is fully managed by the host CPU. Today’s state of the art in programming a GPU-enabled compute cluster uses a two-model approach. An established cluster communication model enables data exchange between compute nodes whereas on each node a GPU programming model allows to offload the computation onto the GPU. A typical combination is MPI and CUDA. A particular focus during the last five years of research was to improve the operability of MPI and CUDA. While this combination allows to improve performance compared to conventional clusters, this combined approach limits full resource utilization. Many see the need for a tighter integration of GPUs into cluster programming but combining MPI and CUDA remains the prevalent approach. We will cover proposed alternatives in our report. In this master thesis we analyzed the limitations of the current solution and propose the novel GPU cluster programming model GPU RMA.

GPU Remote Memory Access (GPU RMA) strives to improve GPU cluster programming in three areas. We provide a new programming model abstraction that avoids the two-model approach and presents the GPU cluster resources as a single large system. We drew inspiration from previous work on MPI and model the cluster computational resources as a set of communicating ranks. Different from MPI, the ranks execute directly on the GPU and our model abstracts the memory management and data transfer complexities faced in GPU cluster programming so the application programmer may focus on describing the computational task. Second, we tackle the limitations of synchronization between MPI and CUDA as well as within the CUDA model itself. The MPI and CUDA approach requires to bulk-synchronize all computation on the GPU with the host to ensure correctness. Furthermore, computational units known as blocks in the CUDA programming model may not communicate outside of host synchronization, limiting the amount of work they can perform until they require communication. In GPU RMA we introduce a more fine grained synchronization of the massive parallelism of the GPU and its interaction with the host enabling more concurrency and higher GPU utilization. Our synchronization allows a more flexible use of the GPU resources than the bulk-synchronous CUDA model. Finally, Nvidia introduced RDMA support for GPUs in 2013 which was adopted for InfiniBand networks by the network manufacturer Mellanox in 2014. As a third goal we investigate how to enable a one-sided communication model
1 Introduction

controlled directly from the GPU. All in all, we introduce the GPU as a first-class computing device into the GPU cluster programming model that acts independent and may communicate with its peers. The model allows for more fine-grained synchronization than the current state of the art investigates the novel RDMA support for GPUs.

The contributions of this master thesis are the following: We study the current limitations of GPU cluster programming and develop a novel GPU cluster programming model. Our investigation surfaces several model design questions regarding parallelism granularity and library design not apparent in the current approach. We develop a specification for the programming model accompanied with a library implementation that allows to assess our novel solution. We provide insight into the key challenges faced implementing our model and highlight design variations. We evaluate our novel GPU cluster programming model and provide a detailed analysis of GPU RMA communication latency in dependence of the different interconnects. We provide a GPU RMA performance model that enables the use of analytical methods to predict GPU RMA-based application performance and allows for guided application tuning. Finally, we performed a case study applying GPU RMA to the horizontal diffusion algorithm applied in climate modelling. The case study entails the comparison of GPU RMA to the state of the art MPI and CUDA programming model and uses the GPU RMA performance model to predict application performance and validate our model parameters. We implemented GPU RMA for Nvidia GPUs on a compute cluster with Intel nodes and an InfiniBand interconnect. The library was developed and evaluated on the greina cluster of the Swiss National Supercomputing Center.

The rest of this thesis report is structured as follows. Chapter 2 reviews state of the art GPU cluster programming, discusses our novel GPU RMA model, and provides a model specification. Chapter 3 reviews GPU programming fundamentals and describes our library implementation. Chapter 4 presents our evaluation results. Chapter 5 covers related work. Chapter 6 concludes our work.
GPU Cluster Programming

In the last decade we have seen the rise of GPU-enabled compute clusters. We review the current state of the art of GPU cluster programming. We follow by highlighting some of the deficiencies in the current approach and introduce our novel GPU RMA programming model avoiding these limitations. We conclude with two programming examples of our proposal.

2.1 State of the Art

The GPU has evolved as a powerful coprocessor in HPC clusters enabling massively parallel computations. Computers called compute nodes are connected via a fast network to form one large a compute cluster. The GPU is a slave device installed on a compute node and explicitly managed by a process as a local resource. The common approach to program a GPU-enabled cluster is to combine a middleware such as MPI for cluster communication and offload node-local computations to the GPU employing a GPU programming model such as CUDA. In this section we provide an introduction to MPI and CUDA.

**MPI** The Message Passing Interface (MPI, [17]) is a widely deployed communication standard available for a variety of hardware architectures and network technologies that was released over twenty years ago and is constantly refined. The standard defines an interface and competing implementations are provided by hardware vendors and academia improving performance as hardware evolves.

In MPI, a set of ranks cooperate to accomplish a common computational task. A rank is commonly implemented as a process running on one of the compute nodes in the
cluster. Ranks are independent in their execution and share no state or memory with other ranks. To communicate information, ranks exchange messages. An origin rank sends a message specifying the data, a target rank and freely chosen tag value. To receive the message from the origin rank, the target rank needs to explicitly receive the message and obtains a copy of the origin data. This rendez-vous communication unifies three important properties for concurrent programming. Most apparent, sending invokes a data transfer from one rank to the other. Second, the completed send or receive call guarantees memory synchronization so the target is guaranteed all data is valid and the origin may modify its send buffers without influencing the target. Finally, the explicit matching of a send and receive call implies rank synchronization. The three properties are not always guaranteed by other communication means as we will shortly see. Because message passing requires a rendez-vous of origin and target, it is also known as two-sided or point-to-point communication. Ranks are uniquely identified with respect to a communicator. Subgroups of communicating ranks can be defined based on the global communicator containing all ranks. An introduction into MPI programming is given in [8].

The message passing interface translates well onto uniprocessor networks with disjoint memory, yet recent advances in computing hardware are not well reflected in its programming model. In symmetric multiprocessing, independent compute cores from multi-core processors and multi-socket machines share access to common system memory. Performing explicit data copies for message passing introduces unnecessary overhead for certain applications. Hardware support to directly read or write memory from a remote compute node such as RDMA on InfiniBand promoted by the Open Fabrics Alliance [32] does not require the target machine to explicitly receive the data or initiate a response in software. In MPI, the programming model requires communication be matched at the target. In search for new programming models several solutions have emerged. One of them is Remote Memory Access (RMA) part of MPI. In MPI RMA, memory regions can be exposed as windows for remote memory access from other ranks. All ranks of a communicator collectively create a window object and the provided window from each rank may then be accessed using RMA operations such as put or get. Because the target doesn’t need to confirm the memory operations, they are also known as one-sided operations. This approach decouples data transfer, memory synchronization and rank synchronization that we mentioned for MPI message passing. Three ways for memory and rank synchronization exist for different rank granularity levels. An introduction to RMA programming including a discussion of benefits and drawbacks compared to message passing is provided in [7].

MPI RMA introduced a programming model that leverages modern hardware features. The interface still prohibits an efficient implementation of certain application as Belli and Hoefler [2] have recently shown. In consumer-producer applications, one often wants to communicate a single result to a peer rank. MPI RMA decoupled data transfer from memory synchronization and rank synchronization. Informing a peer rank that results are available while guaranteeing data is fully visible if only a single data transfer is involved yields comparable overheads to message passing in spite of modern hardware features. They devise notified access, a special set of RMA operations that combine synchronization with data transfer and completes RMA operations with a notification
at the target rank. This allows the target rank to test for received notifications which imply data transfer and memory synchronization. The notification carries a tag value known from message passing that allows for fine-grained synchronization.

The notions of one-sided and two-sided communication are present both in programming models and hardware implementations. In programming, one-sided communication removes the need for the communicating peer to actively respond to the communication request. In hardware, one-sided communication moves work from the CPU to the network adaptor, allowing for faster implementations without involving the operating system. The two models can be mixed between programming models and hardware implementations. For instance, two-sided communication in MPI is often implemented by one-sided operations on the underlying InfiniBand network. On the other hand, two-sided implementations may yield better performance for one-sided communication models as the evaluation of GPU RMA will show.

**CUDA** The Compute Unified Device Architecture (CUDA, [24]) is a programming model, runtime system, and development toolchain from Nvidia for their general purpose graphic processing units. Available features depend on the CUDA version and GPU model. The three product brands *GeForce*, *Quadro* and *Tesla* target the consumer, professional and high-performance markets respectively and have different features enabled. The high-performance market has seen three generations of GPU microarchitectures named named *Tesla*, *Fermi* and *Kepler* with Kepler being the latest. Many scientific results still refer to Fermi GPUs. Note that Tesla refers to both the brand name as well as the first GPU microarchitecture. The virtual machine model and instruction set architecture of Nvidia GPUs is known as *Parallel Thread Execution (PTX)*.

The CUDA programming model exposes the parallelism of the GPU as a hierarchical group of threads with shared memory and barrier synchronization. *Threads* are organized in independent *blocks*. A set of blocks is known as *grid* or *kernel*. The programmer may describe the computation of a kernel by a set of C language extensions and launch the kernel on the GPU. Threads in the same block can access shared memory and perform barrier synchronization. There exists no means for shared memory access or synchronization amongst threads in different blocks and scheduling cannot be controlled.

A barrier synchronization amongst all blocks is performed at the end of kernel execution and results from a kernel are guaranteed to be visible to the next kernel launched. Memory synchronization is a side effect of kernel launches. Running kernels do not overlap by default.

On the GPU, blocks are executed by *streaming multiprocessors* (SMX). A streaming multiprocessor can execute several blocks concurrently, dependent on the launched kernel and the GPU model. The scheduling unit is a set of 32 threads from the same block called a *warp*. Fast context switching is a key feature of Nvidia GPUs and a different warp from a different block may be scheduled to execute in each clock cycle. The fast context switching is important to enable *latency hiding*. On a traditional CPU, a cache provides fast memory access and a lot of chip area is dedicated to cache. The GPU has more computational units and less cache. Rather than reducing memory access time, fast context switches allow to hide the memory latency and let other threads progress.
A CUDA thread is very different from a CPU thread. On the CPU, a single thread controls execution and task-parallel vector instructions are executed by a one thread. Vector memory accesses are coalesced and memory synchronization applies to all data accessed in the thread. On the GPU, all threads in a block share a memory synchronization context. Threads are not independent but execute together at warp-granularity like a single vector instruction on the CPU. Memory access is only efficient when neighboring threads access consecutive memory. While in the programming model the computation is described for each thread individually, the computation suffers from severe performance drawbacks if the threads follow a different code path known as warp divergence. A CPU program can control vector instructions by a mask so computation is only active on some vector fields. The same principle applies to GPU warps but the thread masking is handled by the hardware scheduler during execution.\(^1\) The comparison shows a CPU thread is more accurately translated into a CUDA block rather than a thread from a programming perspective.

We call the compute node a GPU is installed the host and the GPU the device. The device has distinct memory referred to as device memory while the hosts’ main memory (RAM) is called host memory. Device memory is explicitly managed from the host process but may not be accessed directly. The CUDA runtime provides dedicated malloc and memcpy functions to interact with device memory.

Nvidia provides a variety of documentation for their GPUs and development infrastructure. This summary is based on the CUDA C Programming Guide [24], the Parallel Thread Execution manual [28], their ParallelForall developer blog [10], and current architectural white paper [22]. We will cover more details of CUDA and related technologies in the implementation section introducing the device memory model (Section 3.1.2), inter-GPU communication (Section 3.1.3), shared memory access between host and device (Section 3.1.4), and inter-block synchronization (Section 3.1.1).

### 2.2 The GPU RMA Programming Model

Today’s state of the art when programming a GPU cluster combines a host communication model and a GPU programming model. Often, this is MPI and CUDA. In this section we present the challenges of the current approach and devise our novel GPU cluster programming model GPU RMA. We discuss the key design decision of the model and follow with a specification of the final design in Section 2.3.

#### 2.2.1 Motivation

When combining MPI and CUDA, the main actor in the system is the host process. The host initiates communication with other ranks and launches computational kernels on the local GPU. The GPU serves as a coprocessor and cannot manage its own data or

\(^1\)The CUDA debugger allows to view the instruction mask for each warp of a running execution highlighting the similarity of GPU warps and CPU vector instructions.
communicate with other ranks. It is the host’s responsibility to synchronize with other ranks and the local GPU. Given the distinct memory spaces on the host and the GPU, the host must also ensure proper memory coherence between communication using MPI and computation using CUDA. While these tedious memory management tasks have been greatly simplified by the advent of CUDA-aware MPI [14, 20] and Unified Memory [11], the programming model still limits full utilization of the infrastructure. With GPU RMA, we strive to remove some of these limitations and advance the state of the art in GPU cluster programming. In particular, GPU RMA provides a simplified view of the GPU cluster, allowing for higher parallelism due to more fine-grained synchronization and takes into account the recent trend of remote memory access programming.

One goal of GPU RMA is to abstract the inherent hardware-dictated two-level hierarchy of the MPI and CUDA approach. Smart interaction of computation and communication is a key to leverage the resources of a compute cluster. Unfortunate for the MPI and CUDA approach, these two aspects are completely separated from a programming model point of view as well as from an implementation perspective. Often in practice, communication is managed by the host and computation is performed by the GPU only. In this model, the host is a vital component for the system to function. However, from a programmer’s point of view the host is an unavoidable component that must be programmed to use the computational resources and introduces additional complexity. Its management is tedious and requires profound understanding of both MPI and CUDA to implement correct and efficient systems. In GPU RMA, we seek to remove the host as a mediator for communication and provide this functionality directly from within a GPU program. This removes the task of explicitly synchronizing host communication with GPU computation as well as manual buffer management from the programmer. Introducing the GPU as a first-level communication partner raises an interesting design question: What is the communication target? The GPU could represent a single or multiple independent actors to communicate with. We target this question in Section 2.2.2 emerging with a view of the GPU cluster as as a collection of equally privileged communication partners executing on the GPUs.

Another goal of GPU RMA is to improve parallelism by providing more fine-grained synchronization mechanisms than the MPI and CUDA approach. This is apparent on two levels, the CUDA kernels and the CUDA blocks. In the CUDA programming model, the programmer can leverage the GPU by launching a kernel on the GPU. The programming model strongly resembles the bulk-synchronous parallel model established by Valiant [43]. The CUDA kernel executes independent CUDA blocks that compute on local data. The blocks may not communicate amongst each other. In the end, control is returned to the host, implicitly performing a global barrier amongst them. The host may now launch a new kernel based on the previous results. This model is inherently serial, alternating between parallel computation and global synchronization. In GPU RMA, we decided to break this coarse synchronization. We removed the notion of kernel launches and implicitly launch a single CUDA kernel that runs during the whole execution, removing the kernel block barrier. The reader may immediately ask how this

---

2 CUDA allows to run multiple kernels in parallel using CUDA streams [23, Sec. 10.2] but these kernels may not interact. The model remains unchanged. Results from a CUDA kernel may only be used once the kernel has ended.
is supposed to work given the lack of any other communication means. In GPU RMA, we add support for communication between CUDA blocks enabling more fine-grained synchronization. While not foreseen in the CUDA programming model, a careful study of the memory architecture and kernel scheduling shows it can be done. We provide more details on this in Sections 3.1.1 and 3.1.2 of the implementation chapter. To summarize, our design allows blocks to run more independently and share results on the way by removing the global block barrier of kernel launches and introducing inter-block communication.

The last focus of GPU RMA is to investigate the recent trend of remote memory access programming in the context of GPUs. The last decade has seen improved hardware support for one-sided communication and programming models have evolved to leverage the new hardware features (cf. Section 2.1). In 2013, Nvidia announced one-sided hardware support for GPU device memory called GPUDirect RDMA [26]. Following these trends, we decided to accompany the hardware support with an according programming model in our work. We take up the question of one-sided communication in GPU RMA in Section 2.2.2.

### 2.2.2 Design Questions

Moving control from the host to the device raises to question of how to represent a GPU in the programming model. Today's Kepler-class Nvidia GPUs run more than 10'000 threads concurrently to leverage all computing resources. We address the question of how to organize this parallelism and identify reasonable communication targets on the GPU. The design choice will require a precise definition of library calling conventions to guarantee correctness. We close with a discussion of the remote memory access variation chosen.

**Parallelization Granularity** The CUDA programming model defines three hierarchical parallelization levels: kernels, blocks and threads. In GPU RMA we must decide for a self-contained parallelization unit that serves as a communication target. A more fine-grained choice allows for more independent actors to compute and communicate. From a CPU point of view, a thread seems the straightforward choice. But a CUDA thread is not independent. We have seen in Section 2.1 that a CUDA thread executes in lock-step together with other threads as a warp, translating more accurately into one field of a warp-sized vector instruction. While each thread's execution can be described independently, it causes other threads on the same warp to stall and introduces a severe performance penalty when following a different code path. Access to global memory is slow and inefficient when performed by a single thread as full utilization of the memory bus is only achieved if neighboring threads access global memory collectively, generating coalesced memory accesses. For these reasons we don't consider a CUDA thread a reasonable choice for a self-contained parallelization unit.

The next parallelization level in the CUDA programming model is a block. The reader might ask if a warp wouldn't be a reasonable choice as well. We note that a warp is
the smallest scheduling unit on the GPU hardware but lacks a corresponding representation in the programming model and has no independent memory context. Memory coherence and thread synchronization is available on thread- and block-level, but not for individual warps. Warp granularity is also achieved by choosing warp-sized blocks, making individual warp programming obsolete and revealing they primarily represent a hardware scheduling unit.

A CUDA block is a well-defined self-contained unit in the CUDA programming model and the key to program scalability. Independent blocks can be flexibly scheduled onto the GPU computing resources, allowing to adapt for the capabilities of the various GPU models. A block consists of several threads, usually the multiple of the warp size, allowing flexible hardware scheduling or coalesced memory accesses. The GPU provides block-level memory fences and barrier synchronizations for its threads. CUDA shared memory is fast block-private memory accessible to all threads in the block. Its properties make the CUDA block a good choice as a parallelization unit and communication target. One might object block granularity decreases GPU utilization if a block waits for communication and all threads in the block idle. This doesn’t necessarily lead to GPU underutilization. Multiple blocks execute on the same streaming processor that is oversubscribed by the hardware scheduler. For instance, a current Kepler-class GPU can only run 192 single-precision operations yet manages over thousand threads concurrently. Idling threads that wait for memory operations are postponed and a different warp is scheduled (latency hiding), allowing overall progress. Due to the oversubscription the stalling of one block that waits for communication is unlikely to cause underutilization.

The top GPU parallelization level is a kernel. As kernels are launched from the host, implementing a GPU communication scheme within a kernel is not feasible. CUDA Dynamic Parallelism [24, App. C] is a recent addition to CUDA that allows to launch CUDA kernels from a thread running on the GPU. While giving more control to the GPU, the bulk-synchronous programming model prevails and motivates to apply known host-centric models to the GPU. As we strive for a different approach to GPU cluster programming, kernels are no option.

Given the considerations provided, we chose a CUDA block as parallelization unit and communication target. Special care has to be taken to avoid communication deadlocks as CUDA blocks may not depend on each other and the scheduler doesn’t guarantee blocks progress concurrently. We present a solution to this implementation problem in Section 3.1.1.

**Library Conventions** Defining a CUDA block as a communication target introduces an interesting parallelization conflict. The block is seen as a single communicating unit from outside yet it entails a set threads executing in a data-parallel fashion with their own state and independent variable scope. When initiating communication, the two perspectives meet. CUDA allows to let some threads call a function while others don’t. Each thread has an individual scope and may pass different argument values. When passing arguments by reference, the data might have thread-scope, block-scope or device-scope. The library calling conventions need to specify what set of choices constitutes a valid library call to allow reasoning about program and library correctness.
This is a novel challenge we haven’t encountered in literature as for a standard CUDA program, the problem is less apparent. A function call is likely to be a mathematical function that applies to data of thread-local computations without block-wide effects. In GPU RMA, the block represent a single acting unit and the library interface calling conventions mediate the two perspectives.

We have chosen a CUDA block as a communication target as it provides a good encapsulation of self-contained computation. The threads are not fully independent but cooperate to perform a common computation with block-level progress. Communication is regarded relevant with respect to the block-level progress and we ask communication is performed by all threads in a CUDA block collectively. All threads in a block must be involved when calling a GPU RMA library function. We note that thread cooperation in library calls is important to enable coalesced memory accesses. On a side note, it is non-trivial to determine whether a subset of threads called a function or all threads were involved and individual threads cannot synchronize. The same reasons for choosing a block as communication target apply to library function call granularity. Exceptions will be discussed in the API specification in Section 2.3.3.

In the CUDA programming model, function calls are performed by each thread independently. While in GPU RMA all threads call the function collectively, they may pass individual arguments. Most library calls target block-wide state and individual thread state is no concern. Individual arguments become interesting when initiating communication and specifying transfer buffers. Are thread-specific transfer buffers meaningful? We have already established that the block is a single computing unit with block-level progress and its threads will have related data buffers exploiting data-parallelism. While each thread will want to communicate a different data element, they are likely to be consecutive in memory. Whether each thread specifies its own buffer or all threads request a consecutive memory range transfer collectively results in the same data transfer. The question of individual or equal function arguments is a cosmetic one. If the requested transfer buffers are not consecutive in memory, the call actually comprises multiple independent transfers implicitly encoded in varying thread-level arguments. In GPU RMA we ask the programmer to make multiple transfer requests explicit and require him to pass the same argument values in all threads. We argue from a programmer’s perspective the program intent is clearly expressed in code improving maintainability. From a library perspective we argue that detecting argument value variations is non-trivial and requires some sort of internal reduction introducing unnecessary overhead the programmer can avoid due to his problem knowledge. We see the same design choice by Sangman et. al. [13] in their adoption of the sockets API to GPUs blocks.

