Direct Communication Methods for Distributed GPUs

INAUGURAL – DISSERTATION
zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht–Karls–Universität
Heidelberg

von
Lena Oden
Diplom Ingenieurin der Elektrotechnik
aus Moers

Kaiserslautern, 2014

Betreuer: Professor Dr. Ulrich Brüning, Universität Heidelberg

Tag der mündlichen Prüfung:______________________

Abstract

Today, GPUs and other parallel accelerators are widely used in high performance computing, due to their high computational power and high performance per watt. Still, one of the main bottlenecks of GPU-accelerated cluster computing is the data transfer between distributed GPUs. This not only affects performance, but also power consumption. Often, a data transfer between two distributed GPUs even requires intermediate copies in host memory. This overhead penalizes small data movements and synchronization operations.

In this work, different communication methods for distributed GPUs are implemented and evaluated. First, a new technique, called GPUDirect RDMA, is implemented for the Extoll device and evaluated. The performance results show that this technique brings performance benefits for small- and mediums-sized data transfers, but for larger transfer sizes, a staged protocol is preferable since the PCIe-bus does not well support peer-to-peer data transfers.

In the next step, GPUs are integrated to the one-sided communication library GPI-2. Since this interface was designed for heterogeneous memory structures, it allows an easy integration of GPUs. The performance results show that using one-sided communication for GPUs brings some performance benefits compared to two-sided communication which is the current state-of-the-art. However, using GPI-2 for communication still requires a host thread to control GPU-related communication, although the data is transferred directly between the GPUs without any host copies. Therefore, the subsequent part of the work analyze GPU-controlled communication.

First, a put/get communication interface, based on Infiniband verbs, for the GPU is implemented. This interface enables the GPU to independently source and synchronize communication requests without any involvements of the CPU. However, the Infiniband verbs protocol adds a lot of sequential overhead to the communication, so the performance of GPU-controlled put/get communication is far behind the performance of CPU-controlled put/get communication.

Another problem is intra-GPU synchronization, since GPU blocks are non-preemptive. The use of communication requests within a GPU can easily result in a deadlock. Dynamic parallelism solves this problem. Although the performance of applications using GPU-controlled communication is still slightly worse than the performance of hybrid applications, the performance per watt increases, since the CPU can be relieved from the communication work.

As a communication model that is more in line with the massive parallelism of GPUs, the performance of a hardware-supported global address space for GPUs is evaluated. This global address space allows communication with simple load and store instructions which can be performed by multiple threads in parallel. With this method, the latency

for a GPU-to-GPU data transfer can be reduced to 3 μs, using an FPGA. The results show that a global address space is best for applications that require small, non-blocking, and irregular data transfers. However, the main bottleneck of this method is that is does not allow overlapping of communication and computation which is the case for put/get communication. However, by using GPU optimized communication models, depending on the application, between 10 and 50% better energy efficiency can be reached than by using a hybrid model with CPU-controlled communication.

Zusammenfassung

Heutzutage sind Grafikkarten und andere Beschleuniger weit verbreitet im Bereich des Hochleistungsrechnens, da sie eine hohe Leistungsfähigkeit und einen vergleichsweise geringen Energie Verbrauch versprechen. Trotz dieser weiten Verbreitung ist die Kommunikation und der Datentransfer zwischen verteilten Grafikkarten der größte Flaschenhals für Grafikkarten unterstütztes Rechnen. Das bedeute für viele Programme, die mit Grafikkarten beschleunigt werden, Einschränkungen, nicht nur was die Leistung betrifft, sondern auch den Energieverbrauch. Der Datentransfer zwischen zwei Grafikkarten benötigt häufig Kopien im Hauptspeicher, was die Leistung weiter einschränkt. Besonders die Übertragung von kleinen Datenmengen wird von diesen zwischen-Kopien beeinträchtig.

Neue Technologien wie GPUDirect RDMA ermöglichen einen direkten Datentransfer zwischen zwei Grafikarten, ohne Puffer im Hauptspeicher. Aus diesem Grund wird im ersten Teil dieser Arbeit die Unterstützung für diese Technik für den Netzwerkcontroller der Extoll Netzwerkarchitektur implementiert und evaluiert. Die Messergebnisse zeigen, dass diese Technologie zwar Leistungsgewinne für kleine und mittlere Nachrichtengrößen bringt, für größere Nachrichten jedoch ein gepufferter Transfer besser ist. Der Grund dafür ist die schlechte Unterstützung des PCIe-Busses für die Datenübertragung zwischen zwei externen Geräten.

In einem weiteren Schritt werden Graphikarten in die Kommunikationsbibliothek GPI-2 integriert, die vorwiegend einseitige Kommunikation verwendet. Da GPI-2 für heterogene Speicherarchitekturen entworfen wurde, können Grafikkarten einfach integriert werden. Der vergleich mit einer GPU unterstützenden MPI Variante, die aktuell stand der Technik ist, zeigt, dass auch Grafikkarten von einseitigen Kommunikationsmodellen profitieren können. Auch wenn die Daten ohne Puffer im Hauptspeicher übertragen werden können, so benötigen GPI-2 und MPI immer noch die CPU, um die Kommunikation zu starten und zu synchronisieren. Daher widmet sich die folgenden Teile der Arbeit GPU-kontrollierter Kommunikation.

Dazu wird zunächst eine put/get (schreibe/lese)-Kommunikationsschnittstelle für Grafikkarten, die auf Infiniband Verbs basiert, implementiert. Diese Schnittstelle ermöglicht es der Grafikkarte unabhängig von der CPU eine Kommunikationsanfrage zu starten und zu synchronisieren. Leider benötig Infiniband sehr viele sequentielle Arbeitsschritte, für die Grafikkarten nicht optimiert sind. Dies führt zu einer deutlich schlechteren Leistung für die Grafikkarten kontrollieren Kommunikation im Vergleich zu CPU kontrollierter Kommunikation.

Ein weiteres Problem ist die interne Synchronisation der Threads auf einer Grafikkarte, da die Thread Blöcke auf einer Grafikkarte nicht unterbrechbar sind. Dies kann in verbindung mit Grafikkarten kontrollierter Kommunikation schnell zu einer Blockie-

rung der Threads untereinander führen. Dieses Problem kann mit dynamic parallelism, einer neuen Funktion von Nvidia Grafikkarten, gelöst werden. Die Leistung von Programmen, die GPU-Kontrollierte Kommunikation verwenden, ist immer noch schlechter als die Leistung von Programmen, bei denen die CPU die Kommunikation kontrolliert. Da aber bei der Verwendung von Grafikkarten kontrollierter Kommunikation die CPU entlastet werden kann, ist die Energie-Effizient deutlich besser.

Als eine Kommunikationsmethode, die besser mit dem massiv parallelen Model von Grafikkarten zusammenpasst, wird im letzten Teil der Arbeit ein Hardware-unterstützter globaler Grafikkarten Adressraum betrachtet. In diesem globalen Adressraum können mehrere Grafikkarten mit einfachen Lese- und Schreib- Instruktionen kommunizieren, die von mehreren Threads parallel ausgeführt werden können. Auf diese Art können kleine Nachrichten in etwa 3 μs von einer Grafikkarte auf eine andere übertragen werden. Diese Kommunikationsmethode eignet sich am besten für Programme, welche viele kleine und unregelmäßige Datentransfers benötigen. Der Nachteil dieser Methode ist, dass sie kein Überlappen von Kommunikation und Rechnern erlaubt, was jedoch mit put/get Kommunikation möglich ist. Die Verwendung von Kommunikationsmethoden, die für Grafikkarten optimiert sind, führt, abhängig von der Applikation, zu einer 10-50% besseren Energieeffizienz.

Contents

1. Introduction **1**
1.1. Objectives of this work . . . 3
1.2. Outline . . . 4

2. Background **5**
2.1. Parallel Systems . . . 6
2.1.1. Shared memory systems . . . 6
2.1.2. Communication and synchronization in distributed memory systems . . . 8
2.1.3. Collective communication functions . . . 12
2.1.4. Communication performance . . . 13
2.1.5. Communication Interfaces . . . 14
2.2. Modern Parallel Processors . . . 19
2.2.1. Multicore and Manycore architectures . . . 19
2.2.2. Architecture of a modern GPU . . . 20
2.2.3. Other manycore processors . . . 23
2.3. GPU Programming Models . . . 25
2.3.1. CUDA . . . 25
2.3.2. OpenCL . . . 28
2.3.3. Directive-based approaches . . . 29
2.4. Interconnection Networks and Network Interfaces . . . 29
2.4.1. Interconnection networks . . . 30
2.4.2. PCI-Express . . . 30
2.4.3. QPI and Hyper Transport . . . 31
2.4.4. Network Interfaces . . . 32
2.4.5. Remote Direct Memory Access . . . 33
2.4.6. Infiniband . . . 35
2.4.7. Extoll . . . 40
2.5. Communication between Distributed GPUs . . . 43

3. Direct Data Transfer between GPUs **47**
3.1. Inter- and Intra-Node Data Transfer . . . 47
3.2. Data Transfer Methods . . . 47
3.2.1. GPUDirect 1.0 . . . 49
3.2.2. GPUDirect peer-to-peer . . . 50
3.3. GPUDirect RDMA . . . 51

3.3.1. Nvidia GPUDirect Interface 52
3.3.2. Mellanox GPUDirect RDMA support 52
3.3.3. Host mapped GPUDirect RDMA support 54
3.4. Performance Results of GPUDirect RDMA 56
3.4.1. Latency and bandwidth 57
3.4.2. PCIe peer-to-peer performance 58
3.4.3. Intel Ivy Bridge 59
3.4.4. Inter I/O-hub data transfer 60
3.5. Summary 61

4. Host-controlled GPU-to-GPU Communication **63**
4.1. Related work 64
4.1.1. CUDA-aware MPI 64
4.1.2. GPUs in PGAS languages and libraries 65
4.2. GASPI-Standard 67
4.2.1. Shared memory segments in GASPI 67
4.2.2. One-sided communication 68
4.2.3. Passive communication 69
4.2.4. Collective Operations 70
4.2.5. Atomic operations 70
4.2.6. Weak synchronization 70
4.3. Integration of GPUs to the GASPI-specification 71
4.3.1. GPU memory segments in GASPI 72
4.3.2. Initialization 73
4.3.3. GPU memory segment creation 74
4.3.4. Remote write and read operations 74
4.3.5. Passive communication and atomic operations 75
4.3.6. Weak synchronization for GPU segments 75
4.3.7. Allreduce 75
4.3.8. Additional functions for GPUs 76
4.4. Performance results 79
4.4.1. Bandwidth 80
4.4.2. Latency 80
4.4.3. CPU-communication overhead 81
4.5. Application level performance 83
4.5.1. Synchronization between GPU and host 83
4.5.2. Stencil codes 85
4.6. Summary 88

5. GPU-Controlled Put/Get Communication **89**
5.1. Related Work 89
5.1.1. Communication libraries for the Intel Xeon Phi 90
5.1.2. Libraries for GPU computing 90
5.2. Sourcing communication requests to RDMA-capable hardware 91

5.2.1. Work processing on Infiniband 91
5.2.2. Work request generation for the Extoll RMA unit 93
5.2.3. Conclusion for GPU-controlled communication 94
5.3. GPU-controlled communication 95
5.4. Creating a communication environment on the GPU 96
5.4.1. Porting resources to the GPU 97
5.5. Creating an Infiniband communication environment on the GPU 98
5.5.1. Context Setup on the Host 99
5.5.2. Creating Infiniband elements for the GPU 100
5.5.3. Infiniband interface on the GPU 101
5.5.4. Micro-benchmark results for Infiniband 104
5.5.5. Analysis and optimization 109
5.6. Creating an RMA environment on the GPU 114
5.6.1. Setting up an RMA connection on host 114
5.6.2. Porting of an RMA environment to the GPU 115
5.6.3. Micro-benchmark results for the RMA unit 116
5.6.4. Performance counter analysis for the RMA 119
5.7. One-sided Communication Interface on the GPU 120
5.7.1. Communication endpoints 120
5.7.2. Block-save communication 121
5.7.3. Asynchronous one-sided communication 122
5.7.4. Queues 122
5.7.5. Remote Synchronization 122
5.8. Inter-block synchronization 123
5.8.1. In-kernel synchronization and communication 124
5.8.2. Stream Synchronization 126
5.8.3. Synchronization with dynamic parallelism 128
5.8.4. Himeno performance results 131
5.8.5. Energy efficiency 134
5.9. Summary 136

6. Global Address Space for GPUs 139
6.1. Related work 141
6.2. GPU memory coherence and consistency 142
6.2.1. Coherence 142
6.2.2. Consistency 142
6.3. Hardware support for distributed global address spaces 143
6.4. Extending the global address space for GPUs 145
6.4.1. Restrictions for the global memory size 147
6.5. The GGAS Software 147
6.5.1. GGAS setup 148
6.5.2. GGAS GPU API 149
6.6. Micro-benchmark performance 150
6.6.1. Latency 151

6.6.2. Bandwidth 152
6.6.3. Coalescing effects 154
6.7. GGAS barrier 155
6.7.1. Intra-GPU synchronization 155
6.7.2. Barrier performance 156
6.8. Allreduce and reduce using GGAS 157
6.8.1. Reduction with remote read operations 158
6.8.2. Reduction with remote write operations 158
6.8.3. Work sharing: data distribution over multiple GPUs 159
6.8.4. Performance results for the reduce and allreduce operation . . . 160
6.8.5. Comparison to MPI 162
6.9. Application-level performance 163
6.9.1. Stencil code 163
6.9.2. Global reduction benchmark 165
6.9.3. RandomAccess benchmark 166
6.10. Energy Efficiency 168
6.10.1. Energy efficiency of the Himeno benchmark 168
6.10.2. Energy efficiency of the global reduction benchmark 168
6.10.3. Energy efficiency of the RandomAccess benchmark 169
6.11. Summary 170

7. Conclusion and Future Outlook **171**

A. Power Measurement **175**

B. Acronyms **176**

List of Figures **178**

List of Tables **182**

Listings **183**

1. Introduction

In the past, it was very simple for programmers to reach higher performance with the next generation of computing systems. The performance of single core CPUs increased dramatically for two decades while the costs rapidly were reduced. The same software simply ran faster with the next generation of CPUs.

However, since 2003, this growth has slowed down, as heat-dissipation and energy consumption limit a further increase of the CPU clock speed. Only by introducing multiple cores were CPU vendors able to maintain Moore's law. This change of adjustment has an enormous impact on software developers [1] as code has to be rewritten to reach better performance on new generation CPUs, or in other words *the free lunch was over* [2].

While the performance of single threaded CPUs stagnated, the performance of GPUs increased dramatically in the last years, as shown in Figur 1.1. Therefore, it is no suprise that the high performance computing community rapidly adopted GPUs for their purposes, trying to satisfy more of the computational needs of their performance-hungry applications. This interest grew, especially since programming languages like *CUDA* [3], *OpenCL* [4], or directive-based approaches like *OpenACC* [5] make the features of GPUs better available for developers who are not familiar with classic graphical aspects. Today, GPUs are an integral part of high performance computing.

However, high performance is not the only reason for the increased interest in GPUs; their high energy efficiency, since power consumption becomes more and more important due to ecological, economical, and technical reasons, is another important aspect. In particular, the first 15 systems of the Green500 list, the list of the most energy efficient high performance computing systems, from June 2014, are all accelerated with NVIDIA Kepler K20 GPUs [6].

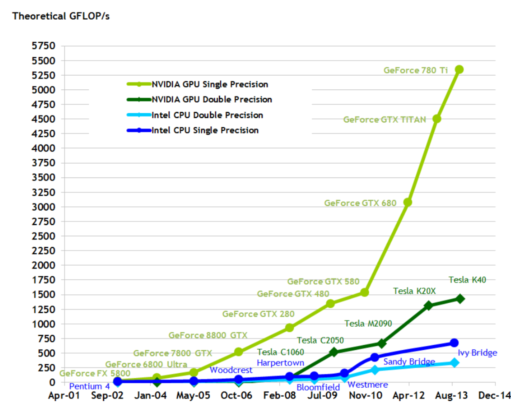


Figure 1.1.: Floating points per second, GPUs vs. Intel Processors [3]

For example, an Intel Xeon E5-2687W Processor (8 cores, 3.4 GHz, AVX) achieves about 216 GFLOPS at a thermal design power (TDP) of about 150 watts, resulting in 1.44 GFLOPS/watt. An NVIDIA K20 GPU is specified with a TDP of 250 Watt and a single precision peak performance of 3.52 TFLOPS resulting in 14.08 GFLOPS/watt.

Still, GPUs were primarily designed for graphical aspects. Therefore, throughput oriented applications can benefit most from GPUs. Furthermore, GPUs only excel if they can perform their calculations *in-core*. That means that they use their own, dedicated device memory. This device memory, however, is a very scarce resource. The latest GPUs, Nvidia's K40 Atlas GPU, provides 12 GB GDDR5 device memory, while the previous model the Nvidia K20 GPU, only provides around 6 GB device memory. Typically, GPUs are connected as peripheral devices through the PCIe-bus. The bandwidth of PCIe is fast (8 GB/s for $\times 16$ PCIe-2 and 15.75 GB/s for $\times 16$ PCIe 3.0) but it is still far behind the memory bandwidth of the device memory (up to 288 GB/s).

To satisfy the requirements of modern applications (for example, *big data*), GPUs are deployed in clusters and used in parallel to fulfill the requirements of memory-hungry applications. However, if GPUs are deployed in clusters, most applications require communication and data transfer among these GPUs. This is the main bottleneck of GPU accelerated high performance computing, with regard to performance and energy consumption [7, 8, 9].

The communication overhead is a potential bottleneck for all distributed memory systems, but gets reinforced for heterogeneous systems with accelerators. A closer look at the first ten systems of the top500 list from June 2014 shows that the accelerated computers attain 60-80% of their theoretical peak performance, whereas systems without accelerators attain 64-93% of the peak performance [10]. The main reason for this is the dedicated device memory of accelerators which is separated from the host memory.

Most communication libraries and frameworks only provide support for the transfer of data that is residing in host memory. Then, an explicit data transfer between GPU and host memory is required. This adds additional work to both, CPU and GPU, and increases the latency of the GPU to GPU data transfer.

New technologies like GPUDirect RDMA help to overcome some of these limitations, as they allow other peripheral devices direct access to GPU memory. This enables network devices to directly transfer data between GPUs without any copies in host memory. However, these methods must be tested and evaluated, and their benefits for common communication models must be discussed.

In most common approaches, the communication and the data transfer among multiple GPUs is controlled by the CPU. This involves a dedicated CPU thread to control the data flow between host, GPU, and network device. If the GPU is enabled to control the communication, the CPU can be relieved this work. Thereby, context switches between CPU and GPU can be avoided and the CPU can be used for other work or set to a sleep state to save power. On the other hand, if the communication is controlled by the GPU, communication overhead is added to the GPU, which may slow down the performance of communication centric applications.

Most common interconnection networks for high performance systems were especially designed for communication and data transfer among processors nodes, controlled

by the CPU, transferring the data between the memory of these systems. So far, GPUs and other accelerators have only been of marginal interest for communication optimization. Still, several studies [11] have shown that data transfer will be one of the main factors for scaling and energy efficiency in next generation high performance computing systems. Therefore, evaluation of communication and data transfer methods for distributed systems, including GPUs and other accelerators, within the scope of energy and performance is mandatory.

1.1. Objectives of this work

In this work, different communication and data transfer methods for GPU accelerated clusters are designed, tested, and evaluated. In a first step, the GPUDirect RDMA technology is analyzed. Therefore, the support is added to the Extoll interconnection network and the *direct data transfer* is compared to previous methods, using staged copies in host memory.

To evaluate the performance of a host-controlled interface for inter GPU communication, GPUs are integrated to the one-sided communication interface GASPI. By this, the performance of a one-sided communication interface can be compared to the performance of a two-sided communication interface. Furthermore, a good performing host-controlled communication interface is required to be comparable against GPU-controlled communication.

So far, little research has been devoted to GPU-controlled communication. To be specific, as far as we know, no previous work exits, in which the GPU is enabled to control the network device and completely bypasses the host CPU. Therefore, one of the contributions of this work is to do exactly that. The first goal is to implement a put/get communication interface on the GPU, similar to the communication model used in GASPI.

As a further GPU-controlled communication model, communication over a shared global address space, based on remote load and stores, will be examined. Although this method shows some drawbacks for CPU-based communication, it is more in line with the GPU programming model and therefore meets some of the claims for GPU-controlled communication.

The objective of this work is not only to implement these communication method, but also to analyze them regarding performance and energy efficiency. As most interconnection networks are not designed for the use with GPUs, the scope of this work is also to analyze the strengths and weaknesses of different communication methods for GPUs and thus find the requirements for the communication of GPU-centric applications, helping the designers of next-generation hardware to optimize these also with the view to GPUs and other data parallel processors.

1.2. Outline

This thesis is structured as follows. The next section gives a short overview of the architecture of GPU-accelerated clusters and communication models for distributed systems.

Section three introduces the GPUDirect technology and compares direct and staged data transfers. In the fourth section, CPU-controlled communication for distributed GPUs is discussed. The GASPI standard and the integration of GPU memory segments is explained.

Section five deepens GPU controlled put/get communication. In this chapter, the individual steps that are required to enable GPU-controlled communication are explained in more detail. Also, the problems of GPU controlled communication and the GPU programming model are discussed in more detail here.

Section six introduces a hardware-supported global GPU address space as an alternative for GPU-controlled communication, which is more in line with the GPU programming model. The seventh section closes with a reflection on the achievements and the contributions of this work.

2. Background

Communication and data transfer in heterogeneous systems are multifaceted problems and many different aspects have to be considered to find an ideal communication method for distributed GPUs. Therefore, in this chapter, the different aspects of a GPU accelerated system are examined in more detail.

Figure 2.1 shows the simplified view of a GPU accelerated cluster. This cluster essentially consists of multiple *compute nodes* which are connected with a high performance *interconnection network*. Each of this compute nodes has got its own local *system memory* and one or more CPUs. The connection point between an interconnection network and the compute node is called *Network Interface (NI)*. In most clusters today, this network interface is located on a peripheral device, the *Network Interface Card (NIC)*. The *Graphic Processing Unit (GPU)* is also attached to the host system over the I/O-bus, normally over PCIe.

To optimize the communication, all this components have to be examined. In the following section, a short introduction to parallel computer architectures as well as communication and synchronization in this architectures is given. The subsequent sections give a short introduction to GPUs and GPU programming models, followed by a section about interconnection networks.

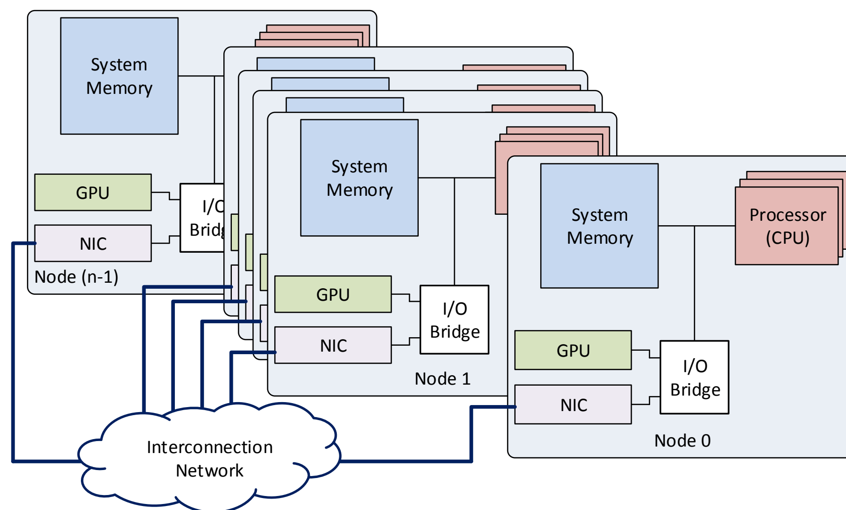


Figure 2.1.: Simplified structure of a GPU accelerated cluster

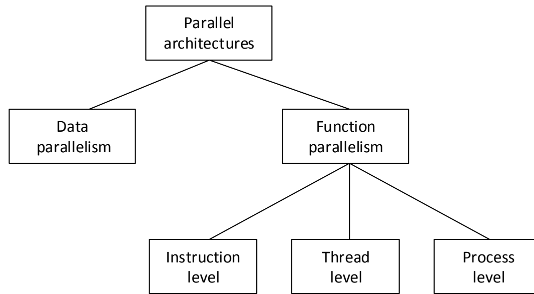


Figure 2.2.: Parallel architectures [12]

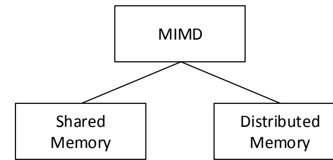


Figure 2.3.: MIMD architectures

With the background knowledge of those sections, in the last section of this chapter, a design space for possible communication methods for distributed GPUs is created.

2.1. Parallel Systems

Parallel architectures can be classified into two kinds of parallelism, *data parallelism* and *functional parallelism*, as shown in Figure 2.2 [12]. Data parallelism requires data structures like vectors or matrices, which allow a parallel execution of instructions.

Functional parallelism is divided into three levels, according to the level of parallelism: instruction level, thread level, and process level parallelism. The large amount of dependencies of instruction-level parallelism (*fine-grain parallelism*) requires a close coupling of the elements which can normally only be achieved in a single processing unit.

Process level and thread level parallelism correspond to the Multi Instruction Multiple Data Stream (MIMD)-architectures as introduced by Flynn [13]. MIMD architectures are divided into shared memory computers and distributed memory computers, also called multiprocessors and multicomputers.

2.1.1. Shared memory systems

A single node of a modern cluster –without the GPU– is a *shared memory* system. The memory of this machine is visible to all CPU cores and can be accessed over the same address space, without the involvement of a further unit. A single GPU with dedicated device memory is also a shared memory system since all compute cores of the GPU can access the device memory.

The communication in shared memory systems is handled over the shared memory. Using shared memory for communication, two things have to be considered: cache *coherence* and *consistency*.

Coherence Cache coherence guarantees that the *caches* of two processes, P1 and P2, return the same value for the same memory region. This is required since both processes can have a copy of the memory region in their caches. If one process updates the value, cache coherence guarantees that all other copies are updated

Listing 2.1: Example for memory consistency

```
int X = 1, Y = 2;
```

Thread_1:

```
void writeXY() {
 X = 10;
 Y = 20;
}
```

Thread_2:

```
void readXY() {
  int B = Y;
  int A = X;
  }
```

as well. Short inconstancies in the caches are allowed, as long as no process reads a wrong value.

Consistency Memory consistency specifies the legal ordering of memory load and stores with respect to all threads. The simplest consistency model is sequential consistency [14] which defines a total order for all load and stores across all threads. In a relaxed consistency model, this is not guaranteed.

To illustrate the effect of memory consistency, the example code in Listing 2.1 is used. In this example, the function `writeXY` is executed from thread_1 while the function `readXY` is executed from thread_2.

With a sequential consistency model, there are three different possibilities for the values of A and B:

- $A = 10$ and $B = 20$: if thread_1 executes `writeXY` before thread_2 executes `readXY`
- $A = 1$ and $B = 2$ if thread_2 executes `readXY` before thread_1 executes `writeXY`
- $A = 10$ and $B = 2$, if thread_1 writes to X before thread_2 reads X and thread_1 writes to Y after thread_2 reads Y

If a more relaxed consistency model is used, it is also possible that $A = 1$ and $B = 20$ for two possible reasons:

- At the time thread_2 reads X and Y, thread_1 has written to Y but not yet to X – at least from the perspective of thread_2
- Thread_2 reads X before Y and the writes to X and Y happen after thread_2 reads X.

Most GPUs have a relaxed consistency model. If consistency is desired, the programmer is responsible to add a fence operation manually. A fence operation guarantees that all instructions before the fence operation are executed before all memory instructions after the fence operation.

Another well-known consistency model is the Total Store Order (TSO) model that was first used in the SPARC architecture and later formalized as the x86-TSO model for X86 CPUs [15]. It is similar to the sequential order model but allows the reordering

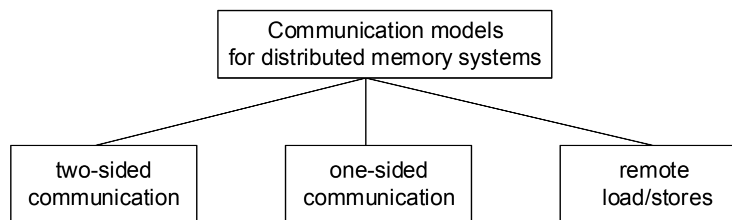


Figure 2.4.: Communication models in distributed memory systems

of a store to a subsequent (in program order) load to a different address. This allows the use of a First-In-First-Out (FIFO) buffer for stores, which enables the core to commit a store to this buffer without waiting for the store to access the cache. If an order between a load and a store request is required, the programmer or the compiler must add a fence instruction.

Atomic operations

To synchronize parallel programs on shared memory systems, often *atomic operations* are used. Atomic operations describe operations (or a set of operations) which appear uninterruptible for the rest of the system. Most systems provide, for example, an atomic compare-and-swap instruction. This instruction reads a value from a memory region, compare it with an expected value and writes out a new value, if the two match. If this operation would not be atomic, another process may change the value during one of these steps.

2.1.2. Communication and synchronization in distributed memory systems

The individual nodes of a cluster build a *distributed memory* system. The CPU of one node cannot directly access the memory of a different node. The memory is divided into multiple address spaces and, from a process perspective, is divided into local and remote memory. Access to remote memory requires special communication functions and additional hardware units since the accesses are routed through an interconnection network.

The implementation of this communication and synchronization is very important for the scalability of distributed memory systems. Figure 2.4 shows the most common communication approaches for distributed memory systems.