The last interface question addresses the memory-scope of the passed arguments. References may refer to thread-local registers, block-level shared memory, or device-level global memory. The question is relevant to provide a correct library implementation. Interpretation of input arguments is no concern as all threads provide the same values. The question is relevant in two cases: Parameter return values and transfer buffers. In a CPU program, a function may set an output value by writing the variable passed by a pointer. On the GPU, the result differs depending on whether the passed value has thread-scope or block-scope. A user program could loop on a library output value and only some threads may see the correct result if the passed variable has thread-scope but
the library assumed it to have block-scope and updates the value in only one thread followed by a memory fence. The same call with a block-scope variable would yield correct behavior. Defining the argument memory scope is a required part for the correctness of the GPU RMA API, or a block-level CUDA function in general. In GPU RMA, arguments shall have thread-level memory scope unless noted differently. Transfer buffers have to be accessible to the communication peers in some way and the programming model needs to capture this property. In GPU RMA we introduce the notion of managed memory (Section 2.3.2) that allows for the implementation to guarantee it is globally accessible. We note that the memory-scope specification of the library needs to consider the global access property.

Remote Memory Access In GPU RMA we strive to match the new hardware support of remote memory access with a one-sided communication model. We draw inspiration from the remote memory access model employed by MPI where ranks register memory regions with window objects to allow one-sided put and get transfers from remote ranks on these memory regions in accordance with different synchronization models (cf. Section 2.1). We refer to related literature [7] for more details.

Choosing one-sided communication introduces two drawbacks compared to two-sided communication. First, process synchronization is no longer a side-effect of communication but needs to be managed explicitly. Second, the tag-matching support for MPI point-to-point communication that allows to transfer metadata next to the actual data no longer exists. Belli and Hoefler [2] have shown that the resulting communication model is not well-suited towards single-element producer-consumer applications and introduce notified access (cf. Section 2.1) where a one-sided RMA operation generates a notification at the communication target. The notifications are used to synchronize the communication targets with fewer roundtrip latencies and convey information in a notification tag-value. We have chosen to adopt this approach in GPU RMA.

In their work, they present three different notification concepts. Overwriting notifications store a tag per communication peer at the target. However, a new notification from the same peer will overwrite the old value. Counting notifications accumulate the size or number of messages arrived and allow for message statistics. However, no user-submitted information such as a tag-value is supported. Matching Queue notifications are stored in a FIFO queue and allow for both, conveying information in tag-values as well as gathering message statistics. They reintroduce the matching power known from MPI point-to-point communication. We chose to implement the matching queue semantics in GPU RMA as it is the most powerful of the three approaches presented. For instance, the tag-value can be used to identify which part of a large memory region was updated in order to steer the next computations to perform. The API we implemented allows to match for specific tag values, specific RMA sources and check for a minimum number of such notifications generated.


2 GPU Cluster Programming

2.3 Specification

GPU RMA is a MPI-inspired programming model and library to program GPU Clusters. A program is executed in independent ranks. Ranks can communicate by reading and writing to memory windows of other ranks. Such a remote memory access triggers a notification on the remote rank. A rank can test for particular notifications. This allows synchronization amongst ranks. Ranks execute on GPUs only. The hosts may prepare initial input data for the ranks but are not involved in the computation and cannot be communicated with.

The GPU RMA programming model provides an abstraction for communication in GPU clusters as well as for manual GPU management. It doesn’t specify how to write particular GPU code. For instance, how a rank performs a computational task is left open in our model. In this work, we targeted Nvidia GPUs using the CUDA programming model. We will use CUDA terminology where appropriate to clarify GPU RMA behavior.

2.3.1 Terminology

Rank A rank is the main entity in GPU RMA. Each rank can execute independent code and communicate with other ranks. A rank is made up of multiple threads. The number of threads per rank is the same for all ranks and can be chosen at library initialization. The number of ranks executing in the system cannot be chosen freely as it depends on program properties. This is detailed in Library Initialization. A rank is implemented as a CUDA block. Any CUDA feature to describe a block’s execution may be used.

Communicator A communicator describes a set of ranks that may communicate amongst each other. Each rank in a communicator has a unique ID with respect to this communicator. In the library implementation, two predefined communicators exist. GPU.RMA_COMM_WORLD contains all ranks part of the execution. GPU.RMA_COMM_DEVICE is a device-local communicator and contains all ranks executing on that particular GPU. In our implementation, no further communicators can be created. This is a limitation of our implementation.

Window Windows are the basis for communication. They are assigned to communicators and created collectively by all ranks of the particular communicator. At window creation, every rank provides a local memory range that is registered with the window. The memory size can be different for every rank and may be zero. These memory ranges are exposed to RMA operations for other ranks part of the window.

RMA Operations (PUT, GET) Remote memory access operations allow to communicate data between ranks. A rank can either PUT data to another rank or GET data from

---

3^Note that we use the term window to refer to all memory ranges, known as a window object in MPI. In MPI, window refers to the memory range on a single rank only.
another rank. The rank initiating these operations is called the origin, the other rank is called the target. The RMA operation is performed on a window both ranks belong to. The operation is only valid if the memory range at the target is part of the range the target rank registered with the window. The memory range at the origin is not required to be registered with that window. Please refer to the memory model and API specification for the precise requirements.

**Notified Access and Notifications** Every RMA operation generates a notification at the target. Therefore, we also refer to an RMA operation as a notified access. The notification carries a tag specified by the origin. Notifications are stored in a matching queue at the target. The target may test for notifications, specifying the tag, source rank and notification count. The target may also wildcard these properties. Notifications are matched in FIFO order.

**Userdata** Userdata refers to the data initially provided to the ranks by the host when launching GPU RMA. It serves two roles. First, it can provide information about the problem being solved. Second, ranks can write results back to userdata that is available to the host after program completion. This is the communication method chosen in GPU RMA since the host is not included in the computation or communication. Please refer to the memory model for how to use userdata in a GPU RMA rank and to the API specification on how to manage userdata from host.

### 2.3.2 Memory Model

In GPU RMA, we distinguish between two types of memory: statically allocated rank-local memory and dynamically allocated managed memory.

**Rank-local Memory** Rank-local memory is private to a GPU RMA rank or private to a thread within a rank. Like GPU RMA doesn’t specify a model for computational tasks in a rank, GPU RMA doesn’t provide its own memory model for rank-local memory. For GPU RMA, rank-local memory is understood to be limited to thread-local memory or static shared memory of the CUDA programming model. These memory types are statically declared in CUDA code and the CUDA memory model applies (cf. Section 3.1.2). Rank-local memory can’t be exposed in a RMA window.

**Managed Memory** Managed memory is the only type of memory that may be exposed in a RMA window. It refers to either userdata or memory dynamically allocated by a GPU RMA rank using the malloc and free functions provided by the GPU RMA library.

---

4In our model and implementation, every RMA operation generates a notification at the target. Other models provide either RMA operations without notifications or both variants. In such a model, the operations are distinguished by calling them e.g. PUT and PUT-Notify. In our context, only notified access is available and the distinction not required.
Different rules apply depending on whether the memory is being used as-is or whether it is registered with a window.

- **Allocated** memory is private to the rank that allocated it. The CUDA memory model for global memory applies.

- **Userdata** memory may be written only if no other rank requires to read it at some point. In this case, the same rules as for *allocated* memory applies. No other rank must attempt to read that memory. If some other rank might potentially access that memory, the memory is read-only. It must not be written and it must not be exposed in a window. Note that these rules can be applied differently to different memory ranges of userdata.

- Regardless of whether managed memory originates from userdata or from the memory allocator, the following rules apply when the memory range is exposed in a *window*: Memory may change at any time due to RMA operations from other ranks. Based on the GPU memory model (cf. Section 3.1.2), reads from windows may be stale when fetched from cache. We guarantee memory reads from windows to be correct only if accessed with a pointer having the *volatile* qualifier or when using the provided wrapper function `readNoCache` that performs a cast to *volatile* internally. It is the programmers responsibility to employ proper synchronization to avoid race conditions.

### 2.3.3 Library Interface

The GPU RMA API is split into the host and the device API. The host is only involved in runtime initialization. After initialization, execution control is transferred to the ranks running on the GPUs. Our implementation assumes a uniform cluster architecture: Every compute node shall have the same number and type of GPUs installed.

**Device API Calling Convention** Unless noted differently, the following convention for calling the device API holds.

1. All threads in a rank must call an API function collectively. The function may synchronize the threads.

2. All threads must provide the same input arguments.

3. All threads see the same output values.

4. All arguments shall have thread-level memory scope.

---

5In fact, the host plays an important role realizing some of the functionality of GPU RMA, as the implementation section will show. From a programming model point of view, the host is passive while for the implementation the host is vital. Adding the host as a full member into the programming model while still allowing it to support the GPU for a working implementation is not straightforward and requires further work.
2.3 Specification

// Types

typedef struct gpurma_host_context* gpurma_host_context_handle;
typedef void (*gpurma_kernel_t)(gpurma_context_handle);

struct gpurma_rank_info {
    int rank_count;  ///< total number of ranks
    int rank_responsible;  ///< number of ranks I manage
    int rank_start;  ///< index of first rank I manage

    int device_count;  ///< number of devices on this compute node
    ///< NOTE: There can be more devices but not
    ///<     enough processes launched to
    ///<     manage them.
    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
};

// Functions

void gpurma_host_init(int argc, char *argv[],
    gpurma_host_context_handle *ctx,
    gpurma_kernel_t kernel,
    int threads_per_rank);

void gpurma_host_get_rank_info(gpurma_host_context_handle ctx,
    gpurma_rank_info *rank_info);

void gpurma_host_run(gpurma_host_context_handle ctx,
    void *gpu_data = NULL,
    size_t gpu_size = 0);

void gpurma_host_finish(gpurma_host_context_handle ctx);

Listing 2.1: GPU RMA Library Initialization: Host API
Library Initialization on the Host  In a GPU cluster, multiple hosts are involved managing the GPUs present. In GPU RMA, one host process is launched per GPU. The host shall prepare the required input data for the ranks running on its GPU and then pass control to the ranks. The API is listed in Listing 2.1.

Types  gpurma_host_context_handle is used to reference the initialized library. gpurma_kernel_t is the type of a valid CUDA kernel for GPU RMA. It is a kernel with no return value that takes as an argument a single gpurma_context_handle. This handle is distinct from the host handle and not relevant for the host. The gpurma_rank_info provides information about the runtime system such that the host may prepare the required input data for the ranks managed.

gpurma_host_init  This function initializes GPU RMA and must be the first API function called. The function must only be called once for the whole program lifetime. A handle to the library is returned in ctx. The programmer must provide the GPU RMA program to be run on the GPU and specify the desired number of CUDA threads per rank. The kernel and thread number provided must be the same on all host processes. The kernel and thread number requested will determine the number of ranks run on each GPU. This also depends on the particular GPU installed. If a different number of ranks shall be run in the system, the programmer shall either request a different number of threads per rank, modify the kernel so it yields different GPU resource usage, or vary the number of GPUs. We refer to the respective programming manuals for details on kernel resource usage [23, 24].

gpurma_host_get_rank_info  This function takes a pointer to gpurma_rank_info and fills in the corresponding information. The host may use this data to prepare the userdata.

gpurma_host_run  This function launches the CUDA kernel and passes execution control to the GPU ranks. The function call returns once all ranks have terminated. In the second and third parameter, the programmer may pass a pointer and size for the GPU RMA userdata. The pointer shall point to a memory range in host memory. The data is available to the ranks the host manages only. Refer to the device API and the userdata memory model on how to access the data. If a rank modifies userdata, the updates are visible on the specified host buffer once the function returns.

gpurma_host_finish  This function cleans up the GPU RMA library state. No GPU RMA function shall be called afterwards. The library shall not be initialized again by calling gpurma_host_init.

Library Initialization on the Device  The device initialization API is listed in Listing 2.2.

Types  The gpurma_context_handle is passed as the sole argument to a GPU RMA kernel (cf. gpurma_kernel_t). It is used to initialize the device-side part of GPU RMA. gpurma_state is used as a library handle after initialization.

gpurma_gpu_init, gpurma_gpu_finish  These functions match the library initializa-
2.3 Specification

// Types

typedef struct gpurma_context* gpurma_context_handle;
struct gpurma_state;

// Predefined Constants

gpurma_comm GPURMA_COMM_WORLD;
gpurma_comm GPURMA_COMM_DEVICE;

// Functions

__device__ void gpurma_gpu_init(gpurma_context_handle gpurma,
                                 gpurma_state& gst);
__device__ void gpurma_gpu_finish(gpurma_state& gst);
__device__ void gpurma_comm_size(gpurma_state& gst,
                                 gpurma_comm comm, int *size);
__device__ void gpurma_comm_rank(gpurma_state& gst,
                                 gpurma_comm comm, int *rank);
__device__ void gpurma_get_data(gpurma_state& gst,
                                void **gpu_data);
__device__ void gpurma_tid(gpurma_state& gst, int *tid);

Listing 2.2: GPU RMA Library Initialization: Device API
tion and cleanup from the host API. No GPU RMA function shall be called before init and no GPU RMA function shall be called after finish. The init function takes as first argument the handle passed to the CUDA kernel. It is used to initialize the library and the gpurma_state gst. gst must be declared as a CUDA shared variable __shared__ gpurma_state gst on the kernel stack. It must be passed as a library handle to any subsequent library calls.

**gpurma_comm_size, gpurma_comm_size** These functions provide the total number of ranks in a communicator as well as the unique rank ID of the current rank with respect to the communicator. The ID is guaranteed to be in the range [0, size-1]. The value is returned in the third argument. Contrary to the standard calling convention, these two functions may be called by individual threads and does not synchronize the threads.

**gpurma_get_data** This function returns in its second argument a pointer to the user-data provided by the host process. Note that while the contents of the buffer are equal, it resides in different memory. Pointers stored within the buffer by the host may not be valid on the device. The data may be different for ranks managed by a different host processes. The data is the same for all ranks managed by the same host process. Refer to the memory model on how to deal with userdata memory. If distinct buffers for the ranks are needed, a common pattern is to use the rank of the GPURMA_COMM_DEVICE communicator as an index into the userdata. Contrary to the standard calling convention, this functions may be called by individual threads and does not synchronize the threads.

**gpurma_tid** This function returns in its second argument the thread index of the calling thread. Contrary to the standard calling convention, this functions may be called by individual threads and does not synchronize the threads.

**Memory and Window Management** The API is shown in Listing 2.3.

**gpurma_mem_alloc, gpurma_mem_free** These functions work like the standard C malloc and free. The allocator returns one buffer address of size size that is the same for all threads.

**gpurma_win_create, gpurma_win_free** These functions create and free RMA windows. The second argument specifies the communicator the window is created in. The create and free functions must be called by all ranks in this communicator collectively. After the function returns, the window is guaranteed to be valid in all ranks. The provided window handle in the last argument is initialized to point to this window. If window creation fails, the window is set to GPURMA_WIN_NONE. The rank must provide a memory range to be registered with the window for this rank in the winbase and size arguments. size is specified in bytes and may be zero. The memory range must point to managed memory (cf. Memory Model). A window handle that was freed may be reused for a new window. The programmer may want to create overlapping windows. It is in the programmers responsibility to prevent potential race conditions.
// Predefined Constants

gpurma_win GPURMA_WIN_NONE;

// Functions

__device__ void* gpurma_mem_alloc(gpurma_state& gst, size_t size);

__device__ void gpurma_mem_free(gpurma_state& gst, void *ptr);

__device__ void gpurma_win_create(gpurma_state& gst, gpurma_comm comm, void *winbase, int size, gpurma_win *win);

__device__ void gpurma_win_free(gpurma_state& gst, gpurma_win win);

template <typename T> __device__ __forceinline__ T readNoCache(const volatile T* ptr)

Listing 2.3: GPU RMA Memory and Window Management API

readNoCache This function guarantees valid reads from a RMA window. Its use is encouraged but not mandatory (cf. Memory Model).

Notified Access The API is shown in Listing 2.4.

gpurma_put_notify, gpurma_get_notify These functions perform a notified PUT and GET operation respectively. The win and rank arguments identify the operation target and its memory range registered with the window. The offset and size specify the subrange of the registered memory range and are specified in bytes. The subrange may not exceed the registered range. The buffer is the source (PUT) or destination (GET) of the data transfer and must point to managed memory. It must be at least the size of the data transfer requested. The notified access operation adds a notification to the matching queue of the target. The tag argument specifies the tag value of this notification. The tag must not be negative.

gpurma_test_notifications, gpurma_wait_notifications These functions query the notification queue of the rank. The return value of the test function specifies whether the query was successful. The call is non-blocking. The wait function blocks until the query can be matched successfully, potentially waiting for new notifications to arrive. If the query was successful, the matched notifications are removed from the queue. The queue is not altered otherwise. The window argument specifies for which window the notification must have been generated. The source argument specifies the rank that generated the notification and may be the wildcard GPURMA_ANY_SOURCE. The tag argument specifies the tag value.
the notification shall have and may be the wildcard GPURMA_ANY_TAG. The count argument specifies how many of these notifications shall be matched in order for the query to succeed.

**Miscellaneous**  Listing 2.5 shows the remaining device API.

**gpurma_comm_barrier**  This function provides barrier synchronization amongst the ranks in the given communicator.

**Logging**  Standard CUDA does not provide logging support that is reported during kernel execution. We provide a rudimentary logging facility in the spirit of C++’s iostream. The gpurma_log object provides operator overloading for int, C-style char* strings and pointers void*. The gpurma_log object has a member flush. The log message is not displayed until the log object is explicitly flushed. Note that the standard GPU RMA calling convention apply. In particular, all threads must log the same values. The interface cannot be used for thread-level logging. The log message is printed to standard output by the managing host process. Note that in a cluster, messages printed by one host may arrive at another host later. In particular, a message printed on one host before a global barrier may arrive in a unified output log after a message printer from a different host after the global barrier.
2.4 Examples

We provide two examples of GPU RMA. The unit tests and performance benchmarks in the test/ directory of the GPU RMA source tree extend this introduction.

Hello World  A Hello World of GPU RMA is shown in Listing 2.6. In Lines 3 and 4, the respective host and device API headers are included. The code features the GPU program my_kernel (L. 6–22) as well as the initialization function main (L. 24–50) run on the host. On the host, we first initialize the library (L. 26–28). We specify the kernel with the GPU program we wish to run and the number of threads per rank. In this part, the library will inspect the kernel to determine how many ranks may be run per GPU as well as initialize part of the GPU state. We may then request this rank information (L. 30–43). In this hello world, we print this information to standard output. For a computational application, the host may want to load and prepare the required input data for its ranks. We now pass control to the GPU (L. 46). Once all ranks completed, the function will return and we can cleanup the library (L. 49). In a computational application, the host may want to save the results returned in the userdata. We will look at this in the next example. On the GPU, we start by initializing the device-side library and retrieve the communicator size and rank ID (L. 8–13). We use the logging facility to print this information (L. 15–18) and cleanup the library again (L. 21).

Rank Exchange  Rank Exchange is used as a basis for the GPU RMA validation tests and latency benchmarks. In these tests and benchmarks, we investigate the GPU RMA

Listing 2.5: Remaining GPU RMA API

```c
// Barrier Synchronization
__device__ void gpurma_comm_barrier(gpurma_state& gst,
    gpurma_comm comm);

// Logging
struct gpurma_log; // ‘<<’ operator overloading for char*, int, void*
__device__ gpurma_log gpurma_log_get(gpurma_state& gst);
__device__ gpurma_log& operator<<(gpurma_log& log, const char *str)
__device__ gpurma_log& operator<<(gpurma_log& log, const int i)
__device__ gpurma_log& operator<<(gpurma_log& log, const void *ptr)
__device__ gpurma_log& operator<<(gpurma_log& log, const gpurma_log_flush f)

// Assertions
__device__ void GPURMA_ASSERT(gpurma_state& gst, bool expr)
```
#include <cstdio>
#include "gpurma_host.h"
#include "gpurma_device.cu.h"

__global__ void my_kernel(gpurma_context_handle gpurma)
{
    // GPU RMA Initialization
    int size, rank;
    __shared__ gpurma_state gst;
    gpurma_gpu_init(gpurma, gst);
    gpurma_comm_size(gst, GPURMA_COMM_WORLD, &size);
    gpurma_comm_rank(gst, GPURMA_COMM_WORLD, &rank);

    // Say ‘Hello World!’
    gpurma_log log = gpurma_log_get(gst);
    log << "Hello World! ";
    log << "I am rank " << rank << "/" << size << "." << log.flush;

    // GPU RMA Cleanup
    gpurma_gpu_finish(gst);
}

int main(int argc, char *argv[])
{
    // GPU RMA Initialization
    gpurma_host_context_handle gpurma_host;
    gpurma_host_init(argc, argv, &gpurma_host, my_kernel, 256);

    // Print Rank Informations
    gpurma_rank_info rank_info;
    gpurma_host_get_rank_info(gpurma_host, &rank_info);
    printf("Hello!
"
        " - I manage ranks %d-%d out of %d ranks.
"
        " - I run device %d out of %d on node %d.
"
        " - Total node count: %d\n",
    rank_info.rank_start, rank_info.rank_start + rank_info.rank_responsible - 1,
    rank_info.rank_count,
    rank_info.device_index,
    rank_info.device_count,
    rank_info.node_index,
    rank_info.node_count);

    // Start GPU RMA Execution
    gpurma_host_run(gpurma_host, nullptr, 0);

    // Done
    gpurma_host_finish(gpurma_host);
}

Listing 2.6: Hello World in GPU RMA
#include "gpurma_host.h"
#include "gpurma_device.cu.h"

struct userdata
{
    int is_first_device;
    int is_first_peer_device;
    int is_first_peer_node;
    int has_peer_device;
    int has_peer_node;
};

// ... GPU Program goes here ...

int main(int argc, char *argv[])
{
    // GPU RMA Initialization
    gpurma_host_context_handle gpurma_host;
    gpurma_host_init(argc, argv, &gpurma_host, my_kernel, 512);

    // Rank Informations
    gpurma_rank_info rank_info;
    gpurma_host_get_rank_info(gpurma_host, &rank_info);

    userdata data;
    data.is_first_device = (rank_info.node_index == 0
        && rank_info.device_index == 0);
    data.is_first_peer_device = (rank_info.node_index == 0
        && rank_info.device_index == 1);
    data.is_first_peer_node = (rank_info.node_index == 1
        && rank_info.device_index == 0);
    data.has_peer_device = (rank_info.device_count > 1);
    data.has_peer_node = (rank_info.node_count > 1);

    // Run program
    gpurma_host_run(gpurma_host, &data, sizeof(data));

    // Cleanup
    gpurma_host_finish(gpurma_host);
}

Listing 2.7: Rank Exchange Host Code
Listing 2.8: Rank Exchange Device Code
behavior for transfers on the same device, to a peer device on the same node and to a device on a remote node. The first step is to exchange ranks from the different GPUs to establish the hardware topology. We use this example to demonstrate the use of userdata and RMA operations. The code for the host and device are shown in listings Listing 2.7 and Listing 2.8 respectively.