The most widely used communication paradigm is *two-sided communication*, as shown in Figure 2.5. As the name suggests, for two-sided communication, both sides, the source and the target process, are involved in the communication process. A *send* request on the source process requires always a *receive* request on the target process to be completed. The main advantage of this method is that the sending side does not require detailed information about the remote side, for example, the remote memory address. However,

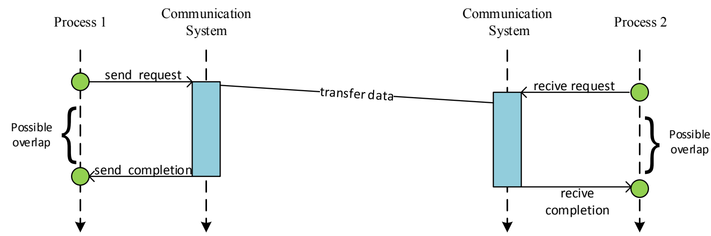


Figure 2.5.: Two-sided communication

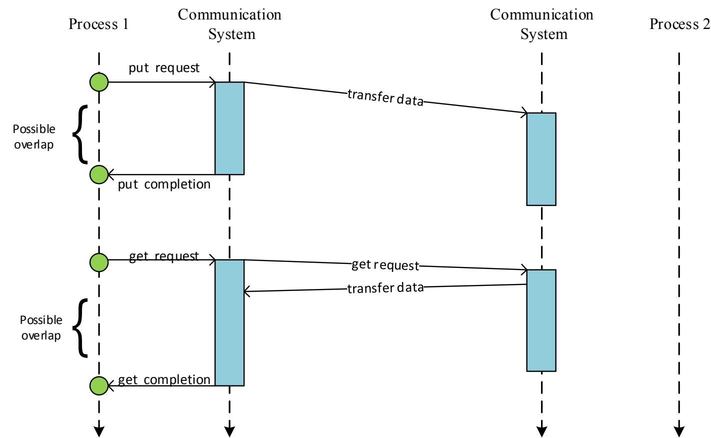


Figure 2.6.: One-sided communication paradigma

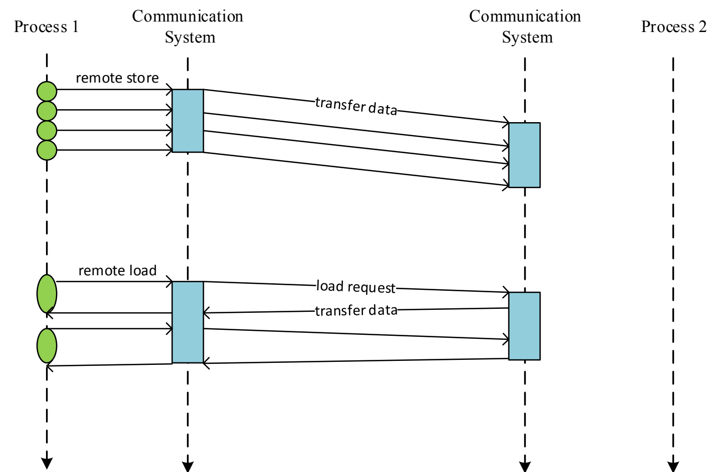


Figure 2.7.: Remote load stores

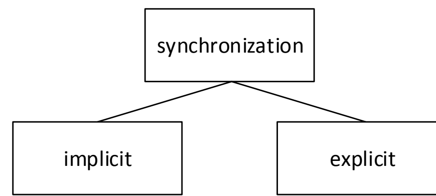


Figure 2.8.: Explicit und implicit synchronization

two-sided communication often also requires *tag matching* to match a received request to a received message.

In contrast to this, by using *one-sided communication*, as shown in Figure 2.6 on the preceding page, only one process is involved in the communication. By using a *get* or *remote read* operation, a process directly *gets* data from a remote memory region and copies it to its own, local memory. By using a *put* or *remote write* operation, a process directly *puts* data from its own, local memory to the remote memory of the target node. A put or get operation requires all information to perform the communication, which includes information about the remote memory region. However, the completion of a one-sided communication request does not depend on the remote process.

In contrast to the previously described approaches, *remote load and store* operations, shown in Figure 2.7, do not require special communication functions. Instead, writing or reading a value to or from a specified address triggers the communication. The remote memory is, so to say, *mapped* in address space of the local system. This approach requires hardware that allows forwarding of remote read and write requests, as, for example, described in [16] and [17], or an underlying framework, e.g., a compiler, that translates remote memory requests to one- or two-sided communication requests. In contrast to real *shared memory systems*, remote memory regions are not necessarily *cache coherent* and a more relaxed consistency model may be used.

Synchronization

A very important aspect of communication is the synchronization between source and target side. Synchronization is always strongly connected with communication: without synchronization, the source side would transfer the data to the remote side, but the target side would not notice this. As shown in Figure 2.8, synchronization can be either *implicit* or *explicit*. Using two-sided communication, the sending and the receiving side are always *implicitly* synchronized since both sides are involved in the communication process and a *send* or *receive* function is only completed with the corresponding function on the remote side.

Using one-sided communication or remote load/stores, an implicit synchronization between the source and the target side of the data transfer is not provided and *explicit synchronization* methods, like barriers or global locks, are required. The risk of race conditions is much higher if explicit synchronization is required. A process may transfer data

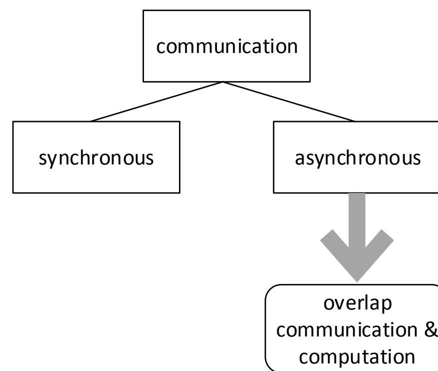


Figure 2.9.: Synchronous and asynchronous communication

to a remote buffer. This buffer may be in use on the target side which is not aware of the data transfer.

Still, the decoupling of synchronization and data transfer can help to improve the performance, especially for programs with irregular communication patterns like graph algorithms [18] or sorting functions [19].

Synchronous and asynchronous communication

Another characteristic of a communication function is the local synchronicity. A communication function can either be synchronous or asynchronous. A synchronous communication function blocks until the communication is completed – at least locally. An asynchronous communication function may return before the communication is completed. The communication is handled in the background, by the hardware or an underlying communication framework, for example. Asynchronous communication allows the overlapping of communication and computation. The communication process can do other work while the data transfer is handled in the background. An asynchronous communication operation consists of at least two functions: one function that starts the data transfer and another function that synchronizes this data transfer locally.

One-sided and two-sided communication can support both synchronous and asynchronous communication functions, as shown in Figures 2.5 and 2.6 on page 9 .

For remote load and store operations, asynchronous communication is not supported. A store operation triggers a communication request to a remote address and returns. In this case, the communication is locally completed. A remote load operation is not completed until the remote value is returned. The process that issues the remote load blocks until the communication is completed. Since the CPU can only handle a limited number of outstanding load operations, this could stall CPU-cores.

The terms *blocking* and *non-blocking* communication functions are often used in conjunction with synchronous and asynchronous communication, but there is a small difference: A non-blocking communication function may also return before the communication is

completed but the communication is not necessarily handled in the background but completed by the next call of the function or a local fence/synchronization function.

Communication buffers

Another property of a communication interface is the location of the *communication buffers*, the memory regions that can be used as source or synchronize of a data transfer.

Using one-sided communication, the source side requires all information for the data transfer, in particular information about the remote memory buffer. Therefore, one-sided communication often is only provided between so-called *windows* or *segments*. These segments have to be registered and the information about these buffers have to be shared between the processes before the communication can be started. The same is true for remote load/stores which also allow communication only between *shared memory regions*. This limitation may require extra data copies between shared and not-shared memory regions.

Two-sided communication can allow a data transfer between the complete memory areas of the participating processes. Thus, send- and receive- buffers can be allocated dynamically. Still, to provide this, the communication framework internally often still requires memory copying or special combination protocols which also add extra overhead to the communication.

Data transfer sizes

A further communication property is the possible *data transfer size* which describes the amount of data that can be transferred with a single communication request. For message passing and one-sided communication, theoretically any size can be transferred for a single communication request; for remote loads and stores, only transactions between 1 and 8 bytes are supported, which corresponds to the register size, respectively, the size of a char or double-value. Since many small data transfers are often not very efficient, internally, the communication framework can coalesce this communication request to a single communication request.

2.1.3. Collective communication functions

The approaches above describe communication functions between two points, a source and a target side. Apart from those *point-to-point* communication functions, also a great area of *collective* communication functions exist. In a collective communication function, all processes of a specified group participate. The simplest collective function is a *barrier*, which synchronizes all participants. Other well-known collective operations are *allreduce* and *reduce*, which combine the input data of all participants with a given operation, for example, a summation. For an allreduce operation, all processes return the result. Besides this, a wide area of collective functions exist to distribute and collect the data in a specified manner. Using a collective operation, all processes are implicitly synchronized, since a collective operation is not completed until all processes have at least entered the collective communication operation.

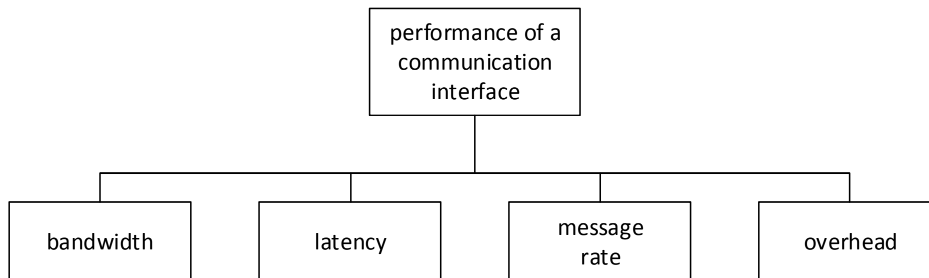


Figure 2.10.: Performance parameter for communication

Collective operations are often implemented on top of point-to-point communication functions. However, newer interconnection networks often provide special support for collective operations.

2.1.4. Communication performance

The performance of a communication interface can be evaluated in terms of bandwidth, latency, message rate, and overhead, as shown in Figure 2.10.

The *bandwidth* or throughput describes the amount of data that can be transferred in a period of time. The bandwidth varies for different data transfer sizes and is normally higher for larger transfer sizes. The *latency* describes the minimal time a message requires to be transferred, including communication request generation and synchronization. Hence, the minimal latency is reached for small messages. The *message rate* describes the number of messages that can be sent in a specific time period from a single process. In other words, it describes how well the processing of independent messages can be overlapped. The message rate is different for varying messages sizes and limited by the maximal bandwidth.

The communication *overhead* describes the time a processor spends in sending or receiving a message and cannot perform other operations [20]. It is a good measure for how well communication and computation can be overlapped.

The data transfer latency consists of the data transfer time, which allows overlapping, and the communication overhead. Therefore, the time span communication and computation theoretically can overlap can be calculated with:

$$t_{overlap} = t_{latency} - t_{overhead}$$

In [21], Grünewald describes the *overlap efficiency* of an application as a further measure for the communication overhead. The overlap efficiency η is defined as the pure compute time, normalized by the complete runtime of an application:

$$\eta = \frac{t_{compute}}{t_{runtime}}$$

If the communication can completely be overlapped, the overlap efficiency is equal to 1. However, due to the communication overhead, the theoretical peak overlap efficiency of 1 cannot be reached.

2.1.5. Communication Interfaces

To allow programmers the use of distributed memory systems, several communication Application Programming Interfaces (API) exist. These APIs use the different communication approaches described above. The following section gives a short overview of the most common communication APIs and languages for distributed memory systems.

MPI

The de-facto standard for message passing and the most commonly used communication interface for distributed memory systems is the Message Passing Interface (MPI). MPI-1 was released in 1994 to define a standard for communication in distributed memory computers [22]. The most important communication functions in MPI are the two-sided point-to-point communication functions *send* and *receive*. MPI-1 supports synchronous and asynchronous two-sided communication functions.

The specification also supports *buffered* and *synchronous* send operations. A buffered send can be called without a matching receive on the remote side. The send function may return before the remote receive operation is called. Therefore, the data must be buffered. On the other hand, a synchronous send operation requires a receive operation on the remote side to be completed. However, although the distinction between these two send operations may be very useful, they are only rarely used [23].

MPI-1 also support communication groups, topology related operations and various other functions. All in all, the standard defines 128 routines [23]. One very important group of functions are the collective operations like *MPI_Barrier*, *MPI_AlltoAll*, *MPI_Gather*, or the global reduction operations *MPI_Reduce* and *MPI_Allreduce*. A great deal of research was done to optimize these global operations, see for example [24, 25]. All these global operations are defined as synchronous and blocking and have to be called by all processes in a specified group.

In Version 2 of the MPI standard [26], MPI-2.0, 192 new routines were added [23]. These new functions add, for example, non-blocking asynchronous collective operations, C++ bindings, parallel I/O functions, and dynamic process management. Another important new feature is the support of *one-sided communication*.

The one-sided communication in MPI is handled between so-called *windows*. A window is created on pre-allocated memory buffers and for a group of processes. The pre-allocated memory buffers can differ in size and address for the different processes and even be set to zero.

MPI-2 defines three one-sided communication functions: `MPI_Put`, `MPI_Get` and `MPI_Accumulate`. All one-sided communication functions must be used together with one of the three explicit synchronization methods: fence, post-start-complete-wait, and locking. The one-sided communication functions are non-blocking (but not necessarily asynchronous) and completed at the end of a synchronization epoch.

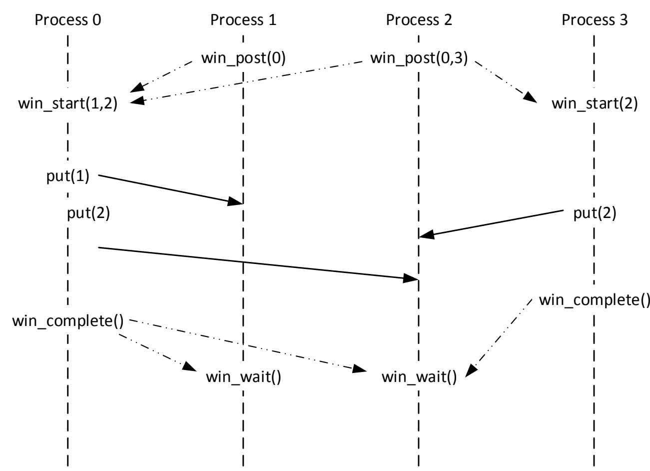


Figure 2.11.: General-active-target-synchronization in MPI [27]

The *fence synchronization* is comparable with a *barrier*. All processes must call the `MPI_Win_Fence` operation to demarcate the beginning and the end of a remote memory access epoch. All previous operations on remote memory are completed if the process leaves the fence operation.

The post-start-complete-wait synchronization is also known as general-active-target-synchronization (GATS). Figure 2.11 illustrates the GATS method for four processes. This method distinguishes between the processes accessing as targets (process 1 and 2) and processes issuing a one-sided communication request (process 0 and 3).

Processes that access as targets must call `MPI_win_post` before any process can access the window. Correspondingly, processes that issue communication requests must call the function `MPI_win_start`. When the source processes have posted one-sided communication requests, they have to call `MPI_win_complete` to force the completion of all these communication requests. The target processes must call `MPI_win_wait` to wait until all remote communication requests are completed.

The fence and the GATS synchronization require an active participation of the target side. In contrast to this is the *locking synchronization*, called *passive target synchronization* since the target side is not actively involved in the synchronization. To start a remote access epoch, the active side calls the function `MPI_win_lock`. This lock can either be exclusive or shared. While one process owns an exclusive lock, no other process can access the window. If a process owns a shared lock, no process can get access to a window with an exclusive lock. An access epoch is completed with `MPI_win_complete.` If the function returns, it is guaranteed that all remote memory operations are completed on the local and the target side.

However, in many MPI-2 implementations, one-sided communication is built up on top of send/receive operations. This adds a massive synchronization overhead to the communication. Furthermore, MPI requires explicit synchronization between the ori-

gin and the target side to guarantee that the communication is completed. Therefore, one-sided communication in MPI often suffers from a massive synchronization overhead and for many MPI versions, one-sided communication performs much worse than two-sided communication [28, 29]. The main problem is the underlying data consistency model of MPI-2, which was designed for high portability [30] and defines a *separate* memory model. This means that the user assumes MPI may maintain two copies of the exposed buffers – one for local and one for remote accesses. If the synchronization is performed, the MPI implementation synchronizes these two copies. Thus, MPI forbids overlapping accesses to the data on the shared window if one access is a write operation. In addition, local accesses cannot be performed concurrently since the MPI implementation is unaware if the local process updates a buffer. Any violation of these semantics is defined as an error.

Some of these limitations are solved with the upcoming MPI-3 standard [30]. MPI-3 supports *atomic operations* on windows. Another feature are request-handles for one-sided operations. One-sided operations in MPI-2 are non-blocking, but the completion depends on the completion of a synchronization epoch. In MPI-3, most of the communication requests can return a handle, which can be passed to a `MPI_wait` routine to ensure the completion of the operation.

MPI-3 also optionally supports a unified memory model. This model will relaxe some of the restrictions of the separate memory model described above. On shared memory machines, *shared windows* are supported. Shared windows allow the mapping of the memory of one process to the other process and thus, communication with load and store instructions.

PGAS

The term Partitioned Global Address Space (PGAS) is used for a number of different communication libraries and languages which relay on the PGAS-memory model. The idea of PGAS is to create global but logically partitioned address space. This means that all parallel processes share one global address space, but this global address space is created out of memory segments in the local memory of every process. For every process the global address space consists of a local and a global part. Besides this global memory which can be used for communication, every process also has got its local private memory which cannot be accessed by other processes. Figure 2.12 illustrates this basic memory model.

The term PGAS is used for both, communication libraries and special parallel programming languages, although most libraries are rather one-sided communication libraries than PGAS-APIs. The most common PGAS-languages are Universal Parallel C (UPC) [31] and Co-Array Fortran [32]; another is Titanium[33]. Relatively new parallel programming languages which are based on a PGAS-model are the Java-based language X10 [34], Fortress[35], and Chapel [36]. In [37], a relatively good overview over these new programming languages is given.

Co-Array Fortran is an extension to Fortran to allow parallel programming with some extension to the Fortran programming language. The data distribution is done with the

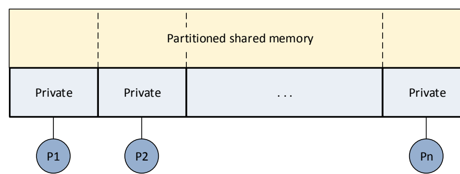


Figure 2.12.: PGAS Modell

co-array extension. A co-array is a data-structure which is distributed over all processes and threads, which are called *images* in Fortran. One process can access the memory of another process by using indices and load and store operations to such a co-array.

UPC is a parallel extension to the C standard. Like Co-Array Fortran, UPC uses distributed data structures which allow load/store access to a remote memory area by using the right indices. UPC also supports put and get functions to directly copy data between different processes, using a one-sided communication pattern.

These PGAS-languages are extensions to common programming languages and require special compilers. These compilers translate local accesses to a shared memory structure to normal, local load/store instructions while remote accesses are translated into communication requests. One possibility here would be to use a special hardware that forwards load and store instructions to remote memory, as described, for example, in [17]. However, since most hardware does not support this, normally the compiler translates remote load and store instructions to one- or two-sided communication operations, using an underlying communication framework.

A communication interface that is often used to deliver the communication for these PGAS-languages, especially UPC, is Global-Address Space Networking (GASNet) [38]. The main goal of GASNet is to provide a high performance, network independent communication interface for PGAS languages. During initialization, GASNet allocates on all nodes a shared memory segment. Remote processes can access these segments with one-sided put and get functions. GASNet is, for example, used in the Berkeley UPC compiler[39]. The distributed structures of UPC are allocated in the shared GASNet segments.

Next to PGAS languages, several PGAS libraries exist. These libraries allow direct one-sided put and get operations from and to special, shared memory regions or segments, but they normally do not allow direct load and store instructions. These shared segments create the partitioned global address space. GASNet, or, more precisely, the extended GASNet API, can be counted into this group.

Another example is ARMCI (aggregated remote memory copy interface), which is optimized for non-continuous data transfer operations. To allow a remote process to perform one-sided put and get operations, the memory must be allocated with a special allocation function.

A relatively new but widely known PGAS-API is *openSHMEM* [40]. The Symmetric Hierarchical Memory (SHMEM) communication library was originally developed for the Cray T3D [41] system and was a proprietary application interface. Different ven-

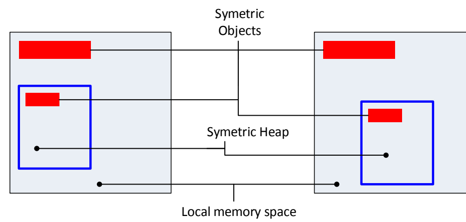


Figure 2.13.: OpenSmem memory model [42]

dors came up with their own SHMEM libraries, which diverged over the years. The idea of openSHMEM is to deliver a standardized, community driven open library for one-sided communication in distributed memory systems [42].

In openSHMEM, data objects can be allocated in local private or in remotely accessible memory. Remotely accessible objects are also referred to as *symmetric objects*. An object is symmetric if it has a corresponding object with the same type, size, and offset on all other processes. Symmetric objects can be allocated statically. Then they are located in the exact same area in the local address space of all processes. Symmetric objects can also be allocated dynamically during runtime. These objects are created in a special memory region called *dynamic heap*. The location of this dynamic heap is determined by the implementation and may be different on different processes, as shown in Figure 2.13. However, the offset of a dynamic object in the heap should be the same on all processes.

Remote dynamic objects can be accessed with put, get, and atomic operations. A *put* operation blocks until the data are copied out of the local buffer. The delivery on the remote side is not guaranteed. A *get* operation blocks until the data has completely arrived in the local buffer. Thus, these operations are not asynchronous.

Next to one-sided communication operations, collective operations are also supported. These collective operations are reduce and allreduce operations and operations to distribute and collect data among all nodes.

OpenSHMEM also supports remote load and store operations, if this is supported by the underling hardware. Otherwise, a remote load or store instruction results in a segmentation fault. Currently, this feature is mainly used on shared memory systems.

The main drawback of openSHMEM is that it does not support heterogeneous memory structures. The symmetric memory segments must have the same size in all processes. This makes the integration of accelerators like GPUs difficult. The lack of asynchronous communication operations is a further disadvantage of this standard.

A similar communication PGAS-API is the Global Address Space Programming Interface (GASPI) [43] and its implementation, GPI-2. In contrast to openShmen, it provides support for heterogeneous memory structures and asynchronous one-sided communication requests. A more detailed description of the GASPI-standard will follow in section 4.3, in which the integration of GPUs to this standard is discussed in more detail.

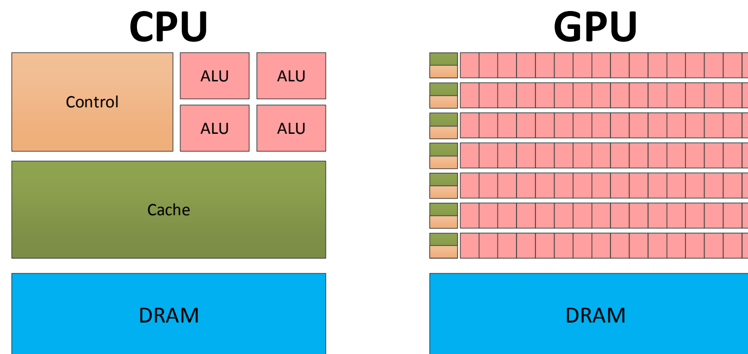


Figure 2.14.: Difference between a GPU and a CPU

2.2. Modern Parallel Processors

After analyzing parallel systems in the previous sections in detail, the next sections concentrate more on the individual parts of this systems. This section gives a short introduction to GPUs and especially the differences between GPUs and CPUs. This introduction primarily refers to the excellent book *Programming Massively Parallel Processors* by Kirk and Hwu[44], in which a more detailed introduction can be found.

2.2.1. Multicore and Manycore architectures

The hardware industry today is devolving into two directions for new parallel microprocessors: *multicore* and *manycore*. The multicore architecture began with a two-core processor by doubling the functionality of a single CPU core on the same chip. A current example is the Intel Xeon Processor E5-2697 v2, which comes with up to twelve cores. Each of these cores comes with a full X86 introduction set and supports hyper threading with two hardware threads per core.

The manycore direction focuses more on the higher data throughput. *Manycore*, therefore, is defined as a *larger number of smaller cores* with a *limited instruction set*. A current example is the Nvidia Kepler 40 GPU, which comes with 2,880 lightweight cores.

The main difference between a general purpose multicore CPU and a manycore GPU is illustrated in Figure 2.14.

The individual CPU cores are optimized for sequential code execution. Sophisticated control logic allows the execution of multiple instructions of a single thread in parallel. Techniques like branch prediction, executing multiple instructions in the same clock cycles (*instruction level parallelism*), and reordering of the instruction stream for out-of-order execution allow for improving the instruction flow on a single CPU and thereby increase the performance [2].

Furthermore , a modern CPU often has a small vector unit which allows performing the same instruction on a vector of input data in parallel, following the Single Instruction Multiple Data Stream (SIMD) after Flynn's taxonomy [13]. Examples for this are

the Streaming SIMD Extensions (SSE), designed by Intel, and the Advanced Vector Extensions (AVX) for the X86 instruction set architecture, proposed by Intel and AMD in 2008. Thereby, the performance of a modern CPU is not only determined by its clock rate. Still, CPU cores are mainly optimized for *sequential* code execution. This all leads to heavyweight, high performance compute cores, with a relatively high power consumption.

In contrast to this, the compute core of a GPU is very lightweight and features like branch prediction or pipelining are not part of the GPU core architecture. The reduced instruction set of a GPU core is not optimized for single threaded executions. Instead a GPU is optimized to perform a massive number of floating point instructions in parallel.

Due to the large number of lightweight compute cores, GPUs reach a very high theoretical peak performance with a lower energy consumption. CPUs and GPUs also differ in cache size and cache functionality. CPUs have very large caches to reduce instructions and the data access latency of complex applications. In contrast, GPUs have very small caches. Starting a massive number of threads in parallel optimizes the performance of GPUs. Caches are used to minimize accesses to the device memory. If a thread is waiting for a long latency memory access, another thread is scheduled. To allow a fast switching between threads, the control logic for an execution thread is minimized. In the following, a more detailed description of the architecture of a modern GPU is given.

2.2.2. Architecture of a modern GPU

Figure 2.15 shows the block diagram of a modern GPU, in particular the Nvidia GK110 Kepler GPU. The cores of a GPU are grouped in so-called Streaming Multi Processors (SM) (or SMX for Kepler GPUs). The GPU in Figure 2.15 has 15 SMXs; however, there are also Kepler GPUs with 13 or 14 SMXs. All SMXs share one L2 cache. Six 64-bit memory controllers provide access to the device memory. The GPU is connected to the host-system via a PCI express interface. The *GigaThreadEngine* is responsible for scheduling the threads to the individual SMXs.

Streaming multi processor architecture

Figure 2.16 shows a more detailed diagram of a Streaming Multi Processor (SMX) of an Nvidia Kepler GPU.

The compute cores of one SMX share the control logic and the instruction cache. One SMX comes with 192 CUDA-cores, 64 double-precision units, 32 special function units, and 32 load/store units. The 32 special function units (SFU) are used for fast approximate transcendental operations.

The scheduler of a GPU does not handle each single thread; instead, threads are organized in *wraps* (typically 32 threads). On a modern GPU, one SMX has more than one wrap scheduler. An SMX of the Nvidia-GK110 architecture has four wrap schedulers, as shown in Figure 2.16. This allows the execution of four wraps concurrently on one SMX. The eight dispatch units allow that two independent instructions of a wrap can

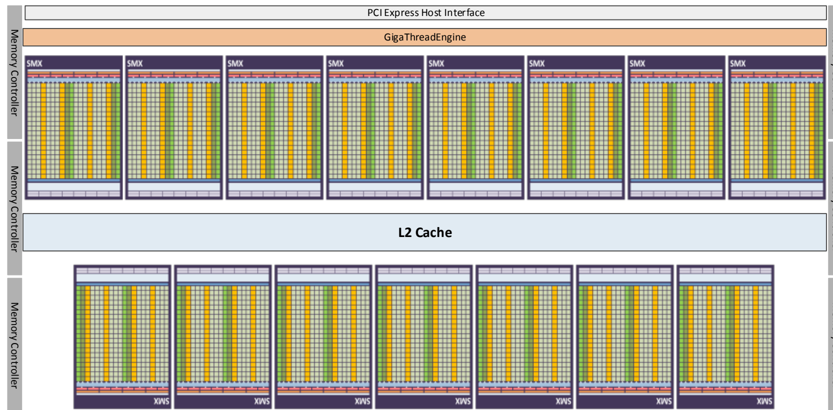


Figure 2.15.: Block diagram of a GK100 GPU (Kepler K20 architecture) [45]

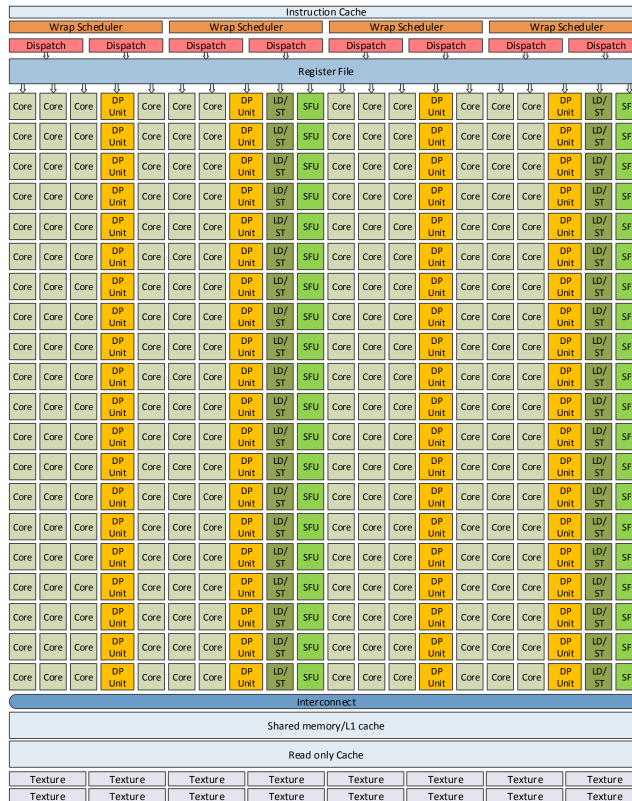


Figure 2.16.: Block diagram of a streaming multi processor of the Nvidia Kepler GK 110 architecture [45]

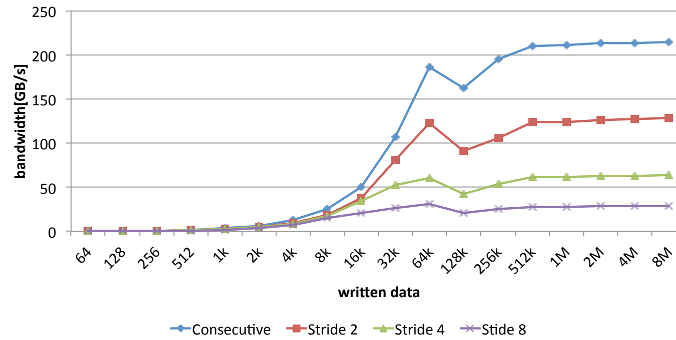


Figure 2.17.: GPU memory write bandwidth

be dispatched each cycle. On a Kepler GPU, every thread can use up to 255 registers. The complete register file has a size of $65{,}536 \times 32bit$.

The shared memory and the L1 cache use the same on-chip memory. Therefore, it can be configured as 48 kB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48 kB of L1 cache. The cores of one SMX share one L1 cache/ shared memory unit and one read only cache. Each SMX also contains sixteen texture filtering units, which are used to sample and filter image data.