On the host, we first define a struct for our userdata (L. 4–12). We use it to inform the GPU ranks about their position in the hardware topology. The host code (L. 16–41) is similar to the hello world. We use the rank information to fill in the userdata (L. 22-34). When launching GPU execution (L. 37), we copy the userdata to the GPU.

On the GPU, we start by initializing the library as in the hello world example. Additionally, we get our device-local rank (L. 7) and retrieve the userdata the host supplied (L. 9–10). Only local rank 0 on each GPU will be involved. At the end of the exchange, rank 0 on GPU 0 should know the rank of the first rank on a peer GPU, if present, and the rank of the first rank on a remote-node GPU, if present. Those ranks will put their ID into a window of rank 0, so we first need to initialize this window. We allocate managed memory to hold two rank IDs (L. 13–15) and register it with a new window (L. 17–21). The rank exchange happens in L. 25–50. As mentioned, only one rank per GPU will take part in the exchange. Based on the userdata, the rank may decide if it will share its ID (L. 26, 41, 45). Rank 0 is the leader. It first determines the number of notifications to receive based on the topology and waits for them to arrive (L. 28–31). Rank 0 waits on the created config_win for notifications for any source GPURMA_ANY_SOURCE with tag 0. When the notifications have arrived, the rank reads the IDs from the RMA window into the local variable (L. 33–40). The peer device, if present, puts its rank at offset 0, the peer node at offset 4 (1 int). The remote ranks that share their ID do so in L. 41–49. Recall that a transfer buffer must reside in managed memory. They write their rank into the config_buf (L. 42, 46) which is managed memory. Notice that it’s not necessary that the source buffer is registered with a window, but it's convenient in this case. Using notified access, the ranks put their ID to the config_win of rank 0 (L. 43, 47) at different offsets. Finally, we free all resources (L. 53–55).
Implementation

This chapter presents our implementation of the GPU RMA communication library. We start by reviewing the important GPU technologies we relied on and follow with the explanation of our library design. We conclude with the deployment of GPU RMA applications and discuss library portability.

3.1 GPU Programming Fundamentals

GPU RMA leverages a number of existing GPU technologies for its implementation. We review fundamental GPU programming techniques including GPU scheduling and synchronization, the memory model for CUDA kernels, the GPUDirect technology family enabling fast inter-GPU communication over PCI-E, and shared memory solutions between a host process and a CUDA kernel.

3.1.1 Device-Level Scheduling and Synchronization

The CUDA programming model does not allow to control the device scheduler and provides no means for inter-block synchronization on the device. Hardware extensions have been proposed [42] but not adopted so far. We review the scheduling issue and introduce previous work on inter-block synchronization.

In CUDA, a kernel of multiple blocks is launched onto the GPU. The blocks are scheduled onto streaming processors (SMX). Depending on the number of streaming processors on the installed GPU a different number of blocks can run concurrently, allowing the programming model to scale across different hardware architectures. To ensure correctness, the blocks must execute independent and may not rely on other blocks
3 Implementation

being executed. In current Nvidia hardware, blocks are scheduled and executed until completion before further blocks are scheduled. The scheduling cannot be controlled. Typically, multiple blocks may execute in parallel in a streaming processor. The GPU computation is synchronized before and after all blocks execute at kernel launch time. Memory is only guaranteed to be coherent at these synchronization points.

In GPU RMA we want to communicate and synchronize between individual blocks during kernel execution. Coherent memory access is possible under certain conditions (cf. Section 3.1.2) yet the block scheduling raises a problem. Many known software synchronization approaches do not apply to the GPU as we might want to synchronize with a block that is scheduled to run after the current block is run to completion and we cannot control scheduling.

Xiao and Feng [46] have provided an in-depth study of the scheduling and barrier synchronization problem. They write a CUDA kernel such that its resource usage only allows a single block to run per streaming processor and run exactly as many blocks as streaming processors exist on the GPU. While this approach works, it causes severe underutilization of the GPU. The approach is no longer necessary as the current CUDA runtime allows to query how many blocks of a given kernel may execute in parallel on the installed GPU. We use this functionality to prevent the scheduling problem while maximizing GPU utilization. We use Xiao and Feng’s barrier synchronization technique in our work. Using the recent CUDA API we can query the maximal number of blocks that can execute concurrently and are not limited to a single block per streaming multiprocessor. We note that this synchronization technique requires manual scheduling from the application programmer. In the CUDA model, the user may freely choose the number of blocks matching his problem size. Using this block synchronization fixes the number of blocks. While the hardware still manages the scheduling of threads within the block, the application programmer must partition and schedule his problem across the fixed number of blocks that are guaranteed to be scheduled concurrently by the GPU.

3.1.2 Memory Model

The GPU execution model is known as Parallel Thread Execution (PTX). The PTX architecture describes a multi-level memory hierarchy to enable the massive parallelism of the GPU. We review the GPU memory model observed as a CUDA kernel covering the memory hierarchy and memory coherency guarantees. We conclude by summarizing programming consequences. This summary is based on the Nvidia CUDA and PTX manuals [24, 25, 28, 29].
3.1 GPU Programming Fundamentals

The GPU hardware has several streaming processors that execute CUDA blocks. Each streaming processor has a set of registers and an L1 cache. All streaming processors share an L2 cache and the device memory. The CUDA programming model knows thread-level, block-level and device-level memory. Thread-level memory is implemented in processor registers, block-level memory in the L1 cache, and device-level memory in the device memory.

PTX employs a weak memory consistency model. The block-level \_\_syncthreads() function performs a barrier synchronization amongst all threads in the blocks and ensures the visibility of all updates of memory accessed by the block. No such synchronization function exists for device-level synchronization. Instead, the \_\_threadfence() function acts as a memory barrier and prevents any read or write reordering before or after. Three versions of the fence exist guaranteeing the visibility amongst all threads in a block, all threads on the GPU, or all threads and further remote accesses over the PCI-E bus. From the description we assume the fences apply to the L1 cache, L2 cache and global memory respectively, though this is not mentioned explicitly. While the memory fence ensures correctness, we never experienced incorrect results when omitting it. Others have reported the same observation [46] but experienced a tremendous overhead when using the memory fences on Fermi-class GPUs. We did not observe this overhead on Kepler-class GPUs.

Caching behavior varies over the different Nvidia GPU architectures, the default behaviors of the compiler toolchain depending on the compilation target, and the source code. The variations apply to reads only. Writes are guaranteed to be written to coherency-level that is the device-wide L2 cache. Atomic operations are implemented in the L2 cache as well. While coherent, only a memory fence guarantees their global visibility. Reads may be cached in the L1 cache. The default behavior until Fermi-class GPUs was to cache reads in L1. Since Kepler and future architectures reads are no longer cached in L1 by default. This is a compile-time property. The PTX assembler may be instructed to issue caching reads by default. PTX inline assembly allows further variations. Finally, reads from variables with the volatile qualifier in C are guaranteed to be fetched from device memory and cache lines in L1 and L2 are invalidated.

The PTX ISA does not guarantee cache coherency and provides no means to flush or otherwise control the caches. A memory fence doesn’t flush the cache either. If a thread has read a value from global memory which was cached in L1 and is later updated by a thread from a different block, the original thread may still see the stale value in L1 even if a memory fence was issued. Only when the kernel finishes and a new kernel is launched on the GPU the CUDA programming model guarantees the results from the previous kernel are visible.

In summary, Nvidia provides a weak consistency memory model for its GPUs. Memory coherency between threads from the same CUDA block can be guaranteed and barrier synchronization functions exist. No such facility is provided for device-wide coherency. Memory fences guarantee data is written to the L2 cache or device memory. Reads may still return stale values if the memory was previously accessed and is cached in L1. Caching depends on many factors, including the GPU architecture, compiler and source code. Coherent reads are enforced by ignoring the caches and accessing device memory.
3 Implementation

Figure 3.1: Multi-GPU Communication Scenarios

directly using the volatile qualifier. Atomics are coherent on the device-level.

For more details we refer to the respective manuals. The CUDA C Programming Guide [24] provides details on memory fences (B.5), synchronization (B.6), atomics (B.12), the volatile qualifier (E.2.2.3) and compute capabilities with their default caching behavior (G). The Kepler Tuning Guide provides [29] provides further details on caches (1.4.4). The PTX ISA Manual [28] gives insight into the memory hierarchy (2.3), cache operators (8.7.8) as well as synchronization, memory fences and atomics (8.7.12.1–3). Refer to the compiler guide [25] on how to specify cache operators for the assembler.

3.1.3 GPUDirect and CUDA-aware MPI

In the last five years Nvidia gradually improved hardware support for inter-GPU communication. This section focuses on inter-GPU data flow from the GPU to the host or peer PCI-E devices. We cover programming aspects in Section 3.1.4.

Nvidia provides a family of technologies to accelerate GPU device memory access called GPUDirect [21]. It is important to note that GPUDirect refers to three related but distinct technologies. Unfortunately, most references to GPUDirect, including the official vendor information, do not make this distinction and use the term interchangeably to relate to any of these. Documentation is fragmented and the user recommended to use a third-party library such as MPI without further technological details. The same applies for scientific articles where it is helpful to look at the publication date to resolve what version of GPUDirect was available at that time.

When looking at data flow between GPUs the hardware topology is an important factor. Data travels the PCI-E bus, may cross several PCI-E switches, move to a different socket over QPI, or to a different node over InfiniBand. We have illustrated a possible topology in Figure 3.1. In the topology presented, communication between GPU1 and GPU2 involves the PCI-E bus only. GPU1 and GPU3 reside on a different socket and are attached to a different PCI-E root complex. Their communication involves QPI. To
communicate with node 2, GPU1 can use the IB Link 1. If the link is not present then GPU1 has to use IB Link 2 which involves both PCI-E busses, QPI and InfiniBand. Depending on the data path, a different GPUDirect technology applies. The Nvidia system management interface allows to query the hardware topology.2 We will now review the three GPUDirect technologies.

**GPUDirect Shared Access** introduced 2010 with CUDA 3.1, applies to inter-node communication. It allows the Nvidia driver and third-party drivers to share data buffers in page-locked host memory to avoid unnecessary buffer copies between them.

**GPUDirect Peer-2-Peer** introduced in 2011 with CUDA 4.0, applies to intra-node communication. It allows GPUs to access each others device memories in a CUDA kernel directly. The other GPU’s device memory is mapped into the current GPU’s virtual address space and allows for NUMA-style memory access without host interaction. GPUDirect P2P requires the GPUs to be on the same PCI-E root complex. The memory access uses a proprietary protocol and does not allow other PCI-E devices to access GPU device memory.

GPUDirect P2P also enables the on-device DMA engines to transfer data in parallel to CUDA kernel executions. The DMA engines can be accessed using the CUDA runtime from the host. Often in multi-GPU computations, a separate process is launched per GPU. Because GPUDirect P2P allows for faster data exchange between the GPUs, it is also known as CUDA IPC.3 The CUDA runtime also features a set of functions called CUDA IPC. They do not refer to GPUDirect P2P but are related. GPU memory pointers are valid in one host process. If GPUDirect P2P is used to move data to a GPU managed by a different process, the corresponding memory pointers need to be translated and registered to the other process’ CUDA runtime. CUDA IPC can either refer to the CUDA runtime functions that translate device memory pointers from one host process to another, or it could refer to the overall approach of combining these function with GPUDirect P2P to allow for fast inter-process-communication.

**GPUDirect RDMA (GDR)** introduced in 2013 with CUDA 5.0, applies to inter-node communication. GDR provides a kernel-level API exposed by the Nvidia driver that allows to map GPU device memory to the PCI-E bus so it can be accessed directly by peer PCI-E devices. The API can be used by third-party drivers to implement RDMA for GPU device memory.

In our work we used an InfiniBand cluster with network adaptors from Mellanox. Mellanox provides support for RDMA on GPUs that leverages GDR. In total, 3 kernel modules are involved. The Nvidia driver nvidia provides the GDR API. The Mellanox IB drivers mlx* manage the IB cards and have an interface to directly read and write to

---

1This worst case scenario was the configuration of our development machine at CSCS.
2\texttt{nvidia-smi topo -m}
3This is not an official name, but both MVAPICH2 and OpenMPI use it in their publications and documentations to refer to GPUDirect P2P.
peer PCI-E devices (Mellanox PeerDirect). Finally, the kernel module `nv_peer_mem` provided by Mellanox mediates between the Mellanox PeerDirect driver interface and the Nvidia GPUDirect RDMA driver interface to allow direct memory transfer between GPU and NIC.

GPU device memory is known to the Nvidia driver as frame buffer memory (FB). Memory is mapped to the PCI-E bus with BAR1 registers and is known to the Nvidia driver as BAR1 memory. These memories are not of equal size. For instance, on a Nvidia Tesla K80 GPU BAR1 is 16GB and FB is 12GB, allowing for all device memory to be mapped onto the PCI-E bus. On the Nvidia Tesla K20c GPU BAR1 is 256MB and FB is 5GB allowing only a fraction of GPU device memory to be exposed for direct access over PCI-E. FB and BAR1 sizes can be queried using the driver's system management interface.\(^4\)

**CUDA-aware MPI** A CUDA programmer doesn’t usually deal with the details of GPUDirect but uses a supporting library such as MPI. In fact, the *Mellanox GPUDirect RDMA Manual* [15] doesn’t provide any details on how to leverage RDMA but to use OpenMPI or MVAPICH2. A MPI implementation that accepts pointers to GPU device memory as transfer buffers is called CUDA-aware. As such, the programmer no longer needs to manually move data between the separated address spaces of the GPU and the host. A CUDA-aware MPI implementation may or may not use any of the GPUDirect technologies. Please refer to the respective MPI implementation’s documentation on which GPUDirect features are available and how to activate them. At the time of this writing, the four MPI implementations CRAY MPI, IBM Platform MPI, MVAPICH2 and OpenMPI are CUDA-aware [20].\(^5\) MVAPICH2 and OpenMPI both support GPUDirect RDMA on InfiniBand for point-to-point and one-sided operations.

As few people need to understand GPUDirect in detail, information is sparse and fragmented. The presented information is based on Nvidia product information, user manuals and developer blogs [14, 20, 21, 24, 26, 38], the Mellanox manuals [15, 16] as well as research publications on improving GPU communication and GPU–MPI integration [36, 37, 45].

### 3.1.4 Host and Device Memory Access

In this section we present the CUDA support to access host memory from the GPU and device memory from the host. A total of three approaches exist to access a shared memory region from both the GPU and the host. By default, the host and GPU access separate memories and manage their own virtual address space. We start with unified virtual addressing which addresses the virtual memory systems and proceed by the three approaches to share a common memory range between GPU and host. The presented

\(^4\) `nvidia-smi -q -d MEMORY`

\(^5\) Note that there are several versions of MVAPICH2. To our knowledge, only MVAPICH2-GDR supports GPUDirect RDMA and is available as binary distribution only. [http://mvapich.cse.ohio-state.edu/downloads](http://mvapich.cse.ohio-state.edu/downloads)
3.1 GPU Programming Fundamentals

Information is based on the CUDA manuals and gdrcopy documentation [23, 24, 26, 27].

**Unified Virtual Addressing (UVA)** was introduced with CUDA 4.0. Prior to UVA, the host and each GPU on the system had their own virtual address space. A memory address could refer to host memory, device memory on one GPU or device memory on another GPU depending on its execution context. With UVA, all GPUs and the host share a common virtual memory layout. That doesn't allow them to access each others memories, but uniquely identifies the underlying physical memory. The CUDA runtime allows to query what memory an address refers to. The addresses are not valid in a different host process unless explicitly translated (cf. GPUDirect P2P / CUDA IPC).

**Device Memory and the CUDA Runtime** GPU device memory is managed from the host using the CUDA runtime. Allocated memory is automatically mapped into the virtual address space of an executing CUDA kernel on the device. The host may not directly access the allocated memory on the device even if UVA is enabled. The host can access allocated device memory by calling the cudaMemcpy family of functions from the CUDA runtime. With CUDA 6.0, **Unified Memory** was introduced which provides coherent memory for both the GPU and the host. This is enabled by enforcing mutual exclusion when accessing a managed unified memory area and lazy migrating the memory pages between device and host memory. We didn't consider unified memory in our work.

**Pinned Host Memory** Pinned Host memory refers to host memory that is page-locked and mapped into the GPU's virtual address space. This allows the GPU to directly access host memory as if it were device-local and provides real shared memory access for the GPU and host in host memory. Because this access avoids the additional copy between the device and host memory when using standard cudaMemcpy functions, pinned host memory is also known as zero-copy memory.

**gdrcopy** With GPUDirect RDMA (GDR) GPU device memory can be mapped onto the PCI-E bus. While this is intended for direct memory access from peer PCI-E devices, the memory may also be mapped into the virtual address space of a process running on the host using an appropriate kernel module. Initial researchers wrote their own module. Meanwhile, Nvidia provides the gdrcopy library distributed separately from CUDA that allows low latency GPU device memory access from a host process by bypassing the CUDA runtime. gdrcopy consists of gdrdrv and gdrapi. gdrdrv is a kernel module that exposes the GPUDirect RDMA kernel interface as a Linux device in /dev/gdrdrv. gdrapi is a user space library to interact with /dev/gdrdrv.

Note that memory from gdrcopy is mapped at a different virtual memory address. First, GPU device memory is allocated using the CUDA runtime. Then, it is exposed to the PCI-E bus using gdrcopy and is manually mapped into the virtual address space using mmap. The device memory is then directly accessed at the mapped address. Yet still
3 Implementation

a CUDA kernel and the CUDA runtime refer to the memory by the original address obtained when allocating it. This introduces the complexity of operating on the same data under different virtual addresses on the host. The effort is rewarded with lower latency access to GPU device memory.

3.2 System Architecture

The GPU RMA library involves many parts. We start by discussing how to design inter-node communication and implement notified access over the InfiniBand network. We continue with the management of a single compute node and how interaction between the host and device is realized. Building on these fundamental parts we present the full notified access implementation control and data flow starting from the communication request on the device until the notification is available at the target rank. The design makes heavy use of queues for host and device interaction which we will cover in detail in its own section. We conclude with the organization of host and device memory and the partitioning of library state.

3.2.1 Network Communication

Inter-node communication amongst GPUs requires host interaction. Today’s technologies don’t offer support to control a network adaptor from a GPU kernel. A recent publication by Oden et al. [31] shows it can be done but their solution includes modifications to the CUDA runtime as well as the GPU and network drivers. While they succeed in bypassing the CPU, the performance they achieve is worse than that of a standard host-controlled approach. In our work, we use host-controlled communication.

We implemented GPU RMA for an InfiniBand cluster. For communication, we considered using either InfiniBand directly or via MPI. GPU RMA implements the novel notified access extension to MPI by Belli and Hoefler [2]. As these semantics are not yet part of MPI, it seems more promising to leverage InfiniBand directly using the IB Verbs API. As Belli and Hoefler mention, IB supports a write with immediate operation. The 32-bit immediate could be used to encode the notification tag. The idea doesn’t translate directly to GPU RMA. A connection endpoint represents multiple ranks running on the GPU, not a single rank on the host. Next to the actual RMA data, the transmission must include the source and target rank, tag value, and RMA window ID to allow for proper notification matching at the target. The information needs to be split into two transfers and cannot use solely the write with immediate operation. We note that such a solution is required anyhow as a corresponding read with immediate operation is not present in the IBV API.

Using IBV rather than MPI removes many convenience functionalities such as address resolving and connection management as well as the mpirun functionalities such as node configuration detection from the cluster schedulers, starting the binaries on the
relevant nodes and passing these informations to the program. As using IBV does not provide the semantics we are looking for but introduces a high engineering cost, we decided to use CUDA-aware MPI for our notified access implementation. The system latency evaluation in Section 4.1 will show the middleware is not the critical part of GPU RMA.

The authors of notified access could rely on hardware support for their notification implementation. The uGNI API of Cray networks allows to post a notification to the RDMA completion queue at the target. This functionality is not available with InfiniBand and we need to rely on a software mechanism to achieve the same goal. Implementing a notification queue using one-sided operations we face the problem of race conditions. One advantage of one-sided operations is that the origin fully specifies the transfer target buffer without target interaction. But how can we avoid a race condition on the target buffer if two remote nodes want to enqueue an element at the same time? A software protocol could be employed but will introduce additional roundtrip times. An atomic variable could arbitrate the requests. However, our system includes operations from the GPU, CPU and InfiniBand, interoperating over PCI Express. All of these four components provide their own atomicity support and memory consistency models. To the best of our knowledge, no unifying model exists. The race condition can be avoided by providing a queue per peer node. First of all, this approach yields linear memory requirements in the number of nodes. Second, the queue may be full and an unchecked write will overwrite unprocessed information at the target. Notified access using matching queues provides a powerful semantic extension to plain one-sided puts and gets. As the discussion has shown, an efficient implementation is hard to achieve without hardware support.

Implementing notified access using one-sided operations bears another problem. While MPI one-sided communication doesn’t need target interaction, accessing the RMA memory regions requires explicit synchronization. This is true for all three synchronization models MPI-3 RMA provides which is problematic for two reasons. First, the code and computation that uses the data runs on the GPU while MPI communication and synchronization is managed by the host. Adding additional arbitration between the host and the GPU for data access is expensive. Second, synchronization is achieved by global fences, explicitly allowing remote memory access for a certain time or acquiring a lock on the window. These mechanism do not match well with the aimed latency reduction through notified access. Fences introduce a rendez-vous of the communicating peers and explicitly allowing access requires the target to anticipate this. In notified access, we synchronize using notifications which don’t go along well with these approaches. Using a lock requires the communication target to acquire one as well, introducing additional roundtrip latencies. Given the synchronization requirements, using one-sided operations to implement single notified accesses including their notifications is no better choice than using two-sided message passing.

Given the challenge of implementing a one-sided notification queue and the synchronization of MPI-3 RMA, we decided to implement notified access using MPI message passing. Notice that while we’re using the message passing API, the underlying implementation still leverages GPUDirect RDMA for data transfer directly from the GPU where appropriate. We implement notified access using two messages. The first mes-
sage contains the notification metadata. The second message contains the data. The metadata is send from host to host, while the data is read and written directly to GPU device memory by means of GPUDirect RDMA thanks to CUDA-aware MPI. The notification is written into GPU memory by the target host when the data transfer has completed. Note that even using one-sided operations, only the host could have the guarantee that the data transfer has finished. Consequently, we have to involve the host in notification generation.

Figure 3.3 shows the sequence diagram of our notified access implementation. We will discuss it in Section 3.2.3 after introducing the device and host interaction on a compute node.

3.2.2 Device and Host Interaction

GPU RMA has independent ranks running on a GPU that rely on the host for memory management and communication. This section explains how communication between the host and device is managed while inter-GPU communication is handled by MPI. Our implementation requires one host process per GPU on each node. Figure 3.2 shows the system architecture for a GPU and its host process.

The main actor in GPU RMA is a CUDA block implementing a GPU RMA rank executing the user program. Each CUDA block is paired with a block manager running on the host. The CUDA block defers tasks it cannot perform on the GPU to the block manager. A total of four queues between the block and the block manager enables this communication. The block performs an upcall by enqueuing a command to the command request queue. The block manager will respond in the command reply queue. The
block receives notifications from notified accesses in the notification queue. Finally, the logging queue is used for debug output. This structure is replicated for every block on the GPU. Each block has local state that mainly concerns the communication queues. The blocks share a common state which tracks the registered RMA windows for fast on-device RMA as well as data for on-device barriers.

The block managers are not executing on their own but represent block state. Host execution is driven by the host event handler started by `gpurma_host_run` from the host API. Both the block manager and the host event handler interact with MPI for inter-node communication. Listing 3.1 shows the host event handler.

System execution starts by initializing the GPU, copying userdata and launching the kernel. A receive request for RMA metadata is posted. We will present the notified access implementation in more detail in Section 3.2.3. The host then enters the main event loop in lines 12–37. First, the host iterates over all block managers in lines 15–20 and polls the GPU queues. Then, it checks for inbound RMA requests in lines 22–34 and forwards them to the responsible block manager. The key elements are the calls to the block manager’s `process_commands()` in line 18 and `handle_rma()` in line 28.

In `process_commands()`, the block manager polls the block’s command request queue. Messages include memory management, window creation, RMA operations, barriers, and termination. The block manager executes the requests immediately and enqueues the response into the reply queue. Window creation, RMA operations and barriers may include inter-node communication using MPI (notice the arrow from the block manager to MPI in the architecture diagram). For such activities the block manager issues a non-blocking MPI call and adds the MPI request to its list of pending requests. Overall, `process_commands()` involves two activities. The block manager first checks its list of pending MPI requests to complete open requests from its GPU block. It then polls for new requests in the command request queue. All operations use non-blocking function calls only as a blocking manager would also block all other block managers and their ranks from making progress. The testing of pending MPI requests in each call to `process_commands()` ensures progress of the non-blocking communication functions.

Inbound RMA operations are polled by the host event handler. The handler checks for arriving metadata messages, identifies the responsible block manager and passes processing to the block manager by calling `handle_rma()`.

Once all ranks have terminated, the host event handler retrieves the resulting userdata from the GPU and makes it available on the host for final processing.

### 3.2.3 Notified Access

We have discussed network communication and explained how the device and host may interact. Based on these essential functionalities we now present our notified access implementation. We start with notified access control flow, discuss device-local and node-local variations, and conclude with notification handling.
3 Implementation

// Copy userdata to GPU

// Launch Kernel

// Post incoming RMA request
gpurma_internode_msg rma_msg;
MPI_Request rma_rq;
MPI_Irecv(&rma_msg, sizeof(rma_msg), MPI_UINT8_T, MPI_ANY_SOURCE,
    GPURMA_INTERNODE_TAG_RMA_NOTIF, MPI_COMM_WORLD, &rma_rq);

bool done = 0;
while (!done) {
    done = 1;
    // Process GPU Requests
    for (auto bman : ctx->block_managers) {
        bman->process_messages();
        bman->process_commands();
        done = done && bman->hasFinished();
    }
    // Check for RMA requests
    int rma_rq_valid;
    MPI_Test(&rma_rq, &rma_rq_valid, MPI_STATUS_IGNORE);
    while (rma_rq_valid) {
        // Notify block manager and post new request
        int block_id = rma_msg.target % ctx->blocks_per_device;
        ctx->block_managers[block_id]->handle_rma(rma_msg);
        MPI_Irecv(&rma_msg, sizeof(rma_msg), MPI_UINT8_T, MPI_ANY_SOURCE,
            GPURMA_INTERNODE_TAG_RMA_NOTIF, MPI_COMM_WORLD, &rma_rq);
        // More requests?
        MPI_Test(&rma_rq, &rma_rq_valid, MPI_STATUS_IGNORE);
    }
    // Assure no GPU failures (assertions, exceptions, misaligned reads, ...)
}

// Copy userdata from GPU

Listing 3.1: Host Event Handler
Figure 3.3: Notified Access Sequence Diagram
3 Implementation

Control Flow

The sequence diagram in Figure 3.3 shows the control flow of our notified access implementation. On the origin, the device side library and the block manager are involved. On the target, the host event handler and responsible block manager are involved. We also depicted the target device side library but it plays no active role.

Recall from Section 3.2.1 that notified access is implemented using two messages. The first metadata message contains source and target rank, the window reference, transfer type, offset and size, as well as the tag. It is transferred from host to host. The second message contains the actual data. While data transfer is initiated from the host, it is directly transferred from one GPU to the other. The host event handler always keeps a pending receive request for a metadata message that is first posted when execution starts and renewed after every inbound message, as seen in Listing 3.1. This is indicated as the init phase in the sequence diagram.

When a rank calls `gprma_put_notify`, the device side library puts a transfer request into the command queue. The block manager will start the transfer when activated by the host handler and issue the metadata and data messages. Once the metadata is sent, the host frees the allocated metadata buffer. No dynamic memory management is used on the fast path. The block manager holds a pool of metadata buffers that is automatically increased if empty. Note that the block manager does not wait for the transfer to be finished but returns control to the host event handler when finished. As mentioned in Section 3.2.2, the block manager maintains a list of pending MPI requests that is polled upon its next activation. On the target, the metadata messages is received when the host polls the pending request. The host then inspects the target rank of the message and forwards processing to the corresponding block manager. Once the block manager has finished, a new metadata receive request is posted. The block manager allocates a metadata buffer to keep track of the RMA state and posts the data receive request specifying the target buffer location on the GPU. When the window was created, both the GPU and the host were involved. The GPU keeps track of all the memory ranges assigned to ranks on the local GPU and the host stores a copy of the ranges as well as the relevant identifiers to reference a window on a remote node. Because of different communicators, not all nodes know all windows and different identifiers are used. The naming is resolved at window creation between the nodes. We do not discuss it further. Using the metadata message and the window information, the block manager can provide the correct device memory pointer to MPI when posting the data receive request and enables GPUDirect RDMA. Once the data transfer is finished, the target block manager enqueues the notification into the target rank's notification queue and cleans up its state. Note that the target device-side library is not involved in the transfer. Its timeline is shown to indicate that at the end, the notification is committed to the GPU device memory and the host is no longer needed. When the target rank tests for notifications, the matching queue handling is performed entirely on the GPU. Notification matching is discussed in Section 3.2.4.

The control flow for `GET` only differs in the data transfer. Rather than sending the data right after the metadata, the origin block manager posts a receive request. The target
block manager issues a send rather than posting a receive. The similarity is evident inspecting our code. Regardless whether a send or receive is performed, MPI returns a request object. The target block manager generates the final notification once the request is completed, may it represent the data is received from or fully transmitted to the origin. Our implementation involves the same code path for get apart from these small deviations.

We use different tags to indicate metadata and data messages. A metadata message is always matched and received by the host event handler and a data message is always matched and received by the block manager. Our host runs as one single-threaded process. When executing an RMA operation, the origin block manager issues both requests before returning control to the host. At the target, the host handler activates the responsible block manager when receiving the metadata message and the block manager issues the matching data request. Because of in-order message delivery in MPI we are guaranteed to match the correct message pair at the target.

**Device-local Transfers**

The control flow is different for device-local transfers between ranks on the same GPU. For on-device transfers the rank copies the data directly to the target location of the peer rank in global memory. The device still enqueues a RMA request into the command queue as the block manager generates the accompanying notification.

Ranks on the same device share global memory. When optimizing applications for GPU processing, data copies can sometimes be avoided by reading data directly from another rank’s memory. The GPU RMA programming model provides no such mechanism and is weaker than a hand-tuned implementation. To accommodate for this, we provide the no-copy optimization. If the source and destination address for a on-device transfer are the same, the copy is omitted. The user may now provide the input in userdata with an optimized memory layout and deliberately register overlapping windows from this memory. When performing notified accesses between ranks where the actual source and destination is the same no copies are performed and no overhead compared to a hand-tuned implementation occurs except for the notification that is generated by the host. Note that even without GPU RMA some sort of synchronization is needed. From a programming model point of view the user still uses notified access on windows and does not violate the model. The optimization is completely transparent to the user and does not alter the GPU RMA semantics. It applies only if the user deliberately choses to arrange memory so it can be applied. Note that we do not allow userdata memory to be used as shared memory for communication amongst ranks on the same device (cf. Memory Model, Section 2.3). The no-copy optimization can be disabled at compile time.

Device-local transfers are performed directly in the block that copies the data. This is fast for small memory transfers. However, memory bandwidth from a single CUDA block is limited and this approach prohibits computation and communication overlap. Alternatively, the on-device DMA engine could be used at full memory speed. This requires that the host sets up the data transfer yet the host is already involved generating
3 Implementation

the notification and the additional overhead is limited. This allows the computation and communication to execute in parallel. While this might improve overall performance, we note they share the same aggregate memory bandwidth. For our experiments, latency dominates and bandwidth is not relevant. An advanced implementation would allow to use both protocols to enable performance tuning.

Node-local Transfers

We use the same MPI protocol for data transfers to a peer GPU in the same node as for inter-node communication. This is not ideal for two reasons. First, all GPUs can be controlled by a single process which avoids context switches and MPI invocations on the fast path. Even worse, passing device pointers from one process to another requires explicit address space translation for the pointer to be valid and accepted by the target process and the CUDA runtime. While CUDA provides the API, it introduces further latency. Second, memory of a peer GPU device can be mapped into the virtual address space of the initial GPU. Peer memory access is activated once and valid for all global memories of the GPUs. This allows to completely bypass the host and CUDA runtime for memory transfers to implement faster small messages transfers as we did for device-local transfers. As our GPU RMA performance modelling in Section 4.2 will show we already achieve a 12.4 GB/s bandwidth for peer transfers. The DMA transfer initiated by the MPI implementation makes good use of the PCI-E 3 x16 link between the peer GPUs in our test system. By applying the mentioned optimizations we expect the notified access latency to improve.

Notification Handling

A notified access generates a notification at the communication target. The target rank may query for notifications which enables rank synchronization. Our implementation employs a single notification queue per rank where notifications are enqueued by the host. Involving the host is reasonable as inter-node communication uses MPI and only the host may check for completed transfers. We use this approach even for transfers where both the origin and target are on the same GPU and MPI is not used. The choice is motivated by two reasons.

The host and GPU communicate over PCI-E. Allowing both of them to write into the notification queue will cause race conditions if not mitigated by a mutual exclusion or a lock-free protocol. Such a solution introduces additional PCI-E traffic. As we will describe in Section 3.2.4 we carefully tuned our queue implementation to minimize PCI-E traffic on the critical path of notified access. Allowing multiple queue writers will degrade queue performance.

An alternative approach is to use additional queues for device-local notifications. We did not implement the solution as it adds costly global memory accesses (cf. Section 3.2.4). In particular, the latency analysis in Section 4.1 shows a queue ping-pong between the device and host is faster than between two GPU blocks. While an on-device queue would
require just an enqueue operation from the origin to the target rather than a roundtrip over the host, the target rank will need to poll an additional queue for notification matching which incurs additional global memory latency.

### 3.2.4 Queue Design and Notification Matching

Queues are a key building block in GPU RMA as they enable communication between device and host for memory management, window setup, and notified access. Particular effort was spent to optimize the queue as its performance is critical for initiating transfers and writing notifications. In this section we present the queue design motivated by the system configuration. We start with the shared memory access in our CPU–GPU system, proceed to queue operations, and end with notification matching.

#### Shared Memory Access

In Section 3.1.4 we introduced three approaches for shared memory access between the device and host based either on host or device memory. We can page-lock host memory and map it into the GPU’s virtual address space so both the host and device may access it. Alternatively, we can allocate memory on the device using the CUDA runtime which automatically maps it into the GPU’s virtual address space. For the host, two options for accessing device memory exist. The CUDA approach calls the `cudaMemcpy` functions from the CUDA runtime as memory is not directly accessible. Since GPUDirect RDMA is available, a custom kernel module such as `gdrcopy` can be used to map device memory into the host’s virtual address space. Device memory can now be accessed as any other memory location and the CUDA runtime is bypassed. However, the memory is mapped at a different virtual address on the host still CUDA functions except pointers with respect to the device’s virtual address space. In all three approaches, data is transferred over the PCI-E bus between the GPU and the host. Our queue implementation supports all three shared memory approaches.

Memory access time is the limiting factor of queue performance. To minimize PCI-E accesses, we chose to always locate the queue buffers in the memory closest to the recipient. The device to host command request queue uses host memory buffers, the command reply queue device memory buffers. In the optimal case, only a single data transfer with a new queue entry traverses the PCI-E bus and the recipient polls the queue in its local memory. To access device memory the host uses `gdrcopy`. We evaluated all nine design variations in a ping-pong benchmark to confirm this combination performs best. In Section 4.1 we present the latency numbers of our chosen design. The other variations are not reported because they perform worse and find no application in GPU RMA. All results can be obtained by running the `flagged_queue_latency_test` which performs a ping-pong benchmark using a ping and a pong queue. All nine combinations using either pinned host memory (H) or device memory accessed using either the CUDA runtime (D) or `gdrcopy` (G) for the ping and pong queues respectively are reported.
3 Implementation

notification we use. The queue implementation uses C++ templates and can adapt for future uses. On the GPU, we access data at warp granularity which yields coalesced memory accesses.

Queue Operations

The queue is implemented as a ring buffer. For the device, the same implementation can be used for all three shared memory approaches. For the host, the implementation varies. A straightforward ring buffer queue uses head and tail pointers to represent queue state. The pointers are read and updated on enqueue and dequeue operations. In our setup, this would generate additional PCI-E traffic and latency for any access by either the GPU or host. We decided to omit the head and tail pointers and indicate valid queue entries by a sequence number embedded in the data. The target inspects the sequence number of the next entry and compares it to the expected value to decide if the entry is valid. A successful dequeue comprises a single cache line read for validation and data access only. For enqueue we introduced a problem. Having removed the shared tail pointer the origin cannot know whether the queue is full. The target could mark consumed queue entries but the origin would need to inspect these fields yielding additional PCI-E accesses. We solved the issue using a lazy-updated free counter at the origin. The counter is decreased with every enqueue operation, but only the data transferred over the interconnect. The origin maintains a private head pointer, the target a public tail pointer. Whenever the target dequeues an element, it also updates the public tail. When the origin’s free counter reaches zero, it fetches the public tail from the target. By comparing the two pointers, the number of free entries in the queue can be computed and the free counter updated. Overall, a single PCI-E transfer is required next to the enqueued data whenever the free counter is updated. With increasing queue size lazy updates become less frequent. In total, the ring buffer and the public tail need to be located in shared memory.

Listing 3.2 shows the resulting device implementation. The enqueue operation first checks the free counter (L. 4–10). If the counter reached zero, __fqueue_update__-entries performs the lazy update by fetching the public tail and comparing it to the local head. If there is space, we add the sequence number to the payload (L. 15) and update the local queue state (L. 12–19). A warp then copies the data into the queue buffer (L. 21–25). If the queue is full, we update the free counter so the update may be avoided at the next enqueue operation. In this code, three operations may cause PCI-E traffic. The free counter update (L. 6, 28) performs a single read and is only executed if necessary. Writing the data entry into the queue buffer (L. 24) is the single compulsory data transfer performed by the enqueue operation. The dequeue operation first reads the next data element (L. 36–41) and inspects the sequence number (L. 43–45). If the entry is valid, we update the queue state and public tail (L. 48–53). As we chose to locate the buffers and public tail in the target memory, none of these operations causes PCI-E traffic.

---

7 On the device, the memory is always mapped into the kernel virtual address space. The host must distinguish between normal memory access (pinned host memory, H), maintaining device and host pointers (gdrcopy, G) or using the CUDA runtime (standard, D).
3.2 System Architecture

```cpp
template<typename T> __forceinline__ __device__
queue_error_t fqueue_device_enqueue(fqueue_t<T> *q, fqueue_t_data<T>& val)
{
    // see if next element is free
    if (q->free_entries == 0) {
        __fqueue_update_entries(q);
        if (q->free_entries == 0) {
            return QUEUE_FULL;
        }
    }
    // fill in metadata and update queue state
    volatile fqueue_t_data<T>* head_write = q->head;
    BEGIN_SINGLE_THREADED {
        val.data.id = q->head_id_next;
        q->head_id_next += 1;
        q->head = __fqueue_it_next(q->head, q);
        q->free_entries -= 1;
    } END_SINGLE_THREADED
    // write data
    volatile uint32_t *dst = (volatile uint32_t*)head_write;
    uint32_t *src = (uint32_t*)&val;
    __syncthreads();
    if (threadIdx.x < 16) dst[threadIdx.x] = src[threadIdx.x];
    __syncthreads();
    // if needed, update free entries now to be ready for next time.
    if (q->free_entries == 0) __fqueue_update_entries(q);
    return QUEUE_OK;
}

template<typename T> __forceinline__ __device__
queue_error_t fqueue_device_dequeue(fqueue_t<T> *q, fqueue_t_data<T>& val)
{
    // load entry
    volatile uint32_t *src = (volatile uint32_t*)q->tail;
    uint32_t *dst = (uint32_t*)&val;
    __syncthreads();
    if (threadIdx.x < 16) dst[threadIdx.x] = src[threadIdx.x];
    __syncthreads();
    if (val.data.id != q->tail_id_next) {
        return QUEUE_EMPTY;
    }
    // update queue state
    BEGIN_SINGLE_THREADED {
        q->tail_id_next += 1;
        q->tail = __fqueue_it_next(q->tail, q);
        // update public tail
        *(q->public_tail) = q->tail;
    } END_SINGLE_THREADED
    return QUEUE_OK;
}
```

Listing 3.2: Device-side Queue Implementation
3 Implementation

Finally, we also optimized the device memory accesses. Global memory access is slow on the GPU compared to CUDA shared memory [28, Sec. 6.6]. In our queue implementation, only the ring buffer and the public tail must reside in global memory so it can be mapped onto the PCI-E bus. The remaining queue state is only relevant for the block and can reside in CUDA shared memory. Note that CUDA shared memory is a fast block-scope memory region on the GPU and unrelated to the shared memory for the queue buffers and the public tail we discussed so far. The fqueue_t struct in Listing 3.2 represents this queue state. In enqueue, no memory operation targets global memory but operates in cache only. For dequeue, reading the queue buffer in line 40 and updating the public tail in line 52 accesses global memory. In GPU RMA, all buffers for elements we enqueue or dequeue, provided by the val argument, are declared as CUDA shared memory. Updating and inspecting their sequence number (L. 15, 43) is a cache operation as well. GPU RMA memory organization is discussed in detail in Section 3.2.5.

Notification Matching

We implemented notification matching as a specialized three-step dequeue operation. In this section we describe the matching operation, analyze the memory accesses, and present possible optimizations.