Context switching on the GPU comes with very low costs and long-latency events can easily be hidden. For optimal performance, the threads of a single wrap should avoid branch divergences since this result in performance losses. If a branch divergence occurs, one thread does the work while all other threads block. Enough threads have to be ready to maintain full utilization when long-latency events occur. A well performing application typically runs with $5{,}000 - 20{,}000$ threads in parallel.

GPU device memory

A GPU comes with up to 12 GByte (Kepler K40) Graphical Double Rate DRAM (GDDR) memory, which is referred to as *global memory* or *device memory*.This device memory is shared by all SMs or SMXs and can be used for communication between the multi processors. GPUs also support atomic operations on this device memory.

Due to a simpler memory model and fewer legacy constrains than CPUs, GPUs are able to achieve much better memory bandwidth than CPUs. The memory bandwidth of the Nvidia Kepler K40 GPU is specified with 288 $Gbytes/second$, while for the Intel Processor E5-2697 v2, 59.3 $Gbytes/second$ are specified as the maximal bandwidth. One of the reasons for this is the higher cache line size. The size of one cache line of a Kepler K20 GPU is 128 Bytes, while it is normally 64 Bytes for X86 CPUs. GPUs can only achieve this bandwidth if the memory accesses can be coalesced. The memory controller of the GPU can coalesce the accesses to continuous memory regions from multiple threads in parallel.

Figure 2.17 shows the memory write bandwidth, measured on an Nvidia K20 GPU. The bandwidth is measured by writing a number of `double` values to GPU device memory to consecutive and non-consecutive addresses, varying the size of the stride

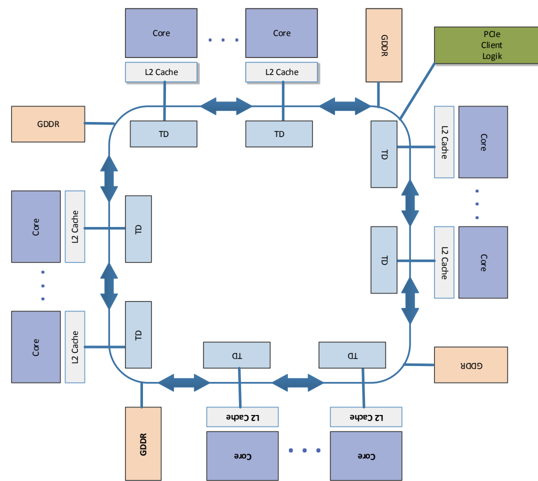


Figure 2.18.: Simplified block diagram of the Intel Xeon Phi [46]

between the written values. The full bandwidth is only reached if the accesses are to consecutive addresses. If only every second value is written, the bandwidth is halved; if the GPU only writes every fourth value, only a fourth of the bandwidth can be reached, and so on.

2.2.3. Other manycore processors

GPUs are the most commonly used accelerators for high performance computing and primarily used in this work. However, GPUs are not the only manycore processors. Therefore, now a short introduction to other manycore architectures – and their differences to GPUs – is given.

Intel Many Integrated Core Architecture

The Intel Many Integrated Core (MIC) architecture, a coprocessor developed by Intel, is, next to GPUs, the most popular accelerator in today's high performance systems. Of the Top500 List from June 2014, 62 systems use accelerators. While 44 of these systems use Nvidia GPUs, 17 use the Intel MIC technology [10]. One of these systems is the No.1 system of June 2014, the Titanhe-2 at the National Super Computer Center in Guangzhou.

The MIC technology is sold under the product name Intel Xeon Phi. In Figure 2.18 the simplified block diagram of an Intel Xeon Phi coprocessor, more precisely the Intel Knights Corner, is shown. Note that this is a simplified block diagram, not showing the actual number of cores. The Intel Xeon Phi actually provides up to 61 cores.

Each of these cores comes with a private L2 cache, that is kept fully coherent by a global distributed tag directory (*TD*). Four memory controllers provide direct interface

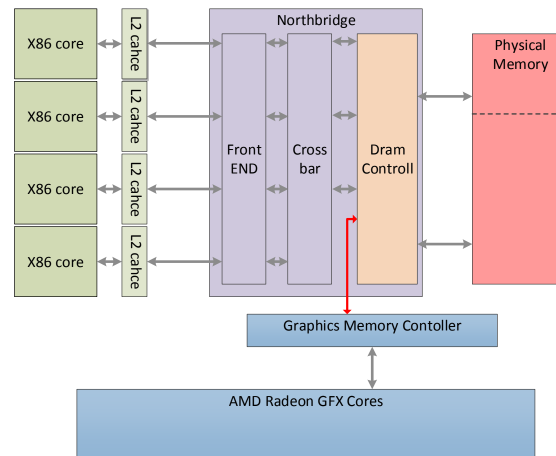


Figure 2.19.: Simplified block diagram of a AMD Liano APU[47]

to the GDDR5 device memory. The host is accessed over a PCIe client. All these components are connected with a ring interconnect [46].

The cores of the Xeon Phi are X86-CPUs, based on the Intel Pentium Processor but with 64 bit support. One core supports four threads in hardware. A further addition to the Pentium architecture is the vector processing unit (VPU) which supports a 512-bit SIMD instruction set. Thus the VPU can execute 16 single-precession or 8 double-precession per cycle, supporting the AVX instruction set.

In contrast to GPUs, the Intel Xeon Phi provides its own Linux operating system and it is possible to log into this system over SSH. The Xeon Phi supports x86 memory order model and IEEE 754 floating-point arithmetic. It is also possible to compile an application written in Fortran, C, or C++ directly for the Xeon Phi.

The current version of the Intel Xeon Phi, called Nights Corner, is only available as an add-in card. However, the next generation, Intel Knights Landing, will be available as both, a stand alone CPU and an add-in card.

AMD APUs

One of the bottlenecks for the performance of GPU applications is the data transfer between host memory and GPU memory over PCIe. To overcome this bottleneck, an Accelerated Processing Unit (APU) combines a general purpose CPU and a GPU-vector processor on the same die, sharing the same memory. In Figure 2.19, the simplified block diagram of the AMD Liano APU is shown. This APU has four general purpose x86 CPUs (AMD Stars), each with 1 MB L2 cache. The AMD Radeon GPU consist of 4 SIMD-processing units. Each of these processing units combine 16 VLIW-5 AMD Radeon^{TM} cores. Each of these cores has four 4 streaming-cores and one *special functions core*, resulting in five processing units for every VLIW-5 AMD Radeon^{TM} core and in 400 processing units for the complete Radeon GFX unit [47].

Since both compute units, the CPU and the GPU, can access the same DRAM memory, copies over the PCIe-bus can be avoided. The first generation of the APUs divides the system memory into two parts: one part is visible to be managed by the operating system running on the x86 cores and one part is managed from the software running on the SIMD cores. This requires a memory copy between the two memory regions. For that, AMD provides a highspeed block transfer engine. Despite this bottleneck, this first generation APU shows better performance than a similar discrete GPU, but lower performance compared to a GPU with more compute power [48].

APUs are currently used especially for mobile computing and low power devices. The third generations of APUs are also used for the Sony PlayStation 4 [49] and the Microsoft Xbox One[50] video game consoles. However, due to the lower performance of the CPUs compared to Intel Xeon CPUs and the lower performance of the GPU cores compared to current GPGPUs, they are currently not used in High Performance Computing.

2.3. GPU Programming Models

GPUs, as the name suggests, originally were designed for graphic processing, more precisely, for 3-dimensional rendering. Over the years, the GPU designs evolved form a fixed-pipeline design to a more unified processor design, resembling high-performance computers. Therefore, they become more and more interesting for researches from other areas.

However, the used APIs, openGL and DirectX, were designed for Graphic processing. Therefore, a programmer was forced to map the problem into native graphic problems. For example, to run multiple instances of a compute function in parallel, the computation had to be written as a pixel shader and the input data had to be stored into texture images to be readable for the GPU.

Nevertheless, some researchers ported applications to the GPUs and reached performance benefits compared to the CPU versions [51, 52]. That was the beginning of General Purpose GPUs (GPGPUs) [53].

Due to the growing interest in GPUs for general purpose computing, vendors and computer scientists started the development of programming languages and models for GPUs that allow the developing of programs for users who are not that familiar with classic graphical aspects.

2.3.1. CUDA

Nvidia was one of the first companies that realized the growing interest in GPUs for general purpose computing. This appears in both the software and the hardware development.

Firstly, their shader became a fully programmable processor, with large instruction memory, instruction cache, and instruction sequencing control logic. They also added memory load and store instructions with random byte addressability.

Table 2.1.: CUDA extensions for function calling

	executed on	callable from
`__device__ float DeviceFunc()`	device	device
`__host__ float hostFunc()`	host	host
`__global__ void MyKernel()`	device	host, device (*since* CUDA 5.0)

Simultaneous, Nvidia develops the Compute Unified Device Architecture (CUDA) [54, 55], a parallel computing platform with libraries, compiler, and runtime software to enable programmers who are not familiar with graphic processing to access the computational capabilities of GPUs.

For a CUDA programmer, a computing system consists of a *host* and one or more *devices*, a CUDA enabled GPU. A CUDA program consists of one or more phases that are either executed on host or on GPU. The source code provides both host code and GPU code. The NVIDA compiler separates both during the compile process.

The GPU device memory is managed from the host. The functions `cudaMalloc` and `cudaFree` are used to allocate and free GPU device memory and can only be called from the host. To transfer data between host and GPU, the host-functions `cudaMemcpy` and `cudaMemcpyAsync` can be used.

A GPU has at least one Direct Memory Access (DMA)-engine to access host memory. This means that a GPU can directly access host memory without any CPU-involvement. Host memory can be pinned and registered for the GPU. Pinned means that the memory is locked and cannot be swapped. That allows the GPU a static virtual-to-physical address translation. For registered memory, the GPU creates own page tables.

This enables the GPU to directly transfer data between GPU memory and pinned host memory region. Data from non-registered memory is buffered in pinned host memory buffers, which are managed by the CUDA runtime system. If pinned memory is used, the asynchronous copy operation `cudaMemcpyAsync` can be used. Then, the memory transfer is handled by the DMA engines of the GPU. Pinned host memory can either be directly allocated with `cudaMallocHost` or subsequently registered with `cudaRegisterHostMemory`.

The GPU device code is written in extended ANSI C, with special keywords labeling the data-parallel functions and data structures. In Table 2.1, the C extension keywords and their meanings are summarized.

Device functions that can be called from host are called *kernel* and must have a `void` type. They are labeled with the keyword `__global__`. The invocation of a CUDA-kernel on the GPU is also called *launch*. When a new kernel is launched, it is executed as a *grid* of parallel threads. These threads are organized in *blocks*. Both, a block and a grid can have a 3-dimensional structure. A grid typically consist of several thousand threads. In Figure 2.20, the structure of a two dimensional grid with four blocks is shown. If the host launches a kernel, it defines the size of the blocks and the grid via the *execution configuration* parameters. Listing 2.2 shows how a kernel is launched on the GPU.

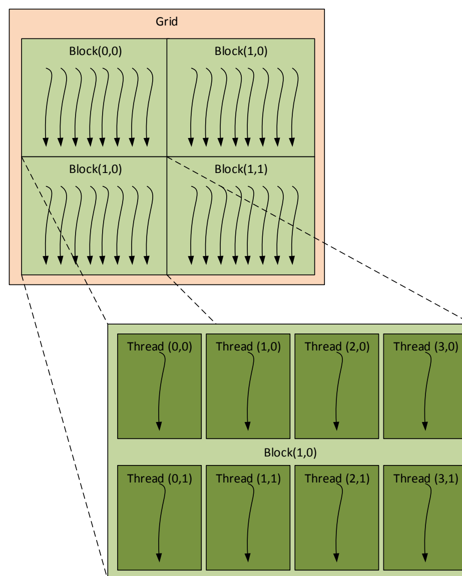


Figure 2.20.: A 2-dimensional GPU-thread grid with 2-dimensional blocks

All threads execute parallel the code in the kernel function. Therefore CUDA is an instance of the Single Programm Multiple Data (SPMD) programming style [56], a classic example of a data-parallel programming style. A kernel innovation is asynchronous; this means that the function returns while the kernel is still running on the GPU.

Listing 2.2: CUDA kernel launch

```
cuda_kernel<<<blocks, threads, 0,stream>>>(par1, par2, ···);
```

A kernel is launched to a so-called *stream*. A stream is a sequence of operations that are executed in order on the GPU. This means that two operations on different streams can be executed concurrently. Streams are often used to overlap computation on the GPU and data transfer between host and GPU. Since CUDA 4 not only is it possible to overlap the data transfer and the computation, but also multiple kernels can be executed simultaneously if they are submitted to different streams. If no stream is specified, the so-called *null-stream* is used.

In the past, only the host was able to launch a new GPU-kernel. However, the new *dynamic parallelism* feature, which was introduced with CUDA 5 and Kepler Class GPUs, now allows launching of GPU-kernels directly from the GPU.

The CUDA memory model

CUDA threads have access to different memory spaces during their execution. The CUDA memory model is closely related to the GPU-architecture, as shown in Figure 2.21. All threads running on the GPU have access to the *global device* memory. The threads of one block access the same *shared* memory region. The memory access latency

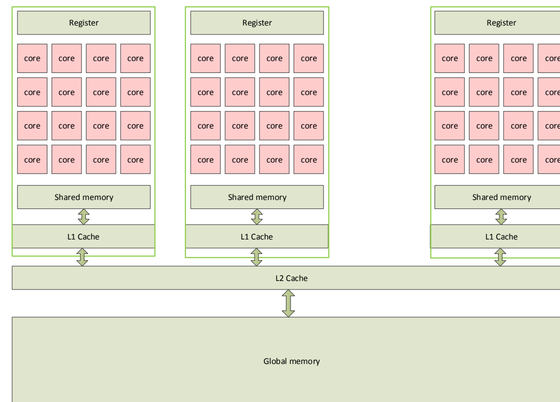


Figure 2.21.: CUDA memory Modell

to shared memory is much lower than the latency to the device memory. This can be used for optimization.

Furthermore, every thread has a private, local memory. The local memory is allocated in the device memory and therefore has got the same long access latencies as access to global memory. Therefore, local static allocated variables are often held in the registers. The registers of one shared multiprocessor are dynamically distributed between the threads and are also used to store local and global data.

Access to the global memory is cached. On Kepler architecture, the L1 cache is reserved only for local memory accesses such as register spills and stack data. Accesses to global memory are only cached in L2 cache [57].

2.3.2. OpenCL

Next to CUDA, several other approaches exist to use GPU for accelerated computing. Open Computing Language (OpenCL) by Chronos [4] is a low-level programming interface for parallel programming which can also be used for GPUs. The syntax and program structure of an OpenCL program is very similar to CUDA. As in CUDA, an OpenCL program consists of parallel *kernels* and sequential host functions. A kernel can be task-parallel or data-parallel; on GPUs only data-parallel kernels make sense. Kernels and data transfers can be submitted to *queues*, similar to CUDA streams.

In OpenCl, the work is distributed between so-called *work-items* which are grouped in so-called *work-groups*. This concept is very similar to the threads and blocks in CUDA.

Also, the openCL memory model is very similar to CUDA. The *private memory* can only be accesses by a single work item (CUDA: local memory), the *local memory* in OpenCL can be accessed by all work-items of one work-group (CUDA: shared memory) and the *global device memory* can be accessed by all work-groups (CUDA: global memory).