First, the queue is traversed and each notification inspected until the requested number of notifications are matched or the end of the queue is reached. During traversal, we keep a pointer to every matching notification in a local array. Every notification has a field that indicates whether it has already been consumed. We don’t touch this field in the first step. If the requested number of notifications are matched, we mark them as consumed in step two. Finally we clean the queue. To reduce global memory operations we never move elements in the queue but remove matched notifications in a FIFO fashion. That is, we remove all consumed notifications from the top of the queue but keep notifications with an unconsumed predecessor. The matching ends with updating the public tail of the queue. On the fly compression would be a design alternative.

The first step performs a linear traversal of the queue to find the matching notifications. The traversal operates on the queue buffer and therefore on global memory. The second step marks matched notifications as consumed. Because we stored pointers to the matched notifications in step one, we can directly update the entries and don’t need to perform a linear traversal again. The final queue cleanup step traverses the queue from the start until it reaches an unconsumed notification. Then it updates the tail. All these operations target global memory.

Performing an initial linear traversal is unavoidable. To prevent additional traversals for future matchings, unconsumed notifications could be moved into CUDA shared memory. An application could benefit from this optimization if it often matches notifications in a different order than they arrive, but only moderately. If the application causes the second buffer to be always full, the advantage vanishes. Also, CUDA shared memory is a scarce resource. Marking the notifications in step two is cheaper than deleting them.
and moving preceding notifications. In the cleanup step, traversing the queue until an unconsumed notification is found can be avoided. The operation could already track the potential future tail in step one so only the public tail update in the end is needed. This could also avoid marking notifications as consumed that won’t be in the queue anymore after the final step and further decrease overall global memory access.

We have written an exhaustive unit test to validate the matching operation available in the GPU RMA source.

### 3.2.5 Memory Organization

We identify two types of GPU device memory that needs to be accessible from the outside. The first is used for library-internal communication between the device and the host. The second type serves as source or destination for notified access, including userdata or dynamically allocated memory possibly registered in windows. This memory type has to be accessible by the underlying communication library that is MPI. The GPU RMA specification captures this property from a programming model point of view by defining *managed memory* as the only memory type valid for transfer buffers (Section 2.3.2, Page 15).

We use two memory allocators for the two types of device memory and allocate a memory pool for each at library initialization. The *GDRAllocator* provides memory for library-internal use. It leverages *gdrcopy* (cf. Section 3.1.4) to map GPU device memory directly into the virtual address space of the host process allowing low-latency memory access bypassing the CUDA runtime. The *gpuwin* allocator provides managed memory used by all data transfers. It ensures the memory pool is registered with MPI and the InfiniBand driver for fast remote memory access.

The library state is partitioned amongst the different actors on each node seen in Figure 3.2. The host keeps track of peer nodes, GPU state, and global windows in *gpurma_host_context*. The block managers encapsulate pending commands and in-flight MPI operations of their block as part of their class implementation *gpurma_block_manager_host*. On the GPU, the state is threefold. The *gpurma_context* is passed as a kernel argument and contains all initial information the GPU needs. The data resides in global memory and is used at runtime as shared GPU state to store window information and implement device barriers. The second part is *gpurma_block_data* which comes with the initial context and holds per-block data such as references to the communication queues of its block manager. The queues are a key to overall system performance as they enable notified access. At kernel launch, all GPU state resides in global memory which is comparably slow [28, Sec. 6.6]. Queue buffers need to reside in global memory for the host to access them, but queue state is block-local. To improve memory access times, we relocate all state that is read and manipulated by one block only into CUDA shared memory. The storage is declared as *gpurma_state* by the user and passed to the device-side library initialization *gpurma_gpu_init* (cf. Page 18).
3 Implementation

3.3 Deployment and Portability

GPU RMA Ecosystem  GPU RMA comes preconfigured with the cmake build system and includes build targets for the library, unit tests, performance benchmarks, and horizontal diffusion application. We have written validation tests for the key GPU RMA components including the queue, barriers, notification matching, RMA operations, user-data handling, and logging facilities. They are provided together with the performance benchmarks in the test/ directory. We recommend looking at these programs as a starting point for own applications. A custom application can be added to the system and requires but to specify the sources and link against g purma in CMakeLists.txt.

<table>
<thead>
<tr>
<th>CUDA_ADD_EXECUTABLE</th>
<th>(my_app apps/my_app/src1.cu apps/my_app/src2.cu)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TARGET_LINK_LIBRARIES</td>
<td>(my_app gpurma)</td>
</tr>
</tbody>
</table>

Portability  We built and tested GPU RMA on an InfiniBand cluster with Haswell nodes and Nvidia Tesla K80 GPUs using CUDA 7.0 and OpenMPI 1.8.5 [33]. The choice of the MPI implementation and GPU might require adjustment of GPU RMA parameters.

Networking is handled completely by MPI and we expect GPU RMA to work on different networks as long as a CUDA-aware MPI implementation supports it (cf. Section 3.1.3). A CUDA-aware MPI implementation may still cause deadlocks if not implemented carefully as we experienced with OpenMPI's default configuration. For large data transfer, OpenMPI stages data buffers through host memory rather than using GPUDirect RDMA because it yields higher bandwidth. The default configuration uses synchronous copies which are serialized by the CUDA runtime to run after kernel execution. In GPU RMA, a single CUDA kernel executes and communicates with the host to initiate transfers concurrently, resulting in a deadlock. There is a configuration parameter that enables asynchronous copies that run concurrent to CUDA kernel execution. Due to a bug in OpenMPI 1.8.5, the parameter is not listed amongst the runtime parameters and we only learned this by asking on the mailing list.8

When choosing a different GPU, not all memory might be available for GPUDirect RDMA. Recall from Section 3.1.3 that the GPU virtual memory system distinguishes frame buffer memory (FB) and BAR1 memory. Frame buffer memory refers to the GPU device memory that is available as global memory to CUDA kernels. BAR1 is used to map GPU device memory to the PCI-E bus for peer-to-peer communication and GPUDirect RDMA. On the Tesla K80, FB size is 12GB and BAR1 size is 16GB, so all device memory can be exposed for remote memory access. On the other hand, the Tesla K20c has 5GB FB memory but only 256MB BAR1, so only a fraction of the available device memory may be remotely accessed. GPU RMA uses two allocators to manage device memory, both of which rely on it being mapped in BAR1 (cf. Section 3.2.5). The memory pools are statically allocated at library initialization. To support GPUs with smaller

8The developers confirmed there is no reason why asynchronous copies should not be used. They promised asynchronous copies will be made the default configuration as of OpenMPI 1.10. The bug in the configuration system preventing the parameter to be listed is fixed in the upcoming OpenMPI 2.0.0.
3.3 Deployment and Portability

BAR1 size, the memory pool sizes `GPURMA_GPUWIN_SIZE` and `GDR_ALLOC_POOL_SIZE` can be adjusted in `include/gpurma_constants.h`.

**Build Configuration**  The build process is managed by `cmake`. To enable compiler optimizations the `CMAKE_BUILD_TYPE` has to be set to `Release`. Note that applications yield different register usage if optimization is enabled. In our experience, default debug builds allow to run more blocks concurrently on a GPU but yield worse performance. The no-copy optimization can be disabled by setting the `WITHOUT_NOCOPY` property.

For debugging purposes, detailed logging of block manager activities can be activated by passing the `BMAN_LOGGING` macro to the compiler. Logging is also enabled by uncommenting the definition in `src/gpurma_block_manager_host.cu`.

**Running and Tuning**  A GPU RMA application is run like a regular MPI program. A MPI rank should be scheduled per node for every GPU to be used. For OpenMPI, a series of runtime configuration parameters exist which are explained in [34]. Five of them are important in the context of GPU RMA.

- `btl_openib_have_cuda_gdr` reports if OpenMPI was built with GPUDirect RDMA support.
- `btl_openib_have_driver_gdr` reports if the necessary drivers for GPUDirect RDMA are installed and loaded on the current system.
- `btl_openib_want_cuda_gdr` tells OpenMPI if it should enable GPUDirect RDMA. Note that in OpenMPI 1.8.5 GPUDirect RDMA is disabled by default even if support was compiled in and the drivers are loaded.
- `btl_openib_cuda_rdma_limit` defines at what message size OpenMPI switches from using GPUDirect RDMA to staging buffers through host memory. Staging through host yields higher latency but better bandwidth. Benchmark your system to find the ideal protocol switch message size.
- `mpi_common_cuda_buffer_copy_async` instructs OpenMPI to use non-blocking asynchronous copies when staging buffers through host. The property is disabled by default but will result in a deadlock when not enabled as explained in the portability section.

GPU RMA default implementation parameters such as queue sizes and memory pools can be adjusted in `include/gpurma_constants.h`.

We give a more complex example on how to run a GPU RMA using OpenMPI based on the topology encountered at the Nvidia PSG cluster. In this particular setup, we want a total of four GPUs on two nodes. The node is a dual socket machine that has two GPUs on the first socket and six GPUs and the NIC on the second socket. Listing 3.3 shows the output querying the node topology from the Nvidia tools. We want to use GPUs on the same socket as the NIC only and all host processes should be scheduled on that socket as well. In line 1 we set the environment variable that tells the CUDA runtime to limit itself to the GPUs 2–7 on the second socket. We export this environment variable to all
3 Implementation

```bash
$ nvidia-smi topo -m
GPU0 GPU1 GPU2 GPU3 GPU4 GPU5 GPU6 GPU7 mlx5_0 CPU Affinity
GPU0 X PIX SOC SOC SOC SOC SOC SOC 0-15
GPU1 PIX X SOC SOC SOC SOC SOC SOC 0-15
GPU2 SOC SOC X PIX PHB PHB PHB PHB PHB 16-31
GPU3 SOC SOC PIX X PHB PHB PHB PHB PHB 16-31
GPU4 SOC SOC PHB PHB X PIX PXB PXB PXB PHB 16-31
GPU5 SOC SOC PHB PHB PIX X PXB PXB PXB PHB 16-31
GPU6 SOC SOC PHB PHB PXB PXB X PIX PHB 16-31
GPU7 SOC SOC PHB PHB PXB PXB PIX X PHB 16-31
mlx5_0 SOC SOC PHB PHB PHB PHB PHB PHB X

Legend:
X = Self
SOC = Path traverses a socket-level link (e.g. QPI)
PHB = Path traverses a PCIe host bridge
PXB = Path traverses multiple PCIe internal switches
PIX = Path traverses a PCIe internal switch
```

Listing 3.3: PSG Node Topology

nodes in line 3. In line 4 we instruct MPI to schedule processes on the CPUs 16–31 on
the second socket only. We request asynchronous CUDA copies between the host and
GPU in line 5. In lines 6–7 we enable GPUDirect RDMA and set to message size for a
protocol switch from GPUDirect RDMA to staging through host memory to 5KiB. Finally,
we specify that a total of four ranks should be launched and two processes scheduled
per node.

```
$ export CUDA_VISIBLE_DEVICES="2,3,4,5,6,7"
```
Evaluation

Our evaluation of GPU RMA is threefold. We measured all components contributing to the notified access latency individually to understand the limiting factors of our library implementation. We used microbenchmarks to test the library performance in isolation and derived a GPU RMA performance model that enables performance modelling of custom applications. As a case study, we implemented horizontal diffusion using GPU RMA. The case study serves two purposes. First, we validate our GPU RMA performance model against actual application performance. Second, we compare the GPU RMA results to a state of the art implementation using MPI and CUDA to assess the performance of our novel GPU cluster programming model.

4.1 System Latency Analysis

We measured all components contributing to notified access latency to understand the influences of our implementation choices and bottlenecks of the GPU communication infrastructure. We introduce our test system and report the latencies measured.

We evaluated GPU RMA on two compute nodes of the greina cluster at the Swiss National Supercomputing Center CSCS [4]. Each node is a dual-socket machine with two Intel Xeon E5-2680 v3 CPUs (Haswell) that run Scientific Linux 6.6. The nodes are connected over 4x EDR InfiniBand with a Mellanox MT27620 (ConnectX-4) network adapter using the OFED 3.0-1.0.1 software stack. Each node contains a Nvidia Tesla K80 GPU connected over PCI-E 3 x16. The Tesla K80 GPU provides two independent GPUs on a single PCI-E slot. A total of four logical GPUs are available, two per node. The Tesla GPUs provide dynamic frequency scaling called Autoboost. The feature was disabled and the application and memory clocks fixed to the maximum values of 875MHz and 2505MHz respectively. Due to the system hardware, the InfiniBand adapter and the
GPU are connected on different sockets and reside on different PCI-E root complexes. Any communication between the network and the GPU has to pass the QPI interconnect between the two sockets on the compute node. We built our system using GCC 4.8.4 and the CUDA Toolkit 7.0. The Nvidia Driver installed was version 352.39. For MPI we used OpenMPI 1.8.5 which is CUDA-aware. We compiled the host code with the -O3 -DNDEBUG flags and the device code for compute capability 3.7. Figure 4.1 shows the system configuration.

Table 4.1 shows a summary of measured system latencies. If not indicated differently, the time reported is the measured time for a ping-pong roundtrip. We used custom microbenchmarks for all measurements except for the networking section. The Memory Access and Queue results stem from the flagged_queue_latency_test, Notified Access from the na_latency_test and GPU RMA from the rma_latency_test available in the GPU RMA source tree. The reported numbers are the mean of at least 1000 iterations and the 99% confidence interval lies within 1% of the reported mean. The Intra-Node GPU Copy results stem from the p2pBandwidthLatencyTest provided with the CUDA 7.0 toolkit and the inter-node MPI results from the OSU Micro-Benchmarks 4.4.1 [19]. We report the numbers as presented by these third-party benchmarks but doubled the value to match the roundtrip numbers from our custom measurements. As far as we know both benchmarks report the mean of 10000 iterations.

**Memory Access** We start with the memory access latencies. Two actors perform a ping-pong test. The first writes a ping value at a given memory location. The second actor polls the memory location and as soon as he sees the ping value he writes the pong value. The first actor measures the time from writing the ping value until he sees the pong value. We report the numbers for actors on either the GPU or the host where the flag value is either in host or device memory. The system diagram in Figure 4.1 clarifies the measurement configuration.
4.1 System Latency Analysis

<table>
<thead>
<tr>
<th>Memory Access</th>
<th>µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU only</td>
<td>0.06</td>
</tr>
<tr>
<td>GPU only</td>
<td>0.84</td>
</tr>
<tr>
<td>GPU to CPU, Host Memory</td>
<td>1.41</td>
</tr>
<tr>
<td>GPU to CPU, Device Memory</td>
<td>1.61</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queues</th>
<th>µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU only</td>
<td>0.08</td>
</tr>
<tr>
<td>GPU only</td>
<td>2.36</td>
</tr>
<tr>
<td>GPU to CPU (H-G)</td>
<td>2.07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inter-GPU Transfers</th>
<th>µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intra-Node GPU Copy</td>
<td>15</td>
</tr>
<tr>
<td>Inter-Node MPI_Send, Host Memory</td>
<td>2.5</td>
</tr>
<tr>
<td>Inter-Node MPI_Send, Device Memory</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Notified Access, Host to Host</th>
<th>Host µs</th>
<th>Device µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer Device PUT</td>
<td>2.13</td>
<td>52.17</td>
</tr>
<tr>
<td>Peer Device GET</td>
<td>1.88</td>
<td>51.69</td>
</tr>
<tr>
<td>Remote Device PUT</td>
<td>3.93</td>
<td>11.14</td>
</tr>
<tr>
<td>Remote Device GET</td>
<td>3.72</td>
<td>15.31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU RMA Notified Access</th>
<th>PUT µs</th>
<th>GET µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device-Local</td>
<td>13.21</td>
<td>13.09</td>
</tr>
<tr>
<td>Peer Device</td>
<td>56.38</td>
<td>44.36</td>
</tr>
<tr>
<td>Remote Device</td>
<td>26.40</td>
<td>32.27</td>
</tr>
</tbody>
</table>

*Table 4.1: System Latency Overview (Roundtrip Time)*
4 Evaluation

In the CPU- and GPU-only measurements the two processes or blocks perform the ping-pong on their local host or device memory involving only the memory bus. First of all we notice the GPU memory access time is a factor 14 slower with a \(0.84\mu\text{s}\) roundtrip time compared to \(0.06\mu\text{s}\) for the CPU. The slower memory access time on the GPU will be apparent on all further experiments involving more memory operations and expectations from CPU performance will need to adapt. In the GPU to CPU experiment the first actor is a Block 1 on the GPU and the second actor is Process 1 on the host. The ping-pong flag value resides either in the host memory local to Process 1 or the device memory local to Block 1. In this experiment the writing or polling of one of the two actors traverses the PCI-E bus. The host accesses the device memory using \texttt{gdrcopy}\ (cf. Section 3.1.3). We observe the ping-pong time increases by the well-known PCI-E delay of \(0.5\mu\text{s}\) to \(1.41\mu\text{s}\) and \(1.61\mu\text{s}\) for the flag value in host and device memory respectively. These four memory access times establish our system performance baseline.

**Queues** In GPU RMA queues are an important building block for device and host interaction as well as for notification handling. Our experiment uses a queue payload of 64KiB which is the single payload size used for all queues in our GPU RMA implementation. The experiment employs a separate queue for the ping and the pong message. For the GPU to CPU experiment the queue targeted at the CPU resides in host memory and the queue targeted at the GPU resides in device memory as is the case for all queues in GPU RMA (cf. Section 3.2.4).

For the CPU-only experiment the queue time only slightly increases compared to the memory ping-pong. The GPU-only queue is a factor 30 slower than the CPU-only queue requiring \(2.36\mu\text{s}\) and a factor 3 slower than the GPU-only memory access time. If we consider an enqueue operation involves a single write to global memory and a dequeue operation requires a read of the payload and a write of the public tail (cf. Section 3.2.4) the resulting queue time is reasonable. The queue ping-pong between the device and the host involving the PCI-E bus lies in between.

The slow GPU memory performance compared to the CPU was previously reported. Oden et al. [31] state that the GPU is not optimized for low-latency memory access provided by sophisticated cache architectures on the CPU but employs latency hiding by several thousands threads executing in parallel.

**Inter-GPU Transfers** In the **Memory** and **Queues** sections we focused on latencies involving a single GPU. Now we focus on inter-GPU communication and establish the remaining baseline parameters induced by our system configuration.

The intra-node latency measures the time to copy data from the GPU 1 device memory to the GPU 2 device memory and back over the PCI-E bus (cf. Figure 4.1). The copy is initiated by the host using \texttt{cudaMemcpy} from the CUDA runtime. The resulting \(15\mu\text{s}\) based on the latency test of the CUDA toolkit is far from what we would expect by two GPUs directly connected over a PCI-E switch. We were in contact with the system administrators of CSCS. To our regret, we could not resolve latency issue. A latency of \(3\mu\text{s}\) on a previous-generation Fermi-class GPU and an older Westmere host with the
additional overhead of MPI has previously been reported by Potluri et al. [37].

The inter-node latency measures the time to transfer data from the GPU 1 device memory on Node 1 to the device memory of GPU 1 on Node 2. The data transfer involves the PCI-E bus from GPU 1 to the host CPU 1, QPI from CPU 1 to CPU 2, PCI-E again towards the IB network adapter, then crosses the InfiniBand link to Node 2 having the same topology where the same path as on Node 1 is traversed in reverse to reach the GPU’s device memory (cf. Figure 4.1). We note that this configuration is suboptimal. The memory transfer is initiated by two processes running on the two nodes. They communicate using MPI message passing using the CUDA-aware OpenMPI that accepts the GPU device memory pointer and leverages GPUDirect RDMA.

The host memory ping-pong time of 2.5µs is provided as a reference for normal MPI operation. When device memory buffers are involved the roundtrip time increases by a factor of 4 to 10µs. The MVAPICH group reports a similar latency of 5.37µs (one-way) but for single-socket SandyBridge nodes with a Tesla K20c GPU and FDR InfiniBand [36]. It is a well-known problem that PCI-E forwarding over QPI as is the case in our system significantly decreases system performance [6, 13, 36, 38]. The Cirrascale company reports a loss of 40% bandwidth and 300% increase in latency compared to peer access on the same PCI-E root and suggests communicating with GPUs on another socket using an InfiniBand connection that uses GPUDirect RDMA and avoids QPI [6]. In addition, a bug introduced with the SandyBridge chipset causes PCI-E bandwidth to slow down as low as 250 MB/s when the data is transferred from a PCI-E device rather than to it. Several related research results suggest copying data buffers to host memory rather than using GPUDirect RDMA and transfer the data from the host to the NIC to achieve better performance [13, 36]. Detailed benchmarks of GPUDirect RDMA under various system configurations published by a Nvidia employee show a minimum latency of 1.9µs (one-way) can be achieved [38].

**Notified Access, Host to Host**  We implemented GPU RMA notified access by a set of two messages exchanged by the corresponding hosts. The GPU requests the transfer by sending a command to its block manager who will perform the corresponding MPI communication. We send the metadata message followed by the data for a PUT request and we send the metadata followed by a data receive for a GET request (cf. Section 3.2.3, Figure 3.3). In the Notified Access experiment we measure the time for the host-controlled part of GPU RMA notified access. The first actor performs a notified PUT or GET. As soon as the notified access has finished, i.e. both messages are exchanged, the second actor replies with a notified access. In GPU RMA notified access, we send the metadata from the origin host memory to the target host memory and the data from the origin device memory to the target device memory. To understand the difference between the performance of our chosen notified access protocol and the additional overhead introduced by direct GPU device memory transfers we measured the notified access twice, reflected by two columns in the Notified Access section of our summary table. The Host column reports the measured roundtrip time where the data buffer resides in host memory whereas the Device column reports the measured roundtrip time where the data buffer resides in device memory, the actual GPU RMA
configuration. For both setups the metadata buffer reside in host memory as is always the case for GPU RMA. In our implementation the metadata message has a size of 28B. For the experiment we chose a notified access size of 4B.

The first two rows report peer device latencies for transfers between GPUs on the same PCI-E root. The host numbers of 2.13\(\mu s\) for PUT and 1.88\(\mu s\) for GET are higher than the memory access times or queue times we reported beforehand. In addition to these results communication is now handled by MPI and involves two processes. The device results for peer notified access are above 50\(\mu s\) and are highly confusing. We already reported the unacceptable 15\(\mu s\) peer-to-peer transfer in the Inter-GPU section and mentioned we could not resolve the underlying cause. For a meaningful discussion of peer device transfers we first need a system with a working peer configuration and remain with presenting the numbers as-is.

The inter-node latencies are as expected. Notified access exchanges two MPI messages and the reported host numbers of 3.95\(\mu s\) for PUT and 3.72\(\mu s\) for GET are below a factor of two from the 2.5\(\mu s\) measured for a single message roundtrip. For GPU device memory buffers the results are even better. The remote PUT takes 11.14\(\mu s\) which is just above the 10\(\mu s\) for a single message. Recall from the notified access sequence diagram (Figure 3.3) that PUT issues two successive MPI sends, the host metadata and the device data. The two messages are sent together and yield a smaller overall latency than the two individual transfers of 2.5\(\mu s\) and 10\(\mu s\) together. The GET roundtrip time of 15.31 is substantially higher. This is the case for all GET numbers reported for Host and Device, be it for inter- or intra-node notified access. We can give reason for the slower GET time be referring to the sequence diagram again. The data message is not sent by the origin but by the target. First, the target must receive the metadata and only than may it initiate the response data message, explaining the higher overall latency. We note that in code there is no difference between a PUT and a GET operation but that for the data transfer the \texttt{MPI\_Isend} and \texttt{MPI\_Irecv} calls are swapped between the origin and target.

**GPU RMA Notified Access** We finally conclude with the GPU RMA notified access latency observed by a GPU RMA rank. In this experiment the two actors are GPU RMA ranks. The first rank performs a notified access and the second rank polls for the notification. Once it matches the notification it replies with a notified access and the overall ping-pong time is reported. We start by reviewing all the contributions to overall latency.

At a notified access call the device-side library first consults the window information in global memory to determine if the data is copied to a peer rank on the same device or to a different GPU, \(t_{Win}\). For either case it enqueues a command to the block manager to generate the notification or initiate the remote transfer, \(t_{Cmd}\). For inter-GPU communication the block manager performs the corresponding MPI communication we just reviewed in the last section, \(t_{NA-\text{Host}}\). For device-local copies the rank performs the copy itself before issuing the command, \(t_{NA-\text{Dev}}\). After concluding the MPI communication, or immediately if the target rank resides on the same GPU, the target block manager enqueues the notification to the rank’s notification queue in device memory, \(t_{Notif}\). Finally, the target rank tests for the incoming notification by polling the notification queue,
4.1 System Latency Analysis

\[ t_{\text{Poll}} \]. We summarize the overall notified access latency as \( t_{\text{NA}} \).

\[ t_{\text{NA}} = t_{\text{Win}} + t_{\text{Cmd}} + t_{\text{NA-Host/Dev}} + t_{\text{Notif}} + t_{\text{Poll}} \]

The command time \( t_{\text{Cmd}} \) and notification time \( t_{\text{Notif}} \) are queue latencies we have already reported and discussed. We also elaborated on the host to host notified access time \( t_{\text{NA-Host}} \) in the last section. The window lookup time \( t_{\text{Win}} \) and the notification testing time \( t_{\text{Poll}} \) remain unmeasured and contribute the remaining time of the measured end to end GPU RMA notified access time. We analyze the device-local and inter-node remote device latencies and omit the peer-device discussion for they remain questionable.

The device-local PUT and GET latencies \( t_{\text{NA}} \) are 13.21\( \mu s \) and 13.03\( \mu s \) respectively. For the queues we previously reported \( t_{\text{Cmd}} = t_{\text{Notif}} = 2.07\mu s \) and the host to host communication time \( t_{\text{NA-Host}} \) does not apply because the target rank resides on the same device. The device-local copy time \( t_{\text{NA-Dev}} \) applies. Rounding the latency numbers we analyze PUT and GET latencies together and determine the remaining latency.

\[ t_{\text{Win}} + t_{\text{NA-Dev}} + t_{\text{Poll}} = t_{\text{NA}} - t_{\text{Cmd}} - t_{\text{Notif}} \\
= 13\mu s - 2\mu s - 2\mu s \\
= 9\mu s \]

The results need careful interpretation. We have combined results from individual microbenchmarks and the overall system might behave differently. For the GPU we have reported a memory access time of 0.84\( \mu s \). If we assume mutually dependent memory accesses, the remaining 9\( \mu s \) latency corresponds to 10 memory operations. We can assign 2 dependent memory accesses to the window lookup and a minimum of 3 dependent memory accesses to notification polling. We count the copy operation as a single dependent memory accesses yielding a total of 6 dependent accesses, requiring 5\( \mu s \). Our analysis leaves 4\( \mu s \) open for further investigation.

For remote device notified access on different node, we measure a roundtrip time of \( t_{\text{NA,PUT}} = 26.40\mu s \) and \( t_{\text{NA,GET}} = 32.27\mu s \). With the queue times \( t_{\text{Cmd}} = t_{\text{Notif}} = 2.07\mu s \) as in the device-local case and host notified access times of \( t_{\text{NA-Host,PUT}} = 11.14\mu s \) and \( t_{\text{NA-Host,GET}} = 15.31\mu s \) as reported earlier, we determine the remaining latencies.

\[ t_{\text{Win}} + t_{\text{Poll}} = t_{\text{NA}} - t_{\text{Cmd}} - t_{\text{NA-Host}} - t_{\text{Notif}} \\
\text{PUT} = 26.40\mu s - 2.07\mu s - 11.14\mu s - 2.07\mu s \\
= 11.12\mu s \\
\text{GET} = 32.27\mu s - 2.07\mu s - 15.31\mu s - 2.07\mu s \\
= 12.82\mu s \]

The time we attribute to window lookup and notification polling comparable for PUT and GET and their latency difference is mainly due to the difference in the host to host notified access time we have already discussed. We note that the remaining latency is also close to the 9\( \mu s \) we observe for the device-local case. The only difference is that the device-local notified access also performs the copy on the device. As this is a single 4B write in our experiment, this time difference is no more but the memory access
4 Evaluation

time of $0.84\mu s$ we reported. The closeness of the additional latency in device-local and inter-node notified access we could not measure reassures us that it is actually due to device-local operations.

In summary, we measured all communication-related latencies of our GPU RMA implementation. Our experiment results yield the same unknown remaining latency of about $10\mu s$ for on-device operations such as window lookup and notification polling. These on-device library operations represent 70% of device-local notified access latency and 40% of inter-node remote device notified access latency. Host and device interaction contributes the remaining 30% latency for device-local notified access which generates the notification. Our MPI-based host to host notified access protocol represents 42% and 47% of notified PUT and GET latency for inter-node communication respectively.

4.2 GPU RMA Performance Model

Performance models are an important tool in high performance computing to predict application performance and allow for guided tuning and optimization. We provide a communication performance model for our GPU RMA library implementation to enable application programmers to use analytical analysis with GPU RMA. We employ the Hockney model and derived the model parameters from empirical measurements on our greina development system detailed in Section 4.1.

The Hockney model [12] describes communication performance with the two parameters latency $L$ and bandwidth $B$. The required communication time $t$ depends on the message size $s$.

$$ t(s) = L + \frac{s}{B} $$

We measured the communication time dependent on the message size to derive the two model parameters for GPU RMA using the rma_latency_test also used for the system latency analysis. Depending on the communication target different control flow and data paths result. We provide distinct model parameters dependent on the communication target location either on the same GPU, a peer device on the same PCI-E root, or a device on a remote node. Our underlying MPI implementation employs a different communication protocol depending on the message size for optimal performance. We provide separate model parameters per communication target where different communication protocols apply. To determine the minimal latency we measured half the ping-pong time for a notified access of size 4B. We increased the message size until the communication bandwidth was saturated to determine the communication bandwidth. Our analysis results are summarized in Table 4.2.

For device-local communication we observe a $6.4\mu s$ latency already known from the system latency analysis and a bandwidth of 4.5 GB/s. There are several ways to measure on-device communication bandwidth. We report the bandwidth for one rank and note that the overall GPU memory bandwidth is shared amongst all ranks. Further, the number of threads per rank allow to generate different memory loads. Our measured
4.2 GPU RMA Performance Model

<table>
<thead>
<tr>
<th>Communication</th>
<th>Latency [µs]</th>
<th>Bandwidth [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PUT</td>
<td>GET</td>
</tr>
<tr>
<td>Same-device</td>
<td>6.4</td>
<td>6.4</td>
</tr>
<tr>
<td>Peer-device &lt; 4KiB</td>
<td>28</td>
<td>22</td>
</tr>
<tr>
<td>Peer-device &gt; 4KiB</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>Remote-device &lt; 5KiB</td>
<td>13.3</td>
<td>16.3</td>
</tr>
<tr>
<td>Remote-device &gt; 5KiB</td>
<td>33.2</td>
<td>24.2</td>
</tr>
<tr>
<td>GPU Memory Bandwidth</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: GPU RMA Performance Model Parameters

bandwidth is for a single block of 256 threads running on the GPU. We note further that concurrent computation and communication will cause additional contention on the memory bus.

For peer device communication on the same node we noticed a reduction of communication latency starting from a message size of 4KiB. While no protocol switch is documented by OpenMPI we report different model parameters for transfer sizes smaller and larger than 4KiB. The latencies are large as we discussed in the system latency analysis. We achieve a bandwidth of 12.4 GB/s. The GPUs are directly connected via a PCI-E 3 x16 switch with a theoretical bandwidth of 15.75 GB/s. With GPU RMA, we achieve 79% of the theoretical bandwidth.

For remote device communication targeting a GPU on a different node the analysis is more involved. We mentioned the problem of low PCI-E bandwidth for peer to peer transfers between devices due to a bug in the recent Intel chipsets in the system latency analysis. This bandwidth is critical to leverage GPUDirect RDMA which involves peer PCI-E transfers between the GPU and the NIC. As a result, many inter-node GPU communication protocols including the OpenMPI implementation revert to the pre-GPUDirect approach and copy the transfer buffers to host memory first once the limited peer PCI-E bandwidth voids the better GPUDirect RDMA latency. The bandwidth can be as low as 250 MB/s [38]. OpenMPI allows to configure at what message size it switches from a GPUDirect RDMA-based protocol to staging transfer buffers through host memory. Further GPU-related tuning parameters exist and we refer to the CUDA section of the OpenMPI documentation for details [34]. The set of other relevant configuration parameters were discussed under GPU RMA deployment in Section 3.3 but only the GPUDirect protocol switch is relevant to our model discussion.

We measured GPU RMA remote device latency for both communication protocols separately and determined a message size of 5KiB optimizing overall communication latency. We provide model parameters for message sizes below and above the protocol switch threshold. With GPUDirect RDMA we achieve a communication bandwidth of barely 310 MB/s. When buffers are first copied to host memory from a message size of 5KiB and upwards, we achieve 8.5 GB/s and 9.1 GB/s respectively for PUT and GET. Our nodes are connected with 4x EDR InfiniBand with a theoretical bandwidth of 12 GB/s.
4 Evaluation

We achieve 70% of the theoretical bandwidth for PUT and 75% theoretical bandwidth for GET. Notice that while copying buffers through host incurs a higher latency the better bandwidth is the determining factor for large messages.

When applying the GPU RMA performance model the user must first clarify the location of the target rank and take the message size into consideration to select the correct model parameters. While GPU RMA provides a uniform view of the GPU cluster resources by means of GPU RMA ranks, modelling application performance cannot neglect the actual hardware topology faced.

We conclude our model by adding the GPU memory bandwidth. While not relevant for GPU RMA communication it is the parameter missing to model memory-bound applications such as our horizontal diffusion case study in Section 4.3. We report a GPU memory bandwidth of 140 GB/s based on our stencil experiments and note that memory bandwidth is highly application-dependent. The Nvidia Tesla K80 theoretical memory bandwidth is 240 GB/s.

4.3 Case Study: Horizontal Diffusion

We performed a case study of GPU RMA performance based on the horizontal diffusion algorithm part of the COSMO weather prediction and climate modelling software developed and deployed by a consortium of European meteorological services that has shown interest by the research community for automated optimization [3, 9]. Our case study serves three purposes. First, we demonstrate that GPU RMA is mature enough to be used as a middleware to solve a computational problem on a GPU cluster. Second, we validate the accuracy of the theoretical GPU RMA performance model derived in Section 4.2 by comparing it to performance measurements. Finally, we evaluate the overhead of GPU RMA by comparing it to an implementation using MPI and CUDA manually.

4.3.1 Algorithm

Horizontal Diffusion is a stencil-based algorithm. Stencils are a common form of structured grid computations representing one of the seven classes of relevant parallel research problems [1]. The simplified algorithm from the COSMO model applies multiple dependent 2D stencils to a grid of data points such as air humidity. A stencil defines a computation on the fixed neighborhood of a grid point and this fixed size local computation is applied to every data point on the grid.

We compute horizontal diffusion iteratively. Each iteration consists of four dependent stencils. The output of one iteration is used as input for the next iteration. The four
4.3 Case Study: Horizontal Diffusion

Stencils lap, fli, flj and out are defined as follows:

\[
\begin{align*}
\text{lap}_{i,j} & = -4 \cdot \text{in}_{i,j} + \text{in}_{i-1,j} + \text{in}_{i+1,j} + \text{in}_{i,j-1} + \text{in}_{i,j+1} \\
\text{fli}_{i,j} & = \text{lap}_{i+1,j} - \text{lap}_{i,j} \\
\text{flj}_{i,j} & = \text{lap}_{i,j+1} - \text{lap}_{i,j} \\
\text{out}_{i,j} & = (\text{fli}_{i-1,j} - \text{fli}_{i,j}) + (\text{flj}_{i,j-1} - \text{flj}_{i,j})
\end{align*}
\]

Stencil Dependencies

\[
\text{in} \to \text{lap} \to \text{fli} \to \text{out} \\
\uparrow \quad \text{flj}
\]

To parallelize the computation we split the input grid into horizontal bands. Every rank is assigned an equal number of rows to perform the computation. All stencils except flj depend on values from the row above and below the current point to be computed. This point is potentially assigned to a neighboring rank. Before the stencil can be computed, these boundary rows need to be exchanged between the ranks. The ranks always exchange the rows needed to compute one iteration of the next stencil. The stencil result, e.g. lap, is then exchanged between the ranks in order to execute the next stencil, fli.

We implemented three different variants of horizontal diffusion. The straightforward variant HD follows the basic stencil pattern: First, the halo rows are exchanged. Then, the stencil is executed. This is not efficient because the GPU is idling while waiting for the needed rows to arrive. To use computation and communication overlap, we implemented the second variant HD-OVERLAP. In this variant we apply horizontal diffusion to two different data sets alternatively. When the first stencil of one computation has finished, its result rows are exchanged. Instead of waiting, HD-OVERLAP now starts to compute the first stencil of the second horizontal diffusion. Because of overlapping computation and communication, this variant is expected to be more efficient. However, HD-OVERLAP cannot be compared to HD directly as it performs two horizontal diffusion computations. The third variant, HD-NOOVERLAP, performs two horizontal diffusion computations as well but doesn’t employ computation and communication overlap. It is used as a serialized reference for HD-OVERLAP. The three variants are summarized in Listing 4.1.

4.3.2 HD Performance Model

Based on the GPU RMA performance model we derive a model for the horizontal diffusion execution time. In the analysis we assume an \( m \times n \) grid of height \( m \) and width \( n \). For the three horizontal diffusion variations, we consider the computation time \( t_M \), the communication time \( t_R \) and the overall execution time \( t \). The time is modelled for one iteration of horizontal diffusion with respect to a single GPU RMA rank. Given the uniform computation and communication among all ranks we expect this time to match the execution time of all ranks running concurrently. We neglect the fact that the first and last rank only need to share their halo rows with one neighbor instead of two. An overview of the introduced variables is given in Table 4.3.

63
4 Evaluation

Listing 4.1: Horizontal Diffusion Variations

```
// HD Variation
foreach iteration:
    send_halos(inb)
    wait_halos(inb)
    compute_lap(inb, lap)
    send_halos(lap)
    wait_halos(lap)
    compute_fli(lap, fli)
    compute_flj(lap, flj)
    send_halos(fli)
    wait_halos(fli)
    compute_out(fli, flj, out)
    swap_buffers(out, inb)

// HD-OVERLAP Variation
send_halos(inb[0])
send_halos(inb[1])
foreach iteration:
    wait_halos(inb[0])
    compute_lap(inb[0], lap[0])
    send_halos(lap[0])
    wait_halos(inb[1])
    compute_lap(inb[1], lap[1])
    send_halos(lap[1])
    wait_halos(lap[0])
    compute_fli(lap[0], fli[0])
    compute_flj(lap[0], flj[0])
    send_halos(fli[0])
    wait_halos(lap[1])
    compute_fli(lap[1], fli[1])
    compute_flj(lap[1], flj[1])
    send_halos(fli[1])
    wait_halos(fli[0])
    compute_out(fli[0], flj[0], out[0])
    send_halos(out[0])
    wait_halos(fli[1])
    compute_out(fli[1], flj[1], out[1])
    send_halos(out[1])
    swap_buffers(out[0], inb[0])
    swap_buffers(out[1], inb[1])

// HD-NOOVERLAP Variation
foreach iteration:
    send_halos(inb[0])
    send_halos(inb[1])
    wait_halos(inb[0])
    wait_halos(inb[1])
    compute_lap(inb[0], lap[0])
    compute_lap(inb[1], lap[1])
    send_halos(lap[0])
    send_halos(lap[1])
    wait_halos(lap[0])
    wait_halos(lap[1])
    compute_fli(lap[0], fli[0])
    compute_flj(lap[0], flj[0])
    compute_fli(lap[1], fli[1])
    compute_flj(lap[1], flj[1])
    send_halos(fli[0])
    send_halos(fli[1])
    wait_halos(fli[0])
    wait_halos(fli[1])
    compute_out(fli[0], flj[0], out[0])
    compute_out(fli[1], flj[1], out[1])
    swap_buffers(out[0], inb[0])
    swap_buffers(out[1], inb[1])
```
4.3 Case Study: Horizontal Diffusion

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>Grid Height (rows)</td>
</tr>
<tr>
<td>$n$</td>
<td>Grid Width (columns)</td>
</tr>
<tr>
<td>$B_{M}$</td>
<td>GPU Memory Bandwidth</td>
</tr>
<tr>
<td>$B_{RMA}$</td>
<td>GPU RMA Communication Bandwidth</td>
</tr>
<tr>
<td>$L_{Put}$</td>
<td>GPU RMA Put Latency</td>
</tr>
<tr>
<td>$R$</td>
<td>Number of Ranks in the System</td>
</tr>
<tr>
<td>$R_I$</td>
<td>Number of Ranks on one GPU (local ranks)</td>
</tr>
<tr>
<td>$r = \frac{m}{R}$</td>
<td>Number of Grid Rows assigned to each Rank</td>
</tr>
<tr>
<td>$B_{MR} = \frac{B_{M}}{R_I}$</td>
<td>Memory Bandwidth per GPU RMA rank</td>
</tr>
<tr>
<td>$S_M = 8(r \times n)$</td>
<td>Stencil Buffer Size</td>
</tr>
<tr>
<td>$S_R = 8n$</td>
<td>Halo Row Size</td>
</tr>
</tbody>
</table>

Table 4.3: Overview of the Horizontal Diffusion Model Parameters

**HD Performance Model**

We start with the HD variation shown in Listing 4.1. Recall that the input grid is split horizontally amongst all ranks. We denote by $r$ the number of rows assigned to each rank and by $R$ the number of ranks. By definition, we get the relation $m = r \cdot R$. For the execution of a single rank we deal with a grid of size $r \times n$. We discuss the computation time $t_M$ first.