In contrast to CUDA, OpenCL cannot only be used for Nvidia GPUs, but it can be seen as a framework for heterogeneous systems, consisting of central processing units (CPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs), the

Intel Xeon Phi, and other coprocessors. OpenCl is also not only usable for Nvidia GPUs, but can also be used on GPUs from other vendors, for example, AMD. In most applications, the performance of OpenCL is worse than the performance of a CUDA-program on an Nvidia GPU, but with some GPU specific optimizations a similar performance can be reached [58][59]. However, since CUDA provides some features which are required in this work, in particular dynamic parallelism and GPUDirect, and that are not yet supported in current openCL versions, CUDA is used in this work.

2.3.3. Directive-based approaches

Some developers are of the opinion that using low level APIs like CUDA or OpenCL result in an unproductive development process since a lot of code has to be ported, which is also very error-prone [60]. Therefore, several directive-based approaches have been proposed. The idea of directive based programming is that the user adds only a few lines with compiler hints to a sequential code while the compiler implements the parallel accelerator code. The most widely used standards for directive-based accelerator programming are *OpenACC* [5] and OpenMP extensions for accelerators [61].

OpenACC is an industrial standard which was developed by Cray, Nvidia PGI, and CAPS. The specification describes a collection of compiler directives for C, C++, and Fortran which allow the offloading of compute intensive tasks to an accelerator, for example, a GPU.

OpenMP was designed for productive programming in shared memory machines. With version 4.0, additional directives to support accelerators were introduced.

However, although for some applications almost 90% of the performance of an application written in CUDA can be reached, for other applications, only 50% of the performance of a CUDA program using a directive-based approach are reached [62].

Besides, for Nvidia GPUs, most openMP/OpenACC compilers internally translate the directives to CUDA or openCL code. The communication concepts analyzed in this work are realized in CUDA. A next step would be to allow the use of these concepts in conjunction with OpenACC or OpenMP. However, this is not the scope of this work, therefore directive based approaches are not considered here in more detail.

2.4. Interconnection Networks and Network Interfaces

Another important part of a *cluster* are interconnection networks. In this section, a short introduction to Interconnection Networks (IN) and Network Interfaces (NI) is given. The section ends with a more in depth analysis of two Interconnection Network (IN), Infiniband and Extoll, which are used in this work. The goal of this section is to get a better understanding of the functionality of IN and Network Interface (NI). This is necessary in order to find a suitable communication model for inter GPU communication and data transfer.

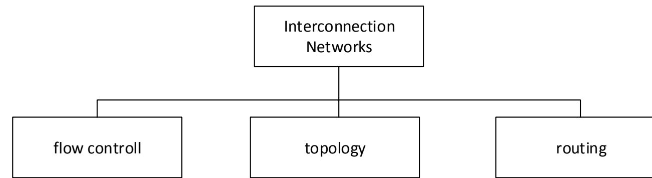


Figure 2.22.: Design space of network interfaces

2.4.1. Interconnection networks

An Interconnection Network is defined as a programmable system that transports data between terminals [63]. It is programmable in the sense it that differences connections at points in time. *Switches* are opened and closed to provide a connection path from one terminal to another [64].

This definition of a IN occurs at many stages, from the on-chip network that delivers data transfers between memory areas within a single processor to wide-area networks that connect distribute systems. In this work, mainly the interconnection networks between distributed nodes with distributed memory are considered. However, also the interconnection network between host and GPU and the GPU and another peripheral device, for example, a network adapter, have to be considered. Figure 2.22 shows the design space of interconnection networks.

The *topology* is very important for the scalability. It describes the arrangement of nodes and the connection between the nodes. The topology can either be direct or indirect. In a direct topology, every node is directly connected to another node. An indirect topology uses intermediate switches.

The *routing* method describes, how a package is send from one node to the destination node. While the topology determines the ideal performance of a network, the routing is one of the key factors of how much of the potential performance could be realized. Routing algorithms are classified to *deterministic* and *adaptive* algorithms. Deterministic algorithms always choose the same path between two nodes, without using further information like the current network traffic, for example. Adaptive algorithms adapt the route to the state of the network, using state information like the status of a node or historical channel information. The *flow control* is required to manage the allocation of resources to packages. The key resources of most interconnection networks are buffers and channels. Buffers are registers or memories that allow holding a package temporarily. In the following, some interconnection networks which are of interest in this work are examined in more detail.

2.4.2. PCI-Express

The GPUs – and in most cases also the network interface for the connection to a remote node are connected to the host system over the Peripheral Component Interconnect Express (PCIe)-bus. The PCIe-bus is a *serial bus* with point-to-point connections. In Figure 2.23, the topology of the PCIe-bus is shown. The *Root Complex* connects the CPU

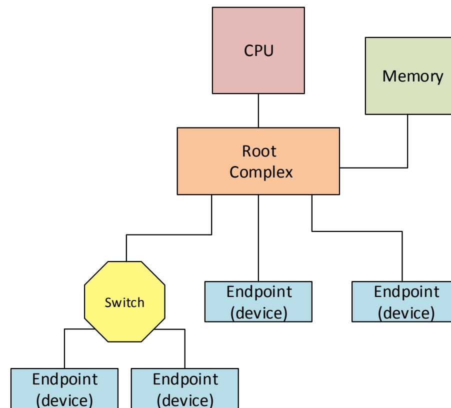


Figure 2.23.: PCIe topology

and system memory to the PCI Express fabric and may support several PCIe ports. A port can be connected to an endpoint or a *switch* which then forms a sub-hierarchy.

The CPU can order the root complex to send requests over the PCIe-fabric. The root complex transmits packages out of the port to the devices and also forwards incoming requests to the CPU or the system memory. The root complex does not necessarily support the transport of a package from one port to another; this is the case for many older systems. However, this peer-to-peer data transfer is necessary for a direct data transfer between a GPU and a other peripheral device like a network interface. Therefore, one of the conditions for a direct communication is that the used PCIe-bus and root complex support peer-to-peer communication.

PCIe transaction can be divided into posted (write operations) and non-posted (read operations) transactions. A non-posted transaction requires a completion (with the read data) to be completed, while a posted-transaction consists of a memory write packet and does not require a completion.

A PCIe interconnection consists of either a $\times 1$, $\times 2$, $\times 4$, $\times 8$, $\times 12$, $\times 16$, or $\times 32$ link (physical connection).The bandwidth of one lane (without protocol-overhead) is 500 MB/s for PCIe 2.0 and 984.6 MB for PCIe 3.0. This leads to a bandwidth of 4 GB/s or 15.754 MB/s for device that is connected through a $\times 16$ slot, most commonly used to connect GPUs to the host system.

2.4.3. QPI and Hyper Transport

QuickPath Interconnect (QPI) and Hyper Transport (HT) are processor interconnects. Intel's QPI connects one or more processors and one or more I/O-hubs on a single motherboard, enabling all components to access the other components over this network, see in Figure 2.24. QPI is a point-to-point interconnect. As the block diagram in Figure 2.24 shows, the DRAM memory banks are directly attached to one of the processors. All processors can access the complete memory, with more latency if accesses are routed through the QPI interconnect. Therefore, such a machine is called a

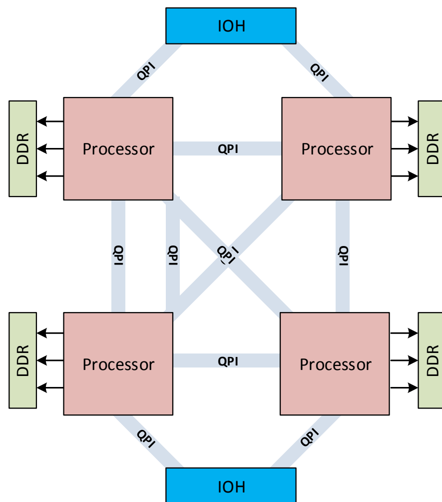


Figure 2.24.: QPI interconnect

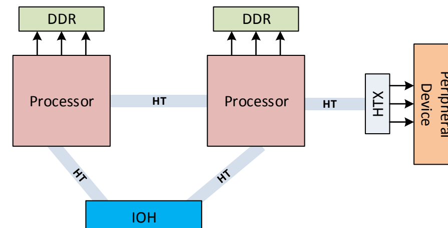


Figure 2.25.: HT interconnect

NUMA (non uniform memory access) machine or, more precisely, a Cache-Coherent (CC-NUMA) machine as the caches of the processors are kept coherent.

In the system shown in Figure 2.24, peripheral devices can be connected to both I/O-hubs. This can lead to different access times to the host memory from a peripheral device, depending on the location of the memory bank memory and the peripheral device. Peer-to-peer accesses between two peripheral devices that are located on different I/O-hubs also have to be routed through the QPI link.

Hyper Transport(HT) is an interconnection network for multiple Processors and the I/O-hubs, similar to QPI. HT is mainly used for AMD CPUs, while QPI is used for Intel processors. In contrast to QPI, the HyperTransport eXpansion (HTX) also allows slot-based peripheral devices a direct connection to a micro processor.

This allows the use of plugin cards that are directly connected to the CPU and have direct access to the system memory, without going over the I/O-hub or a PCIe bridge, as shown in Figure 2.25.

2.4.4. Network Interfaces

The connection point between an interconnection network and the network client is called the Network Interface (NI). The NI is often one of the main factors for the performance of an interconnection network. If such network interface is located on a peripheral device, it is called Network Interface Card (NIC).

Two register interface

The simplest network interface is the *two register interface* shown in Figure 2.26 [63]. This interface has two registers, one for sending and one for receiving messages. To send

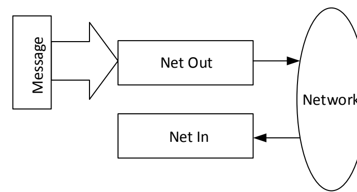


Figure 2.26.: Two register interface

messages, a processor moves a message word by word to the send register. A special move instruction marks the last work and terminates the message.

On the receiving side, the processor reads the message word by word out of the register. To synchronize with the sending side, the processor may test the register for a new message or block it, until a new message arrives.

However, this network interface has got two disadvantages: First of all, it adds a lot of overhead to the processor, because it has to move the message to the register word by word, so it is serving as a DMA-engine. The second problem is that the network interface is not protected from software running on the processor. If a processor starts to send a message and then delays infinitely, the partial message can also occupy resources like buffers infinitely.

For a safe network interface, any shared resource must be released in a limited amount of time.

Descriptor-based interface

A descriptor-based interface overcomes the limitations of the two-register interface [63]. The processor composes a descriptor for the message that should be transferred. The descriptor may contain an immediate value or an address to a memory block and the size of the required data transfer. The processor moves this descriptor to a set of dedicated descriptor registers in an atomic manner. A dedicated hardware on the network interface now can process the message. Thereby, the overhead for the CPU is minimized.

On the receiving, side also the network hardware can handle the incoming messages. Here, a descriptor may be required to prepare the buffer for an incoming message. Another solution would be the use of pre-allocated buffers.

2.4.5. Remote Direct Memory Access

A message transfer always requires a source process and a destination process. Although the data transfer can be handled by dedicated hardware, a processor is still required to initiate the data transfer (prehaps with a descriptor) on the sending side and to accept the data on the remote side.

Direct Remote Memory Access (RDMA) capable interconnects, like Infiniband, Extoll or Myrinet are widely used in high performance computing. RDMA means the

network device can directly transfer data from the memory of one node to the memory of another node, without involving the processor on the target side. On the target side, the network hardware can handle an incoming request from a remote process by completely bypassing the host CPU.

Virtual memory and memory registration

User space applications normally use virtual addresses (VA) to address host memory. The CPU translates these addresses to the physical addresses (PA). Normally, the memory belonging to a virtual address is physical not continuous.

A user space program usually only knows the virtual address due to security reasons, while a peripheral device requires the physical address to access the memory area. There are two possible ways to allow user space applications communication with virtual memory areas as communication buffers: Either, every communication request requires access to the operating system to perform a physical to virtual address translation, or the peripheral device, here a NIC, has is own memory management unit (MMU), which allows a virtual to physical address translation. One-sided RMDA communication is only enabled in the second case. However, to allow a static virtual to physical address translation, the host memory must be locked, since a locked memory area cannot be swapped.

Therefore, the device drivers of RDMA capable NICs provide a function to lock and register host memory. This function locks the memory, performs a virtual to physical address translation and creates page tables for the NIC.

Memory pinning and registration is the main bottleneck of RDMA-based communication: it is a very expensive operation [65], not only because of the memory registration but also because of the data exchange with the remote side. To allow RDMA communication, the sourcing processor requires information about the remote registered memory segments. Normally, the virtual address is not sufficient for communication, due to security reasons, and keys or special network addresses are required. Therefore, different strategies exist [65] and much work was done in optimizing memory registration for higher-level communication libraries [66].

RMDA in communication APIs

As RDMA is a common data transfer method, it is widely used in common communication APIs. In two-sided MPI communication, two different protocols, depending on the message size, are used.

The eager protocol, which is mostly used for small messages, uses pre-allocated and pinned buffers to cache data and meta-data [67]. For a data transfer, the data is copied to and from these buffers. The advantage of this protocol is that latency is very small, but it adds some overhead to the CPU due to the extra memory copies. Because of the size of the buffers and the overhead, this protocol is not suitable for larger messages.

Therefore, for larger messages the rendezvous protocol is used [68]. One side (it can be either the sending or the receiving side) initially sends the meta-data about the communication request to the remote side. This meta-data is matched with a local send

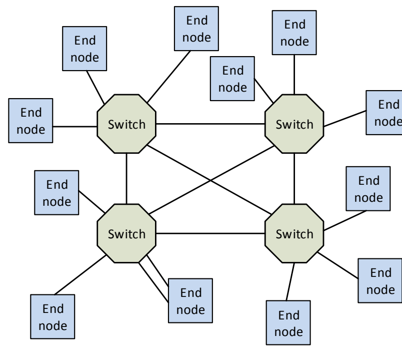


Figure 2.27.: Infiniband subnet topology

or receive call and then the data transfer is started. If the data transfer is completed, both sides are notified about the completion. Using the eager protocol, the memory buffers for the data transfer are registered dynamically. This approach avoids extra memory copies, but due to extra data transfers and the memory registration, the latency is affected.

These techniques are often used together with *lazy* pinning- and unpinning strategies [69, 70, 71]. The memory is not directly unpinned if the communication is completed, but stays pinned for as long as possible. A pinned memory region may be used in further communication and thus unnecessary pinning and unpinning can be avoided.

In this work, two different RMDA capable interconnection networks are used: Infiniband and Extoll. The next sections give a short overview of these interconnection networks.

2.4.6. Infiniband

Infiniband is an industry-standard architecture for server I/O and inter-server communication. It was developed by the *InfiniBandSM TradeAssociation* (IBTA)[1] to provide high levels of reliability, availability, and performance. In the last years, Infiniband gained high popularity in high performance computing. In fact, over 40% of the Top500 systems today use Infiniband as interconnect [10]. In the next paragraphs, a shot introduction to Infiniband is given. A more detailed description can be found in [72].

The topology of a so-called Infiniband subnet, the smallest unit in an Infiniband network, is shown in Figure 2.27. The endpoints of an Infiniband network are connected to switches. Infiniband provides an *indirect* topology. The switches can also be connected to other Infiniband switches. All data transfer in Infiniband is point-to-point, not busted. This should provide failure isolation and allow scaling to large networks.

Messages are sent from one endpoint, for example, a host node, to another endpoint over the links and routed by the switches. To create larger networks, routers can join multiple subnets. Infiniband uses a deterministic routing.