Computation time is limited by the memory bandwidth. All of the five data buffers ($\text{inb}$, $\text{lap}$, $\text{fli}$, $\text{flj}$, $\text{out}$) are represented as an $r \times n$ double precision array plus an additional halo row on top and on the bottom. For the discussion of the performance model, we neglect the two halo rows. This yields a data buffer size of $S_M = 8(r \times n)$ bytes. The memory bandwidth $B_{M}$ is given in our GPU RMA model and is shared amongst all ranks running on the device. We call them the local ranks $R_I$. When considering memory access times, we have to consider the memory bandwidth available to one rank only. We denote the per-rank memory bandwidth as $B_{MR} = \frac{B_{M}}{R_I}$. The stencils $\text{lap}$, $\text{fli}$ and $\text{flj}$ each read exactly one buffer and write one buffer and their execution time is determined by the read and write times of these buffers. The out stencil reads two buffers and writes one buffer. In HD, these stencils are executed sequentially. If we sum
up the stencil execution times, we get the overall HD computation time $t_{\text{HD},M}$:

$$
t_{\text{M,\text{lap}}} = t_{\text{M,\text{fli}}} = t_{\text{M,\text{flj}}}
= \text{Buffer Read Time} + \text{Buffer Write Time}
= \frac{S_M}{B_{\text{MR}}} + \frac{S_M}{B_{\text{MR}}} = 2\left(\frac{8(r \times n)}{B_M/R_l}\right)
$$

$$
t_{\text{M,out}} = 2 \times \text{Buffer Read Time} + \text{Buffer Write Time}
= 2\left(\frac{S_M}{B_{\text{MR}}} \right) + \frac{S_M}{B_{\text{MR}}} = 3\left(\frac{8(r \times n)}{B_M/R_l}\right)
$$

$$
t_{\text{HD,M}} = t_{\text{M,\text{lap}}} + t_{\text{M,\text{fli}}} + t_{\text{M,\text{flj}}}
+ t_{\text{M,out}}
= 9\left(\frac{8(r \times n)}{B_M/R_l}\right)
$$

The RMA communication time $t_R$ is characterized by the GPU RMA put latency $L_{\text{Put}}$ and bandwidth $B_{\text{RMA}}$. For the three stencils lap, fli and out, we need to exchange the halo rows. The stencil flj only depends on horizontal neighbors which are part of the rank’s horizontal band. Using double precision data, a halo row uses $S_R = 8n$ bytes. We send the top row of our buffer to our top neighbor and the bottom row to our bottom neighbor. Because we issue these data transfers at the same time, we only count the RMA latency once. Summing up the three halo exchanges for the dependent stencils, we get the overall GPU RMA communication time $t_{\text{HD,R}}$.

$$
t_{\text{Halo}}(x) = \frac{x}{B_{\text{RMA}}} + L_{\text{Put}}
$$

$$
t_{\text{HD,R}} = 3 \cdot \text{Halo Exchange Time} = 3 \cdot t_{\text{Halo}}(2S_R) = 3\left(\frac{2 \cdot 8n}{B_{\text{RMA}}} + L_{\text{Put}}\right)
$$

In HD, input halos are exchanged after the computation of the previous stencil and the computation has to wait for the rows to arrive in order to compute the next stencil. Summing up computation and communication time, we get the overall execution time of horizontal diffusion in HD for a single rank $t_{\text{HD}}$.

$$
t_{\text{HD}} = t_{\text{HD,M}} + t_{\text{HD,R}}
$$

**HD-NOOVERLAP Performance Model**

The HD-NOOVERLAP performance model follows directly from the HD model. The difference between the two is that HD-NOOVERLAP computes two independent horizontal diffusions. As the buffers are independent, the computation time doubles. The communication time doubles as well. However, since we exchange all the halos at the same time, we only consider the RMA latency $L_{\text{Put}}$ once. A total of four rows are sent. Taking
4.3 Case Study: Horizontal Diffusion

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>lap₀</td>
<td>lap₁</td>
<td>flx₀</td>
<td>flx₁</td>
<td>out₀</td>
<td>out₁</td>
</tr>
<tr>
<td>Communication</td>
<td>out₁</td>
<td>lap₀</td>
<td>lap₁</td>
<td>flx₀</td>
<td>flx₁</td>
<td>out₀</td>
</tr>
</tbody>
</table>

*Figure 4.2: Phases of a HD-OVERLAP Iteration.*

due to these differences in account, the performance model follows immediately.

\[ t_{\text{HD-NOOVERLAP},M} = 2 \cdot t_{\text{HD},M} \]
\[ t_{\text{HD-NOOVERLAP},R} = 3 \cdot t_{\text{Halo}}(4S_R) \]
\[ t_{\text{HD-NOOVERLAP}} = t_{\text{HD-NOOVERLAP},M} + t_{\text{HD-NOOVERLAP},R} \]

**HD-OVERLAP Performance Model**

HD-OVERLAP executes two runs of horizontal diffusion and employs computation and communication overlap. As seen in Listing 4.1, the computation alternates between stencil computations of the two runs. While computing the stencil of one run, the halos for the next stencil of the other run are exchanged. We illustrate this computation and communication overlap pattern in Figure 4.2 as multiple phases of one horizontal diffusion iteration. The execution of one horizontal diffusion run follows a zig-zag pattern between computation and communication. We refer to the execution of the stencils \(\text{fli}\) and \(\text{flj}\) together as \(\text{flx}\).

Every phase is dependent on the previous one and cannot start before it has ended. This is obvious for the computation as a rank always executes one stencil. The computation must also wait for the communication part of the last phase to finish because it depends on these halo rows being present. The communication phase may not start earlier as it needs the results from the computation of the previous phase. We model the total execution time of a HD-OVERLAP iteration as the sum of the execution times of each phase. In the analysis, we assume communication requires no execution time on the GPU and depends only on the network. This simplification neglects issuing the communication requests and testing for incoming notifications\(^1\). The execution time of each phase is the maximum time required by either computation or communication.

In the communication part, the halo rows of one stencil are sent to the neighboring ranks. This is the same as the communication of the two halo rows in the HD variation and the same for all phases: \(t_{\text{Halo}}(2S_R)\).

The computation times for the stencils were already derived for the HD variation as \(t_{M,\text{lap}}, t_{M,\text{fli}}, t_{M,\text{flj}}\) and \(t_{M,\text{out}}\). The computation \(\text{flx}\) denotes the stencils \(\text{fli}\) and \(\text{flj}\) computed in sequence and requires the time of the two stencils combined.

\(^1\) The issuing and testing overheads are partly included in the GPU RMA latency numbers. Further, the GPU RMA performance depends on the communication target. The no GPU time assumption for communication only holds as long as device-local communication is not dominating. Device-local communication is implemented in the kernel and is not offloaded to DMA or RDMA (cf. Section 3.2.3). We will come back to this when evaluating device-local horizontal diffusion performance.
4 Evaluation

Having established these times, we can formulate the time required by each phase $p$ and provide a formula for the overall execution time of HD-0VERLAP.

$$t_{\text{HD-0VERLAP}, p=\{1,2\}} = \max \{t_{M,\text{lap}}, t_{\text{Halo}}(2S_R)\}$$

$$t_{\text{HD-0VERLAP}, p=\{3,4\}} = \max \{t_{M,\text{r1i}} + t_{M,\text{r1j}}, t_{\text{Halo}}(2S_R)\}$$

$$t_{\text{HD-0VERLAP}, p=\{5,6\}} = \max \{t_{M,\text{out}}, t_{\text{Halo}}(2S_R)\}$$

$$t_{\text{HD-0VERLAP}} = \sum_{p=1}^{6} t_{\text{HD-0VERLAP}, p}$$

**Remark: Choosing GPU RMA Model Parameters**

The horizontal diffusion performance model uses the GPU RMA communication bandwidth $B_{\text{RMA}}$ and latency $L_{\text{PUT}}$. The GPU RMA performance model discussed in Section 4.2 provides different numbers for these parameters. Choosing the correct numbers depends on two factors.

First, the latency and bandwidth depends on whether the RMA communication target resides on the same GPU, a peer GPU on the same node, or a GPU on a remote node. In horizontal diffusion, all ranks perform the same communication pattern and volume. Therefore, we need to discuss this question only once for all ranks. To model an upper bound, only the maximum communication time is relevant. The parameters that maximize communication time must be chosen. Which of the three target types to include into this selection depends on the scenario of interest. Only the RMA target types that actually occur in the hardware configuration modelled should be included.

Second, the GPU RMA model provides different numbers for the same RMA target type. Depending on the transfer size, the underlying MPI implementation uses a different protocol. While the three horizontal diffusion variations implemented exchange different number of halos concurrently, the transfer size issued is always the same: one halo row. If multiple numbers are provided for a target type, the correct parameters are chosen according to the halo row size $S_R = 8n$.

The three performance models are available as Python scripts in the implementation directories of the respective GPU implementations. Next to the characteristic numbers discussed they also report the GPU RMA model chosen.

**Corollary: Computation and Communication Ratio**

Choosing the computation grid size determines the computation-communication ratio. Because the grid is split into horizontal bands, the communication load is unchanged when changing the grid height and only computational load changes. When varying the grid width, the number of grid points to compute and the number of grid points to exchange with neighbors changes proportionally, allowing to investigate the tradeoff at different transfer sizes. While the proportion of grid points to compute and grid points
4.3 Case Study: Horizontal Diffusion

to communicate remains constant, the computation to communication time ratio does not as they are related by the memory and network bandwidth respectively. One would expect the communication time to become dominant when increasing the grid width because network bandwidth is much lower than the GPU memory bandwidth. While the latter is true, GPU memory bandwidth $B_M$ is shared amongst all ranks executing on the same GPU resulting in the per-block memory bandwidth $B_{MR}$ which is often lower than the network bandwidth. In addition, only two of the ranks on the GPU exchange halo rows with a remote GPU. The other ranks exchange halos with on-device neighbors and take another share of the memory bandwidth. Taking these effects into account, computation time quickly becomes the dominating factor in overall execution time. To achieve a maximum communication share, the grid height and width have to be chosen as small as possible.

4.3.3 Implementation

We implemented the three horizontal diffusion variations twice. One implementation uses the GPU RMA library. The other implementation does not depend on GPU RMA but uses MPI and CUDA manually. We refer to this implementation as MPI CUDA. A rank in GPU RMA or a block in the MPI CUDA implementation is called a worker.

The input grid for horizontal diffusion is split horizontally amongst all workers. If a stencil depends on the result of a previous stencil, the row above and below the worker’s row range have to be exchanged with the neighboring workers. This is indicated by `send_halos` in Listing 4.1. When the halo rows have arrived, `wait_halos`, the stencil can be executed. A total of five buffers are involved in every horizontal diffusion computation: The input field `inb`, the laplace result `lap`, the fli and flj results `fli`, `flj`, and the output `out`.

In the GPU RMA implementation, the host initializes the GPU memory with the rows the ranks on this GPU are assigned to. Then the GPU RMA kernel launches. Every buffer is registered as an RMA window having an extra halo row on the top and the bottom. The halo rows are communicated to the neighboring ranks using a notified PUT operation. The PUT is tagged with the current horizontal diffusion iteration. Before computing a dependent stencil, the kernel waits for the notification from the neighboring rank to arrive. The windows are chosen such that the ranks residing on the same GPU register overlapping windows. The top halo row required by rank $r$ is aligned with the bottom buffer row of rank $r - 1$. This allows for the GPU RMA library to use no-copy optimization: As the origin and target address match, the copy is omitted and only the notification is sent. This won’t lead to data races. A dependent stencil will only read the data once it has received the notification. Therefore, it doesn’t read old data from rank $r - 1$. Further, since rank $r - 1$ will compute other stencils that need halo rows from the current rank $r$, it won’t overwrite the halo rank $r$ currently needs to compute its stencil. Note that this is a performance optimization that is neither needed nor required. GPU RMA checks for equal origin and target addresses and then omits the copy. This is also not explicitly visible in the horizontal diffusion implementation code and we don’t need to modify the library calls. The implementation would work correctly if
the memory addresses were not aligned accordingly. The optimization is applicable by choosing the memory layout when implementing horizontal diffusion using GPU RMA (cf. Section 3.2.3).

The MPI CUDA implementation is built of several CUDA kernels with MPI communication in between. As for the GPU RMA library, we used OpenMPI which is a CUDA-aware MPI implementation that accepts pointers to GPU device memory and employs GPUDirect RDMA or memory transfers to the host internally. This avoids manual memory transfers between the host and the GPU and is the recommended way of using MPI with CUDA. Compared to the GPU RMA implementation, only the halo rows residing on different GPU are explicitly exchanged in the code using MPI message passing. We use a CUDA kernel for every stencil type and the same memory layout as in the GPU RMA implementation. In GPU RMA, notified PUTs let the neighboring rank know that the stencil computation has finished and the dependent stencil can now be executed. In MPI CUDA, the stencil computations are separated by individual CUDA kernel launches. This guarantees process and memory synchronization.

Recall that blocks of a CUDA kernel in MPI CUDA correspond to ranks on the same device in GPU RMA. We note that the GPU RMA implementation has more overhead for on-device communication: All ranks on the same GPU send notifications to each other whereas in MPI CUDA, a single kernel launch achieves the same and no synchronization between CUDA blocks occurs. On the other hand, a kernel launch enforces device-wide synchronization whereas GPU RMA is more fine-grained. Finally, when exchanging data with remote GPUs in MPI CUDA, the host needs to synchronize with the GPU to ensure the MPI messages don't exchange stale data. In the GPU RMA library, the host issues MPI communication calls concurrently to kernel execution and allows more parallelism.

For the actual stencil computation, GPU RMA and MPI CUDA share the same code. Only infrastructure setup, communication and synchronization is handled differently by two implementations. After having received the notifications, the GPU RMA implementation calls the stencil function. In MPI CUDA, a wrapper kernel is launched that calls the same stencil function. This allows for a fair comparison of the two implementations. In our implementation, a GPU RMA rank or a MPI CUDA block always has 256 threads organized as a rectangular block to process the grid and we use double precision computations. There are several ways to organize 256 threads. We provide implementations for two-dimensional 16x16, 8x32, and 4x64 thread configurations. The thread configuration height defines the minimum number of rows $r$ that must be assigned to the worker. The minimum number of rows per worker multiplied by the number of workers provides a lower limit on the minimal overall grid height. We know from our horizontal diffusion model that a smaller grid height yields a higher communication ratio which is desirable for our system evaluation.

We close this section by giving a summary of the different implementations and configurations provided. We implemented the iterative horizontal diffusion algorithm that is made up of dependent stencil kernels. Three variations were implemented. The straightforward HD variant computes horizontal diffusion by exchanging halos and computing the dependent stencils alternatively. The HD-OVERLAP and HD-NOOVERLAP variants perform two independent horizontal diffusion computations. HD-NOOVERLAP fol-
4.3 Case Study: Horizontal Diffusion

Horizontal diffusion follows the HD pattern. HD-OVERLAP overlaps the two computations by executing stencils of one run while the other run exchanges halos and vice versa. This is expected to yield higher utilization and performance. HD-NOOVERLAP and HD-OVERLAP can be directly compared with each other. The three variations were implemented once using the GPU RMA library and once using MPI and CUDA directly. Both implementations share the same code to perform the actual stencil computations. The stencil is computed by organizing 256 threads in a rectangle and moving that band over the assigned grid band. To evaluate different computation and communication tradeoffs, the stencil code is provided in the three configurations 16x16, 8x32 and 4x64 threads.

The implementations are available in the apps/ directory of the GPU RMA source tree. The executables follow a naming scheme that tells the configuration implemented.

\[
\text{Binary Name} = \text{hd}_{\{\text{variant}\}}_{\{\text{implementation}\}}_{\{\text{thread-config}\}}$

\[
\text{variant} \in \{\emptyset, \text{overlap}, \text{nooverlap}\}
\]

\[
\text{implementation} \in \{\text{gpurma}, \text{mpi_cups}\}
\]

\[
\text{thread-config} \in \{16, 8, 4\}
\]

4.3.4 Results

We complete our horizontal diffusion case study with the performance analysis of the variations implemented. We contrast the GPU RMA performance with the MPI CUDA reference to assess the impact of our novel programming model. We further validate the stencil runtime against the horizontal diffusion model based on the GPU RMA performance model.

We conducted the experiments on our development system at CSCS under the same configuration as the system latency analysis in Section 4.1. Recall that our development system consists of two nodes with a Nvidia Tesla K80 installed on each node. We report results for a one GPU setup (local), and a two GPU setup (remote) with communication between the two nodes. The K80 provides two logical GPUs as peers that would allow to analyze intra-node communication as well. We have seen in the system latency analysis that the GPU peer-to-peer communication performance is far from what we would expect independent of our implementation. As little insight is gained from this configuration we omit the discussion and focus on device-local and inter-node communication performance. Our plots show the execution time of a single horizontal diffusion iteration. The reported values are the mean of a minimum of 1000 iterations and the 95% confidence interval lies within 5% of the reported mean.

**Hardware Parallelism** The level of parallelism on the GPU is given by the number of blocks executing concurrently and the number of threads per block. The register and shared memory usage per block determines the maximal number of blocks that can run concurrently on a given GPU. Table 4.4 summarizes the GPU resource usage of our
4 Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>Max. Blocks</th>
<th>Registers</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU RMA Baseline</td>
<td>104</td>
<td>28</td>
<td>696B</td>
</tr>
<tr>
<td>GPU RMA HD</td>
<td>52</td>
<td>112</td>
<td>952B</td>
</tr>
<tr>
<td>GPU RMA HD-0OVERLAP</td>
<td>39</td>
<td>134</td>
<td>952B</td>
</tr>
<tr>
<td>GPU RMA HD-NOOVERLAP</td>
<td>39</td>
<td>139</td>
<td>952B</td>
</tr>
<tr>
<td>MPI CUDA lap</td>
<td>34</td>
<td>0B</td>
<td></td>
</tr>
<tr>
<td>MPI CUDA ftx</td>
<td>32</td>
<td>0B</td>
<td></td>
</tr>
<tr>
<td>MPI CUDA out</td>
<td>34</td>
<td>0B</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: GPU Resource Usage

implementation as reported by the Nvidia Profiler². Every blocks has 256 threads.

The GPU RMA Baseline is a minimal program that initializes GPU RMA and performs no further computation or communication. It represents the minimal resource usage of our GPU RMA library. In the horizontal diffusion implementations we see shared memory usage increasing due to additional state for RMA communication buffers. MPI CUDA performs its computation in distinct stencil kernels launched from the host (cf. Section 4.3.3). The same kernels apply for all implementations and show much smaller resource usage as they perform one particular stencil only. The GPU RMA implementation combines the functionality of all three MPI CUDA kernels in addition to the GPU RMA library. While the lower register usage would allow for more blocks to be scheduled onto the GPU, experiments show the runtime performance does not increase. We attribute this to the limited memory bandwidth on the device.

Recall from our scheduling discussion in Section 3.1.1 that GPU RMA must never launch more blocks on the GPU than run concurrently. The MPI CUDA implementation is not restricted in the number of blocks. In our numbers reported, MPI CUDA was always run with the same number of blocks than GPU RMA. Experiments show launching MPI CUDA with more blocks does not benefit performance.

**HD: Local Performance** The HD local experiment executes horizontal diffusion on a single GPU to assesses the overhead of using GPU RMA instead of plain CUDA. As a result, the MPI CUDA implementation doesn’t perform any MPI communication but is restricted to the CUDA computation.³ As in the GPU RMA implementation, the horizontal grid bands are contiguous in memory. MPI CUDA achieves memory coherency and process synchronization using multiple kernel launches for every stencil computation. GPU RMA uses notified access.

We chose a fixed height grid for our experiments and report the execution time of a single horizontal diffusion iteration under a linearly increasing grid width. Our horizontal diffusion model analysis in Section 4.3.2 has shown an increasing grid width influences

²$\text{nvprof --print-gpu-trace ./program}$

³We use the same MPI-based implementation as for multi-GPU communication yet the runtime configuration yields no communication requirement.
4.3 Case Study: Horizontal Diffusion

Figure 4.3: HD Local Performance

the computation from being communication- to being memory-bound. Figure 4.3 reports the numbers for our single-GPU experiment showing the iteration runtime on the vertical axis and the grid width on the horizontal axis. The proportion of communication compared to computation according to our horizontal diffusion model is indicated as a background overlay and refers to the vertical scale at the right hand side of the plot. The displayed runtimes refer to the scale on the left hand side of the plot. We report two results for GPU RMA. GPU RMA names the default build. Because of overlapping RMA windows in our horizontal diffusion memory layout, the GPU RMA no-copy optimization applies (cf. Device-Local Transfers in Section 3.2.3) and the library doesn’t perform copies but sends notifications only. While this is a reasonable choice for library performance, it doesn’t allow to validate the GPU RMA performance.\footnote{Note that this is a property of our horizontal diffusion implementation and doesn’t hold for device-local communication in general.} To validate the GPU RMA performance model, we measured the extra build GPU RMA (no nocopy) with the no-copy optimization disabled (cf. Section 3.3). For the performance model, we report the overall execution time as well as the communication-only time.

The plot shows the MPI CUDA, the two GPU RMA and the theoretical model runtimes grow linearly with the problem size as expected. For large inputs with a comparably smaller communication ratio, GPU RMA performs best, followed by MPI CUDA and GPU RMA (no nocopy) with the model en par with MPI CUDA. At high communication ratio with small input sizes, the model performs best, followed by MPI CUDA and GPU RMA. We evaluate the model and the runtime performance separately.

Model Validation. For large inputs a with comparably low communication ratio the model lies between the GPU RMA and GPU RMA (no nocopy). From a modelling perspective we would expect the model to align with GPU RMA (no nocopy). As expected, the model performs slower than standard GPU RMA with the no-copy optimization enabled. For small input grids the model is faster than GPU RMA which we attribute to a
simplification in our model. We chose the input grid with the minimal height possible to maximize the communication ratio resulting in just $r = 4$ rows assigned to each rank for processing. The model neglects the additional two halo rows that are fetched from memory and underestimates the memory load in this particular configuration predicting better performance. We attribute additional difference to the overhead of function calls and notification polling being more apparent for small input sizes.

**Performance Comparison.** GPU RMA and MPI CUDA show comparable performance. GPU RMA (no no-copy) performs worse which is expected as it performs unnecessary copies on aliased halo rows GPU RMA and MPI CUDA avoid. The GPU RMA performance is surprising. Given the large latencies in our GPU RMA performance model we expected it to perform worse. While it avoids copies the notifications are still sent over the host and latency remains relevant. Up to a grid width of 512 elements MPI CUDA is faster while GPU RMA performs better afterwards. We conjecture GPU RMA benefits from the more fine-grained synchronization we introduced in our programming model. It avoids the global barrier amongst all blocks at the end of each stencil computation MPI CUDA implicitly performs by kernel launches. The GPU RMA ranks are independent and while some wait for notifications other may proceed having more GPU resources such as memory bandwidth at their disposal.