[1] `http://www.infinibandta.org/`

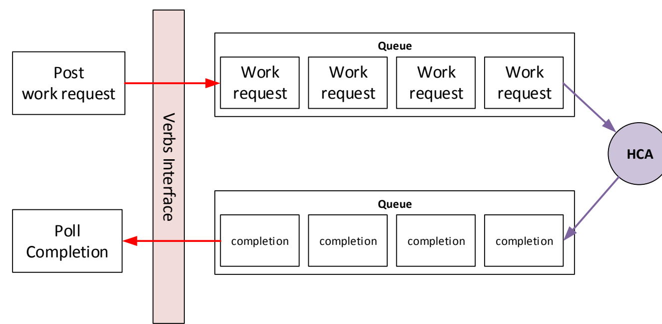


Figure 2.28.: Work request processing in Infiniband

The network interface card for Infiniband is acalled Host Channel adapter (HCA). Normally, the HCA is a network device, which is connected to the node over the PCIe-bus. The software interface required to control the HCA from host is called *Infiniband verbs (ib verbs)*.

Memory registration in Infiniband

To allow an Infiniband HCA to access host memory, the memory has to be locked and registered. This applies to both memory that is required for communication and memory that is required for communication resources. To manage the memory, a table of the registered pages, the Memory Translation Table (MTT), is used. This page table is normally allocated in host memory. To allow the HCA to access the memory, the virtual address and a specified key is required. For local accesses, the so-called local-key (l-key) and for remote accesses the remote-key (r-key) are used. The keys are created during memory registration. The Infiniband HCA uses the key and the virtual address to find the right entry in the MTT and to determine the physical addresses.

Communication in Infiniband

Communication in Infiniband is handled between so-called queues. There are three kinds of queues: send, receive, and completion queues.The send and receive queues are always used together as Queue Pair (QP). Each send and receive queue is assigned to one completion queue, but a single completion queue can be used by several send and receive queues.

Queues are ring buffers in host memory, which are pinned and registered for the network device, so the HCA can directly read and write the data from this queues.

Figure 2.28 shows the processing of a work request in Infiniband . A host process submits a work request to a queue and the HCA processes this work request. When the HCA completes the processing of a work request, it optionally places a completion notification to the associated completion queue. This notification is called completion element. The host can process this notification to ensure that the data transfer is com-

pleted. Posting of work requests and polling for a completion is handled from the Infiniband verbs API. Infiniband supports the following communication requests:

RDMA_WRITE is a one-sided communication request to *put* data to a remote memory region. This request does not require a remote work request to be completed. Therefore, the work request needs information about the local and the remote memory region. Write requests are submitted to the send queue.

RMDA_READ is a one-sided communication request to *get* data from a remote memory region. This request does not require a remote work request to be completed. Therefore, the work request needs information about the local and the remote memory region, including local and remote memory keys. Read requests are also submitted to the send queue.

SEND is a request to *send* data to a remote side. It has a two-sided semantic, so it requires a receive request on the remote side to be completed. A send request only requires information about the local memory segment and is submitted to the send queue.

RECEIVE is a request to *receive* data and is required to complete a send request from a remote process. A receive request is submitted to the receive queue and must be posted before the send request on the remote side, otherwise the send request fails. The receive request requires information about the local memory segment. It is submitted to the receive queue.

RMDA_WRITE_WITH_IMM is a communication request some what inbetween one-sided and two-sided communication. Remote write requests with immediate data require the same information as normal write request plus an additional immediate value. In contrast to normal remote write requests, a receive request on the remote side is needed to successfully complete the communication. The destination address and the key of the receive request can have an arbitrary value and will be ignored. Yet without the receive request, a remote write request with immediate data fails. The benefit of a remote write request with immediate data is that a completion element is created on both sides. The completion element on the target side contains the immediate value. This technique provides an implicit synchronization mechanism for one-sided communication.

SEND_WITH_IMM is a request similar to normal send requests but with an additional immediate value that is added to the completion element on the receiving side.

FETCH_AND_ADD performs an atomic *fetch-and-add* operation on a remote memory segment. This request is also submitted to the send queue. This atomic operation atomically fetches a value from a remote memory segment, adds a specified value and writes the result back. The completion element returns the old value to the process.

COMPARE_AND_SWAP performs an atomic *compare-and-swap* operation on a remote memory segment. This request is also submitted to the send queue. It atomically compares a remote value to a given value. If the values are equal, the old value is replaced with the predefined new value. The completion element returns the old value to the process.

Transport types in Infiniband

If a QP is created, it must be associated with one of the following transport types:

Reliable Connection (RC) One RC-QP is connected to exactly one other RC-QP. The data transfer is reliable, the order of the data is guaranteed.

Unreliable Connection (UC) One UC-QP is connected to exactly one other UC-QP. Data may get lost during transport and the order is not guaranteed. The data transfer size is limited to a maximal transfer unit (MTU), which is between 2 kB and 4 kB for current hardware. RDMA-write requests are not supported.

Reliable Datagram (RD) An RD-QP can send and receive data from multiple RD-QPs using a reliable datagram channel. RD-QPs are optional and therefore often not supported in current hardware.

Unreliable Datagram (UD) A UD-QP can send and receive data from multiple other UD-QPs. It is the most basic transport for Infiniband and therefore has got a number of limitations. The message size is limited to the MTU size and RDMA operations are not supported.

Most communication libraries like GPI or MPI use RC-QPs for communication as they provide a lot of benefits compared to the other QP types. First of all, all RDMA communication types are supported. The data payload is not limited to the MTU size and therefore, a large message can be transferred with a single work request. The main advantage, however, is the data reliability. The hardware recognizes if data is dropped and the order of messages is guaranteed. For unreliable transport, this has to be guaranteed by the software which is more costly.

However, the main disadvantage of RC-QPs is the *memory footprint* [73]. Every RC-QP can only be connected to one other QP, therefore, for every connection, a new QP is required. As mentioned above, for every QP, the queues for the work requests have to be allocated. For larger systems with several thousand processes, this memory footprint can become a problem.

To overcome this limitation, in [74] and [73] UD-QPs were used for two-sided MPI communication. The memory footprint of MPI could be reduced from 140 MB to 40 MB for every MPI process with 8k MPI endpoints. However, due to the limitations in message size and reliability, the latency increased by around 23% for 1 kb messages and the bandwidth droped about 9% for the optimized version compared to a data transfer using RC-QPs.

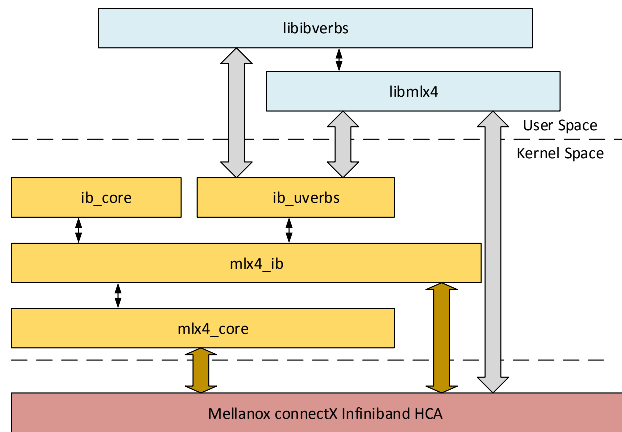


Figure 2.29.: Simplified view of the Infiniband software stack [75]

Infiniband Software Stack

In Figure 2.29, a simplified diagram of the Infiniband software stack is shown. The block diagram shows the parts of the Infiniband stack that are important for this work.

User space applications use the Infiniband verbs library (*libibverbs*) to access the Infiniband hardware. However, since this library mainly defines the Infiniband verbs API, which is valid for multiple devices, below this general library, a device specific library is required to implement the device specific function calls. This example shows the required libraries and device drivers for the Mellanox ConnectX, ConnectX-2, and connectX-4 Infiniband HCAs.

The same applies to the kernel space. The *ib_core* driver defines the device-independent functions and forwards requests to the hardware specific drivers. For communication between kernel space and user space, a special driver, *ib_uverbs* is required. This driver forwards user space requests to the core driver.

The Mellanox ConnectX cards can operate as both, an Infiniband adapter and an Ethernet adapter. Therefore, the device driver is split up. The *mlx4_core* driver handles low-level functions like device initialization and resource allocation. The *mlx4_ib* driver handles infiniband specific function. Another special driver (not illustrated in Figure 2.29) handles the ethernet specific functions.

Infiniband also allows direct access to the hardware from user space by completely bypassing the operating system. For this, device specific registers are mapped to the user space. The required functions are implemented in the device specific user space library.

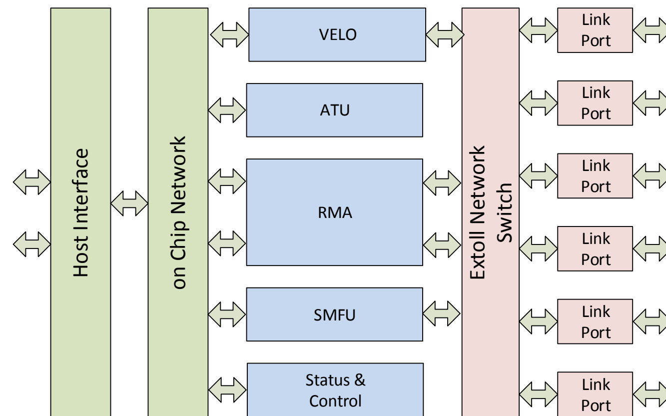


Figure 2.30.: The principle block diagram of the Extoll NIC[76, 80]

2.4.7. Extoll

The Extoll interconnection network was developed at the university of Heidelberg and is now sold by the Extoll Company.[2] In Extoll, the endpoints are directly connected and no switches are used, so Extoll provides a *direct* topology with up to 64k nodes and naturally supports a 3-d torus topology. Extoll supports deterministic and adaptive routing strategies on package granularity. The messages are sent from one node to another and routed through the network interface cards.

The Extoll NIC

The general block diagram of the Extoll NIC is shown in Figure 2.30. A more detailed description of Extoll can be, found for example, in [76]. The complete architecture is formed through major blocks: the host interface, the on-chip network, the functional units, the Extoll network switch, and multiple link ports.

The host interface was first implemented by the Hyper Transport (HT) core [77], which was optimized for low latency operations. However, since Intel systems do not support HT, the current version also provides a PCIe interface in which a translation layer, a PCIe-Bridge, is required.

To connect the functional units to the host interface and the functional units with each other, the HyperTransport Advanced Crossbar (HTAX) forms a dynamic network on chip.

The network interface part is formed by the Extoll crossbar [78], and the link ports (LP) [79]. The Extoll crossbar implements the switching and routing routines, which use a table-based routing algorithm.

Extoll provides the following five main functional units:

Virtual engine for low overhead The Virtual Engine for Low Overhead (VELO) [81, 78] is optimized for low latency data transfer and high message rates for small

[2] `http://www.extoll.de/`

messages up to 64 byte, which corresponds to the cache line size on X86-based processors. The VELO can also transmit larger messages bit with a decreased efficiency.

The Remote Memory Access The Remote Memory Access (RMA) [82] unit is used for larger message sizes. It supports one-sided put and get operations. tn can work with virtual and physical memory areas. A more detailed description of one-sided communication with Extoll is given below.

Address Translation Unit The Address Translation Unit (ATU) [83] is required to translate so-called Network-Logical-Addresses to the physical addresses for the RMA unit without involving the host CPU.

Shared Memory Function Unit The Shared Memory Function Unit (SMFU)[17] forwards load and store instructions to remote nodes. This unit can be used to create non-coherent, shared address spaces between multiple nodes.

Control and status These are registers that provide control information for all internal units. For example, they can be used to count the number of incoming requests to one unit. The register file can also be used to change settings like the node ID or routing tables [83].

One-sided communication in Extoll

This section gives a short overview of RDMA-communication in Extoll. A more detailed description can be found in [82].

In Extoll, the one-sided communication is performed by the RMA unit. Before a virtual memory region can be used as source or destination for a one-sided communication request, it must be pinned and registered. Then the address translation unit (ATU) of Extoll can perform the address translation without involvement of the host CPU. In contrast to Infiniband, Extoll does not use the virtual user space address and a key for identification of a virtual memory region, but the so-called Network Logical Address (NLA). The use of the NLA allows a low latency translation mechanism. The physical addresses are fetched from main memory or from the ATU-integrated Translation Lookaside Buffer (TLB). If a physical address is not stored in the TLB, only one access to main memory is required to get the physical address.

In contrast to Infiniband, the descriptor for a work request is directly written to the network device. To allow this from user space, a window within the Extoll Base Address Register (BAR) is mapped to the user space. A BAR is a memory window in the physical address space of the host system which is used for communication between CPU and the peripheral device. The RMA unit supports a transfer size between 1 byte and 8 MB with a single put or get command.

The RMA unit consists of three function units: The *requester*, the *responder*, and the *completer*. Every command first passes the requester unit. A *get* command is forwarded to the responder unit on the remote side. Every command is completed by the completion

Table 2.2.: Possible configurations for notifications[76, 80]

Command	Requester	Completer	Responder	
PUT	0	0	0	no notification
	1	0	0	Put with local completion
	1	1	0	Put with local completion and remote notification
GET	0	0	0	no notification
	0	1	0	Get with local completion
	0	1	1	Get with local completion and remote notification

unit, which usually writes the results to main memory. For a *get* command, the completion unit is passed on the active side, for a *put* command it is passed on the target side.

Each of the three function units can create a notification if it is successfully passed. For a *get* command, a local requester notification is created if the command passes the requester unit. On the remote side, the *responder* notification is created as soon as the data is successfully read from memory. The *completer* notification is created as soon as all data has been written to the destination address. For *put* commands, no responder notification is created. Table 2.2 shows which combinations of notifications are supported and useful for put/get operations. The Extoll device writes a notification to a pre-allocated buffer in host memory. The use of this notification allows an easy, explicit synchronization method for one-sided communication.

Atomic operations in Extoll

The RMA unit of Extoll also supports atomic operations,which are called *lock* operations. The idea of the lock operations is to enable efficient mapping of software locking algorithms in hardware. The lock operation performs a compound fetch-compare-and-add (FCAA) atomic operation.

In contrast to Infiniband, the *lock variables* are not located in registered and pinned host memory regions. Instead, the locks are located in their own *lock address space* [76].

Extoll software stack

Figure 2.31 shows the Extoll software stack. The *extolldrv* forms the PCI device driver. It performs the basic configuration and enumeration of the device and offers the possibility to get the physical und virtual addresses of the device resources (BARs) to the other device drivers. It also manages the interrupts.

The *extoll_rf* driver forms the interface for the Extoll register file. It implements a *sysfs* interface, a pseudo file-system of Linux, that provides user space access to driver level configuration and status. The registers of the device can be accessed by reading or

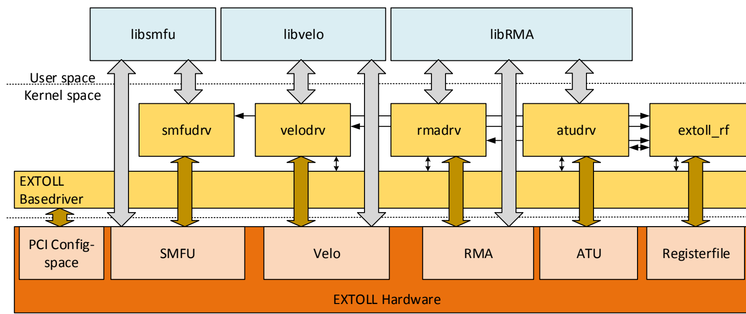


Figure 2.31.: Extoll software stack [76]

writing a file in the *sysfs* file system. The *smfudrv*, *velodrf*, *rmadrv*, and *atudrv* drivers handle the particular function unit. They also expose an interface to the user space.