**HD: Remote Performance** With the HD remote experiment we evaluate horizontal diffusion computation in a two-GPU setup across two nodes where GPU RMA and MPI CUDA must communicate over InfiniBand. Figure 4.4 displays the measured iteration times as in the HD local experiment. We don’t report GPU RMA (no no-copy) anymore as in our test configuration inter-node communication time is always dominating device-local communication and the no-copy optimization has no influence on the measured performance. The OpenMPI implementation used by MPI CUDA and the GPU RMA library performs a protocol switch from using GPUDirect RDMA to staging transfer buffers through the host for GPU memory transfers (cf. Section 4.2). The protocol switch yielding higher bandwidth is indicated by a vertical bar in the plot and clearly visible by a flattening of the performance curves for both MPI CUDA and GPU RMA.

**Model Validation.** We observe an overall higher communication ratio than in the HD local experiment as communication over InfiniBand takes more time. The protocol switch is well reflected in GPU RMA and the model, in particular for the model communication-only time. The model and measurements follow the same trend.

**Performance Comparison.** We observe a similar pattern to the HD local case. MPI CUDA performs better for small grids while GPU RMA dominates for large grids. In HD local divergence started at a grid width of 512 elements. For HD remote they show comparable performance remote up to a grid width of 1024 elements.

**HD-OVERLAP Remote Performance** HD-OVERLAP performs two runs of horizontal diffusion alternating stencil computation between the two enabling computation and communication overlap. This experiment emphasizes the GPU RMA communication performance as either communication or computation dominates iteration runtime. The
4.3 Case Study: Horizontal Diffusion

Figure 4.4: HD Remote Performance

results are shown in Figure 4.5. Note that while iteration time is fully communication bound, indicated by a 100% RMA ratio in the background, the modelled overall and transfer time are identical in the plot.

Model Validation. Before the OpenMPI protocol switch reflected in the horizontal diffusion model, the model predicts a higher iteration time than we measured. This is the first time measured performance is better for small grids than the predicted time. After the protocol switch the model predicts faster execution. We note that for HD-OVERLAP we compute the iteration times differently than for HD (cf. Section 4.3.2). For large grids the model predicts a up to 30% faster iteration time than we observe. For these inputs iteration time is fully memory-bound as indicated by the model communication ratio in the background. We conclude that both MPI CUDA and GPU RMA cannot achieve the memory bandwidth our model assumes but note that our model neglects the fact that for perfect computation and communication overlap communication takes a share of memory bandwidth away from the computational task.

Performance Comparison. Again GPU RMA and MPI CUDA performance is close for small inputs with MPI CUDA. For the first time we can clearly see the OpenMPI protocol switch in the MPI CUDA performance as well but for GPU RMA it is less apparent. From our performance model we know the iteration time is fully communication bound explaining why we now see this for MPI CUDA as well. While the well-known pattern of GPU RMA dominating for large grids prevails we gain an additional insight. The previous HD experiments had mixed communication–computation runtimes. In HD-OVERLAP communication is fully dominant for small input grids. We observe GPU RMA dominates MPI CUDA even in communication-bound scenarios starting from a grid width of 640 elements.
HD-OVERLAP and HD-N0OVERLAP Comparison  
Comparing HD-OVERLAP and HD-N0OVERLAP we are interested in communication-dominated results only as performance matches once computation dominates overall iteration runtime. The performance of the two variations for MPI CUDA and GPU RMA are shown in Figure 4.6. We report the modelled communication percentage for both models in the background but omit the model performances for clarity.

We observe no difference for the smallest inputs which is expected as computation time is low and both variations have to communicate the same amount of data. We cut the large input sizes as they are fully memory-bound and no difference occurs either. There is only a limited range for medium sized inputs from a grid width of 256 to 640 where a difference between HD-OVERLAP and HD-N0OVERLAP is observable for MPI CUDA. The difference is still considerable for GPU RMA but will gradually dissolve as well.

For GPU RMA HD-N0OVERLAP is always faster. This is not the case for MPI CUDA. Initially HD-OVERLAP is faster but aligns with HD-N0OVERLAP when computation starts to dominate. We notice HD-OVERLAP is always slightly slower from a grid width of 640 elements on. The counterintuitive result is explainable by the implementation. Before starting communication, the host has to synchronize with the GPU to ensure memory coherency. The HD-N0OVERLAP performs the stencil computation for both horizontal diffusion runs and returns to the host. The HD-OVERLAP variation computes one stencil, then returns to the host to communicate the stencil result and goes back to the GPU to compute the stencil from the other horizontal diffusion run. While this approach allows for computation and communication overlap, it involves twice as many host-GPU synchronizations than in HD-N0OVERLAP. Judging from the measurement, the effort is not rewarded for memory-bound inputs.
4.3 Case Study: Horizontal Diffusion

Figure 4.6: HD-OVERLAP and HD-NOOVERLAP Comparison

Results Summary  In the horizontal diffusion case study we evaluated GPU RMA in device-local and inter-node communication scenarios. We contrasted the results with the standard approach of programming a GPU cluster with MPI and CUDA and validated the theoretical performance model.

In general, we saw the model and the two implementation approaches result in similar runtime performance for the HD experiments. For HD-OVERLAP the model underestimates the iteration time for memory-bound inputs and also for communication-bound inputs where the underlying OpenMPI implementation uses explicit transfers over host memory rather than using GPUDirect RDMA. A precise understanding of the model estimates will require further system analysis. We note that the GPU RMA model parameters were derived from isolated microbenchmarks and had no contention on GPU resources. For instance, on-device copy achieves a bandwidth of $B_{RMA} = 4.5$ GB/s for a single block (cf. Table 4.2). In horizontal diffusion the aggregate memory bandwidth $B_M = 140$ GB/s is shared amongst $R_l = 52$ blocks for HD allowing a per-block bandwidth of $B_M R = \frac{B_M}{R_l} = 2.7$ GB/s (cf Table 4.3).

Comparing our novel GPU RMA model to the state of the art MPI and CUDA approach we observe a better GPU RMA performance for both memory- and communication-bound inputs except for very small grids. We conjecture the more fine-grained synchronization amongst the GPU RMA ranks allows for more parallelism increasing GPU utilization while avoiding kernel-launch induced global barriers. For small inputs the standard approach is preferable. We conjecture at that problem size the more fine-grained synchronization we employed degrades overall system performance. The no-copy optimization showed to be a valuable feature of GPU RMA automatically avoiding unnecessary copies and if the programmer choses a corresponding memory layout. We note that this is an important addition to GPU RMA performance next to the improved GPU utilization.
4 Evaluation

Our evaluation provided a first impression of the novel GPU RMA programming model providing cluster abstraction and introducing one-sided inter- and intra-GPU communication controlled directly from the GPU. While many components require further analysis to fully understand their system performance impact and we are curious to the system's scaling to multiple nodes our preliminary results show look promising.

**Scalability** During our work we were granted access to the Nvidia PSG cluster for evaluating GPU RMA. PSG is a shared cluster for demonstration, benchmarking and testing of Nvidia GPUs. During our final evaluation phase the cluster showed high utilization with the computing resources constantly booked. We managed to run a few initial tests and the results differed from what we had observed on greina. While GPU RMA performance was comparable, MPI CUDA performance degraded by a factor of 2.5. We could not reproduce these results even for a two-node configuration on our development system. As the high utilization of the PSG cluster prevented a systematic analysis of the cause we cannot report reliable scalability results.
Related Work

We started the discussion of the GPU RMA programming model based on the popular MPI and CUDA approach and developed our novel model inspired by the ideas of notified access and GPU hardware support for RDMA [2, 7, 8, 17, 24, 26]. We present related work to our results in four categories. The gradual entry of GPUs into high performance computing has raised the question of an appropriate representation of this massively parallel computing device amongst conventional CPUs representing the first category of related work. The introduction of RDMA hardware support on the GPU has gained the attention of cloud providers with service-based infrastructures we present in the second category. We follow with the evaluation of inter-GPU communication performance and close with GPU programming techniques relevant for the implementation of GPU RMA.

5.1 GPU Cluster Programming

A series of previous research investigated programming models to approach GPU-enabled compute clusters. We provide a chronological overview.

DCGN (2009)  Distributed Computing on GPU Networks (DCGN, [40]) is a MPI-inspired message passing API for heterogeneous CPU-GPU computing networks. While the GPU is a powerful data-parallel compute device, it is installed as a CPU coprocessor. DCGN introduces the GPU as a first-class computational resource that can source and sink communication independent from the CPU. The GPU is represented by a user-defined number of slots that can directly be addressed. From the description we assume slots are represented as CUDA kernels but that is not stated explicitly. The model introduces the notion of a kernel which is a self-contained computational unit that is deployed either
on the CPU or GPU. The kernels must be written specifically for the target architecture yet the library fully abstracts launching kernels on either the CPU or GPU and handles all communication. The library employs three different types of processes on the host, one for CPU kernel execution, one for GPU kernel execution, and one for communication. While the host controlling the GPU is removed from the programming model it still manages the GPU and all host and device communication is handled by the CUDA runtime as none of the GPUDirect technologies were available at that time. Their work also discusses differences between CPU threads and GPU threads and mentions the block-scheduling deadlock problem.

In a follow-up paper [41] the authors enumerate options to map MPI ranks to GPUs on a philosophical level unrelated to the actual hardware mapping or an implementation of their ideas. Additionally, they introduce the idea of kernel-launching MPI calls. They suggest an extra communicator for MPI-enabled accelerator devices. New MPI calls allow to launch kernels on ranks in this communicator and these ranks run on accelerator devices such as the GPU and may source and sink communication.

**rCUDA (2009)** Remote CUDA (rCUDA, [5]) is a GPU virtualization infrastructure that operates a transparent layer on top of CUDA. It makes all GPUs on a compute cluster available on the local node. All GPUs on the cluster appear as if they are installed locally and CUDA library calls are forwarded to the responsible host. The model hides networking and communication complexities from the programmer while preserving the standard view presented by CUDA.

**FLAT (2012)** FLAT [18] is a GPU programming framework to provide embedded MPI on the device. The application can perform MPI calls from the device. Before execution the application code is processed by the FLAT compiler that splits execution into multiple CUDA kernels at communication points. At MPI calls, the kernel returns, the host handles all communication and launches the next kernel as a continuation. While this is convenient for the application programmer, the resulting implementation is inherently serial and prevents full GPU utilization.

**GGAS (2013)** Oden and Fröning introduce a Global GPU Address Space (GGAS, [30]) where GPUs on a network have access to all the memory on any of the remote GPUs completely bypassing the CPU. They make use of the novel GPUDirect RDMA support for Nvidia GPUs that allows to map device memory on the PCI-E bus and employ a custom-built FPGA-based network adaptor that enables the GPU to initiate communication without host support. Their model and implementation allows zero-copy communication amongst GPU devices and provides bulk synchronization and communication functionality.
5.2 GPU-controlled Communication

The introduction of GPUDirect RDMA enabling direct access to GPU device memory from any PCI-E device in 2013 and its adoption to InfiniBand by Mellanox in 2014 has opened a new range of communication options. In the last section we reviewed approaches to integrate the GPU into a cluster programming model and the GGAS results already leverages GPUDirect RDMA. We now present work that ports existing networking APIs to the GPU that allow to integrate the GPU as an accelerator device into existing production infrastructure. The presented adoptions introduce the GPU communication targets at block granularity as we decided for GPU RMA as well. While their goals were different than ours, they faced many similar implementation challenges.

GPUNet (2014)  

GPUNet [13] investigates networking abstractions for GPU programs and implements the standard rsocket interface on GPU with focus on service-oriented architecture allowing the GPU to function as a network server processing incoming requests. The network adaptor is configured to read and write directly from device memory using GPUDirect RDMA. While current Nvidia GPUs can map host memory into their virtual address space, memory-mapped I/O regions that are backed by devices rather than RAM are not supported, though we will relativize this when reviewing the other 2014 paper in the next paragraph. The host and device communicate over ring buffers to trigger communication from the GPU. While parts of their discussion covers topics relevant to service infrastructure such as I/O balancing, we draw inspiration from similar implementation challenges addressing communication.

Network communication cannot be fully controlled from the GPU and requires the host. GPUNet mediates between device and host using ring buffers and optimizes memory access by mapping device memory into the host’s virtual address space using GPUDirect RDMA. While they employ their custom kernel module integrating further functionality, we decided for a similar design using Nvidia’s gdrcopy kernel module. They also chose to represent communication targets as CUDA blocks. Finally, they reported bad communication bandwidth over PCI-E due to the Intel chipset bug that is mentioned by other scientific publications as well.

InfiniBand Verbs on GPU (2014)  

While controlling a network adaptor from the GPU is not officially possible, Oden et al. [31] proved an InfiniBand network device can be controlled from a GPU alone without CPU interaction after an initial setup phase and they implement the InfiniBand Verbs API fully on the GPU at block granularity.

Their solution involves two steps. First, they provide their own patch for the Mellanox IB driver not relying on the official kernel modules to enable RDMA. The second step targets the memory-mapped I/O registers of the network adaptor that trigger communication. While the CUDA runtime allows to map host memory into the GPU’s virtual address space, it prevents the mapping of memory-mapped device addresses being the reason network communication still requires host interaction. Oden et al. circumvent this limitation by intercepting system calls between the CUDA runtime and the device
driver and manipulating the arguments. Their work shows this is sufficient to map peer
PCI-E device memory into the GPU’s virtual address space and let the GPU control the
network adaptor fully on its own.

While technically the GPU can now control network transfers, their performance results
are worse than using a host-initiated approach. The authors provide three reasons.
First, InfiniBand Queue Pair operations are single-threaded operations that the GPU is
not optimized to perform and the CPU excels at. Second, controlling the network adap-
tor involves several PCI-E bus transactions that are non-coalesced in their implementa-
tion. Finally, the GPU architecture is not optimized for low-latency memory access. We
can confirm this problem from our own results.

Their solution involves many changes to the environment, making it ill-suited towards
cluster installations. The authors note they could not test their work on a large cluster
because of this.

5.3 Inter-GPU Communication

Inter-GPU communication was studied in depth particularly for integrating GPUs into
existing MPI applications and optimizing data transfers to fully leverage the computa-
tional resources of GPU-enabled compute clusters.

We found the publications of the MVAPICH group of integrating the GPU into their MPI
implementation pioneering CUDA-aware MPI particularly helpful as it revealed many
important details about the sparsely documented GPUDirect family of technologies and
clarified on its working and usage. We note that because GPUDirect evolved gradually it
is not always apparent from reading the publications what particular variant of GPUDi-
rect they are referring to and referring to the publication date helps determining what
GPUDirect technology was available at that time.

Their first publication [45] discuss design challenges of enabling GPU device buffers
in the MPI library and discusses pipelined data flow techniques to lower latency. Most
of the initial investigations are obsolete nowadays thanks to the introduction of the
GPUDirect technologies. Integration of GPUDirect Peer-to-Peer is presented in [37] and
adoption of GPUDirect RDMA in [36], covering technical details such as how to combine
InfiniBand programming with GPUDirect RDMA or the reported chipset issues. In [44]
they elaborate protocols for further MPI functionality such as data type or strided data
transfer support on the GPU and stress the difficulty of effective latency hiding from
the application programmer. While these features find no application in our GPU RMA
implementation this work may prove helpful when introducing these features.

The publications from MVAPICH are complemented by Pena and Alam [35] investigat-
ing GPU cluster communication performance, data flow pipeline design and resulting
bottlenecks to assess GPU computation scalability. Effects of server architecture design
on GPU communication latency and bandwidth are presented by the CirraScale com-
pany [6] and an in-depth performance benchmark of GPUDirect RDMA under various
server architectures are presented in the Nvidia ParallelForall blog [38].
5.4 Single-GPU Programming

We have introduced the inter-block barrier synchronization of Xiao and Feng [46] we employed in GPU RMA in our implementation chapter in Section 3.1.1. In a more recent publication Scogland and Feng [39] discuss the design of fast lock-free contention-heavy queues for use on GPUs. Their results may prove useful when investigating a different notification handling architecture in GPU RMA with multi-writer queues. To this day block scheduling on the GPU cannot be controlled. Tanasic et al. [42] have proposed two hardware preemption techniques for Nvidia GPUs compatible with the current CUDA programming model yet none of them has been implemented so far and GPU scheduling remains a challenge for alternative GPU programming models.
Conclusions

In this master thesis we addressed the programming of GPU-enabled compute clusters. The massively parallel computational resources of the graphics processing unit has turned it into a popular accelerator device for general purpose computing that has found wide adoption in today's supercomputers and raised the question of their adequate representation in GPU cluster programming models. While the combination of established communication models such as message passing with a single-GPU programming model such as CUDA allowed to improve cluster performance over conventional systems, they prevent the full exploitation of cluster resources. In this master thesis we have proposed, implemented, and evaluated the novel GPU cluster programming model GPU RMA and draw conclusions from our work.

We strived to improve GPU cluster programming in three key areas. GPU RMA shall provide a uniform view of cluster resources that hides complexities of the underlying hardware architecture and topology while allowing efficient implementations. GPU RMA shall allow improved cluster utilization by introducing more fine-grained synchronization primitives into the programming model and enable the GPU to source and sink communication in the network. Finally, GPU RMA shall leverage the newly introduced RDMA support for GPUs and employ a one-sided communication model.

GPU RMA exposes the GPU cluster as a group of ranks executing on GPUs that may communicate with one-sided notified access. Our model removed the CPU as a mere communication mediator between the networking device and the GPU and presents a flat hierarchy of interacting ranks. While this view abstracts the real nature of the underlying topology, the application programmer may not be completely oblivious of the two level hierarchy imposed by GPUs installed on compute nodes. Initial setup on the host still requires a two level partitioning of the computational tasks first between nodes and second between ranks on the local GPU. A careful memory layout must be chosen to avoid unnecessary data movement for communication amongst ranks on the
same GPU and allow the GPU RMA no-copy optimization to achieve results competitive to the state of the art. We note that hiding the real nature of physical memory is a challenging design question on its own. Reintroducing the CPU as a computation and communication peer might mitigate the situation. We note that the CPU is already a key component in the GPU RMA library implementation and its role is deemed to be twofold once reintroduced, supporting the GPU and representing a rank at the same time. These limitations are primarily due to the current implementation of our GPU RMA library. GPU RMA provides a fundamentally different programming model than combining MPI and CUDA, removing bulk-synchronous kernel launches and providing a richer notified access communication model controlled from the GPU and amongst GPU blocks. Our programming model allows a better mapping to the cluster resources apparent in application performance.

GPU RMA strives to improve cluster utilization by introducing more fine-grained synchronization amongst GPUs as well as within the local GPU. First of all, GPU RMA achieved a correct implementation of such synchronization and communication primitives by leveraging previous work in inter-block communication on GPUs and by introducing the concept of a block manager on the host that allows blocks to communicate independent of others in a CUDA kernel. Our preliminary results from the horizontal diffusion case study are encouraging. In spite of large communication latencies the GPU RMA based implementations showed better execution time for memory-bound and for most communication-bound scenarios. We are aware that the case study only provides a first impression of GPU RMA performance and its characteristics for different computational and communicational problems as well as its scalability remains to be examined yet our results encourage further study of GPU RMA.

We report mixed results from employing the recently introduced RDMA support for Nvidia GPUs. The technology is still in its early state and suffers from limitations in current hardware implementations such as peer to peer communication bandwidth between the GPU and NIC over PCI-E. In addition, server architecture needs to adopt in order to avoid unnecessary forwarding of traffic over multiple interconnects as we experienced ourselves on our development machines and as other HPC providers have reported. We measured worse communication latency on our machine than were reported otherwise. The larger latency could be due to either a deficiency of the machine or induced by the MPI implementation. Performing direct GPU to GPU transfers initiated from the host incurs a higher latency than conventional host to host communication. Our experiments show the chosen notified access protocol is not the limiting factor but hardware support for RDMA to GPU memory needs to mature.

Next to the GPU RMA programming model we summarize insights from our implementation efforts. The key strength of the GPU is at the same time its strongest weakness. The GPU achieves massive parallelism by thousands of concurrent threads scheduled with fast context switching on streaming multiprocessors. The oversubscription of GPU resources allows hiding of memory latency while the hardware-accelerated context switching enables overall progress. This powerful execution model fails to deliver performance once serial control flow based operations with dependent memory accesses dominates GPU execution in contrast to massively data-parallel computations. In our system latency analysis we measured a $0.84\,\mu s$ roundtrip latency for memory access be-
tween two GPU blocks compared to $1.41 \mu s$ for a ping-pong over the PCI-E bus. We note that great care was required in our work to avoid global memory accesses in favor of GPU performance resulting in library state partitioning. Given the high memory access time for multiple operations compared to fetching a single result over PCI-E we ask whether predominantly control flow based computation should actually be performed on the GPU or whether a careful CPU-GPU co-design will allow higher system performance. This architectural consequence also raises the question whether the GPU can actually be considered a first-class citizen in the context of high performance computing or remain a respectable and highly-powerful co-processor for data-parallel computation.

We draw a positive conclusion from this master thesis. With GPU RMA we introduced a novel GPU cluster programming model removing the bulk-synchronous nature of conventional CUDA programming and exposing the GPU as a first-class communication target. Our model provides a rich communication semantics enabling fine-grained synchronization amongst and within graphics processing units. Despite deficiencies in the recently introduced RDMA support for GPUs our evaluation and case study revealed better application performance employing GPU RMA instead of the conventional MPI and CUDA. Our results encourage further adoption of GPU RMA to different computational problems and study the system performance and scalability in more detail.
Bibliography


Bibliography


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

Bibliography


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):

GPU Remote Memory Access Programming

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

Name(s):

First name(s):

Bär

Jeremia

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

Zürich, 27. Oct. 2015

Signature(s)

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