The low-level user space library *libvelo* provides a user space API with all basic services available from the VELO; the *libsmfu* provides the interface for the SMFU. The *librma* API provides all services for the RMA-function unit. It also provides an interface to register user space memory to allow one-sided communication from user space. However, the memory registration is handled by the ATU-driver.

The user space libraries also provide direct access to the hardware by bypassing the operating systems. This is done by mapping device registers to the user space with memory mapped I/O (MMIO).

2.5. Communication between Distributed GPUs

In the previous sections, an introduction to communication models, interconnection networks, and GPUs was given. In this section, all this is brought together to discuss communication models for distributed GPUs.

In a GPU-accelerated application, the communication can either be controlled by the CPU or by the GPU, shown in Figure 2.32. If the CPU controls the communication, a hybrid-programming model is required. For the GPU, a GPU programming language like CUDA or openCL or a directive based approach like openACC is used, while for the inter-GPU communication, a communication library like MPI is required.

If the communication is controlled by the GPU, either the GPU must be able to control the network device or an underlying communication-framework is required. In this framework, the CPU accepts communication requests from the GPU and forwards them to the hardware. This case is called *host-assisted*. However, the control flow stays on the GPU and therefore we count this to the GPU-controlled class. If the communication is controlled by the GPU, the communication must be described in a GPU programming language like CUDA. If the GPU directly controls the network device, a communication between the two devices is required. This is referred to as device-to-device communication.

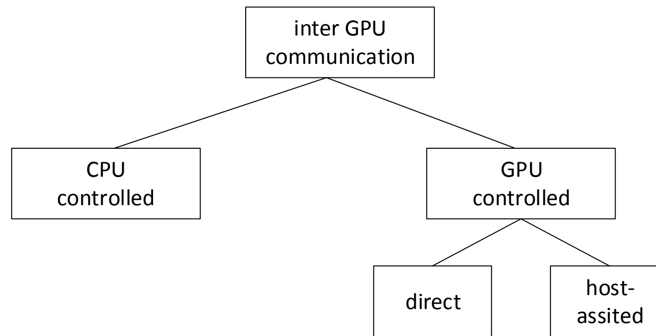


Figure 2.32.: Communication Control for inter-GPU communication

Figures 2.33 and 2.34 on the facing page show the control flow of an application for both approaches. For the hybrid approach, the control flow reverts from the GPU to the CPU to synchronize the computational tasks on the GPU with the communication tasks on the CPU.

This approach allows very good overlapping of communication and computation, as the communication is *offloaded* to the CPU. The communication overhead on the GPU is equal to zero. However, the synchronization between CPU and GPU adds some overhead and therefore increases the latency of the whole application. Furthermore, a CPU thread is required to control the GPU and GPU-related communication.

If the GPU controls the communication, synchronization between GPU and CPU can be avoided. A CPU thread is only required to start a single initial kernel on the GPU which then handles communication and computation. The CPU thread can be used for other work or set to sleep. On the other hand, this approach adds communication overhead to the GPU. This overhead may also slow down the performance of a GPU application.

Figure 2.35 shows the design space for communication methods between distributed GPUs. If the communication is host-controlled, one-sided and two-sided communication is supported, whereas remote loads and stores are only supported for GPU-controlled communication. A remote load/store communication is based on copying data between local and remote memory regions with load and store instructions. From the CPU point of view, GPU memory regions are always remote.

All communication methods have advantages and disadvantages. While a lot of research was done on optimizing CPU controlled communication, GPU controlled communication has, up to now, rarely been examined. There are several reasons for this. The GPU memory was not accessible for other peripheral devices as NICs and the data-parallel programming model for GPUs does not fit in parallel programming APIs like MPI.

Recent technologies like GPUDirect RDMA and dynamic parallelism help to overcome some of these limitations. Therefore, GPU-controlled communication requires furthe, more detailed consideration.

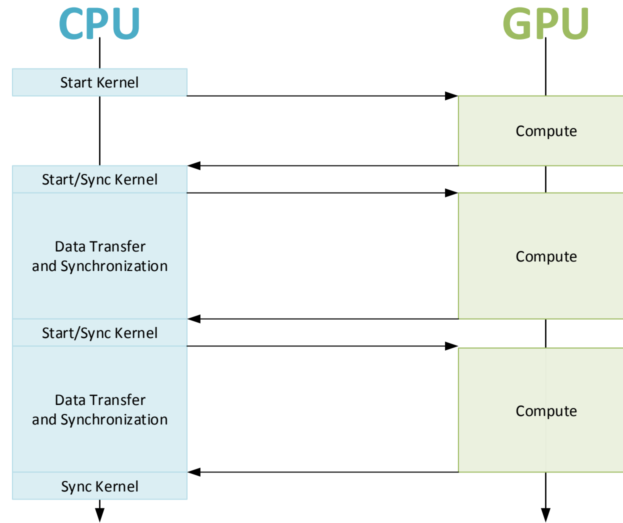


Figure 2.33.: Workflow of a GPU program, when communication is controlled by the host

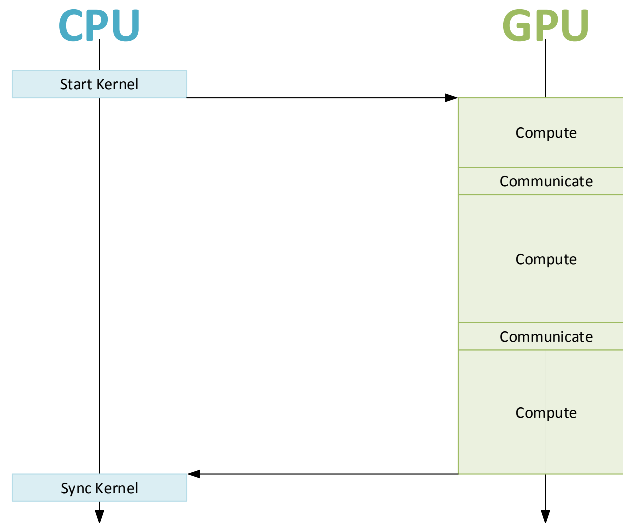


Figure 2.34.: Workflow of a GPU program when communication is controlled by the GPU

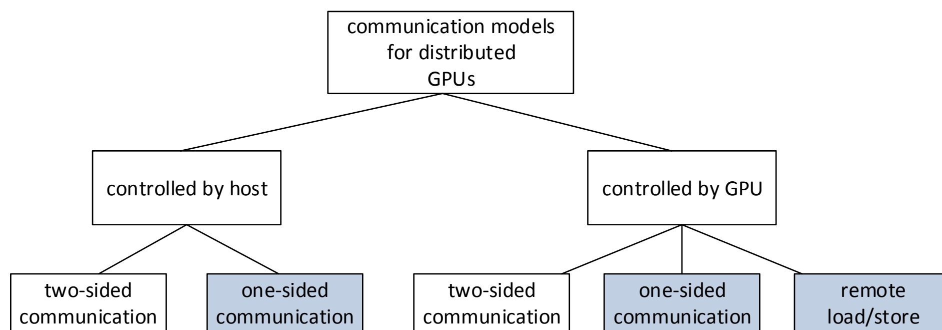


Figure 2.35.: Communication methods for distributed GPUs

The next section applies to the first problem, the *data transfer* between distributed GPUs and the direct access to GPU memory from a peripheral device. The subsequent sections deal with the different communication approaches for distributed GPUs. The focus will be on one-sided communication. CPU-controlled two-sided communication was handled in a lot of previous work and will be used for comparisons. Due to the use of RDMA-capable interconnection networks, a two-sided communication framework for a GPU can be implemented on top of a one-sided communication framework, probably using the *eager* or *rendezvous* protocol. As a communication model that may be more in line with the data-parallel programming model of GPUs, another focus will be communication using remote load/store instructions.

3. Direct Data Transfer between GPUs

One important factor of efficient communication in heterogeneous systems is the data transfer, meaning in which way a data package is transferred between the memories of two GPUs. The data transfer has always been one of the main bottlenecks for distributed GPU computing, as shown, for example, in [84, 85, 86]. In the past, often several copies in host memory were required to transfer the data between two GPUs.

Recent technologies like GPUDirect RDMA help to overcome some of these limitations since they allow a peripheral device direct access to GPU memory. This direct access is also one condition for one-sided communication between distributed GPUs.

In this chapter, an overview about data transfer methods between distributed GPUs is given. We describe how GPUDirect RDMA support is introduced to the RMA unit of the Extoll device and Infiniband. The performance of a direct data transfer is compared with previous methods of data transfer between distributed GPUs and the assets and drawbacks of different techniques are discussed.

3.1. Inter- and Intra-Node Data Transfer

One important factor of data transfer between GPUs is the relative location of the GPUs to each other. GPUs can either be located on the same node or on different nodes. If the GPUs are located on the same node, data transfer can be routed through the interconnection network of the host system, for example, through the PCIe-bus or the QPI-link. This is referred to as *intra-node* data transfer. If the GPUs are located on different nodes, the data has to be routed through the interconnection network of the cluster. This also means that the data is transferred between the GPU and the NIC. This is referred as *inter-node* data transfer.

This work primarily focuses on inter-node data transfer and communication, and thus on the data transfer between GPU and the NIC. However, as some of the knowledge of the intra-node communication is also useful for the inter-node communication, some of the aspects of intra-node data transfer will be explained in more detail.

3.2. Data Transfer Methods

Figure 3.1 shows the design space for data transfer between distributed GPUs. Data transfer between GPUs can either be *direct* or *staged*. Direct means that the data is directly transferred between the memories of two GPUs, whereas a *staged* transfer uses one or more copies in host memory.

A *staged* transfer can be either *sequential* or *pipelined*. In a sequential transfer, the data is completely transferred to one buffer before the transfer to the next buffer is started.

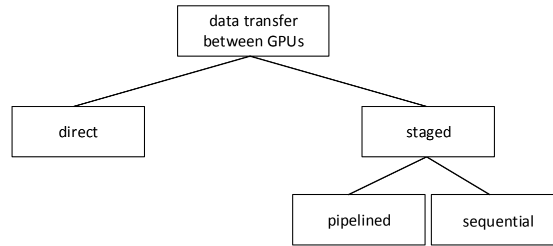
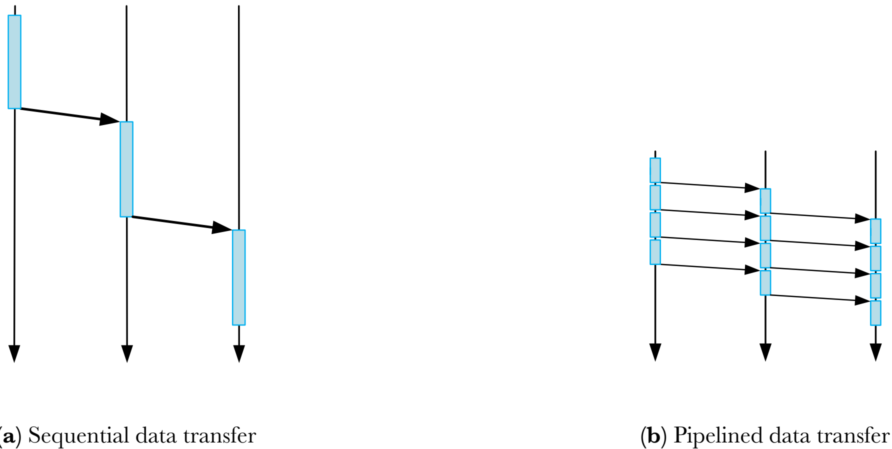


Figure 3.1.: Data transfer methods between distributed GPUs



(**a**) Sequential data transfer (**b**) Pipelined data transfer

Figure 3.2.: Sequential and pipelined data transfer

In a pipelined transfer, the data is transferred in chunks. As soon as the first chunk is transferred to a buffer, the transfer to the next buffer is started. Figure 3.2 shows the difference between both approaches. At first sight, a pipelined data transfer allows a faster data transfer between source and destination. Still, a staged data transfer requires a CPU thread to control the data flow between host memory and GPU memory buffer and adds more overhead to the CPU. Therefore, the size of the chunks should be carefully selected.

A staged transfer between two GPUs requires a two-sided communication scheme. On both sides, a host thread is required to control the data flow. A direct data transfer supports both one-sided and two-sided communication paradigms. Remote loads and stores require also a direct data transfer.

A direct data transfer between two GPUs seems to be the best solution: copies in host memory are avoided, the communication overhead is minimized, and one-sided communication is supported. To allow direct inter-node data transfer with RDMA-capable devices, the NIC must be able to read and write GPU device memory. However,

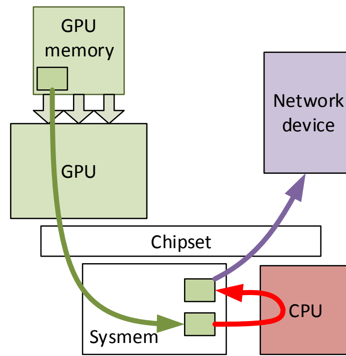


Figure 3.3.: Data transfer without GPUDirect 1.0

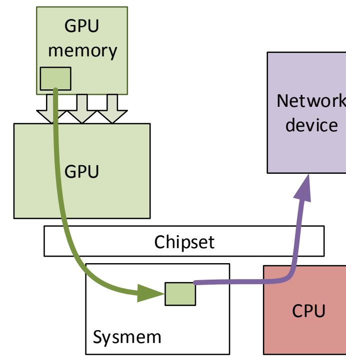


Figure 3.4.: Data transfer with GPUDirect 1.0

GPU memory normally is neither directly accessible from host the nor from the network device. Only recent technologies enable support for this.

To improve staged and allow direct data transfer between distributed GPUs, several optimization strategies for Nvidia GPUs were introduced in the past. These techniques are referred to as *GPUDirect technologies* and are now explained in more detail.

3.2.1. GPUDirect 1.0

The first step to a more efficient data transfer between GPUs is called GPUDirect 1.0. It was introduced in 2009 with CUDA 4. This technique allows a GPU and another peripheral device to share a locked memory region in host memory.

Before GPUDirect 1.0, *two* copies in host memory on both sides were required to transfer the data between distributed GPUs, as shown in Figure 3.3. The GPU and the network device uses different buffers in host memory, so the data has to be copied between this two host memory buffers. This adds also a lot of overhead to the CPU since the CPU was responsible for copying data.

GPUDirect 1.0 was first developed for Infiniband network devices and helps to improve the performance of several applications up to 40% [87]. To allow this, changes in the Linux operating system as well as in the device drivers of the GPU and the network device were required. The support of shared pinned pages was added to the Linux kernel. Both the Nvidia driver and the Infiniband network driver were enabled to use these shared pinned pages. The Extoll device driver uses a different memory pinning strategy than Infiniband [76], therefore, no changes in the Extoll RMA and ATU drivers were required to support GPUDirect 1.0.

Using GPUDirect 1.0 technology, the actual data transfer can be offloaded to the network device and the DMA engines of the GPU. The GPU DMA engine copies the data between host and GPU while the network device transfers the data between the host memory buffers. However, the CPU is still required to control the data flow between

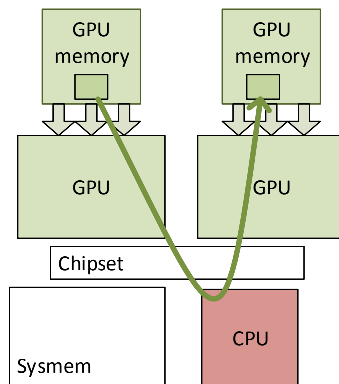


Figure 3.5.: Data transfer between two GPUs on the same host with GPUDirect peer-to-peer

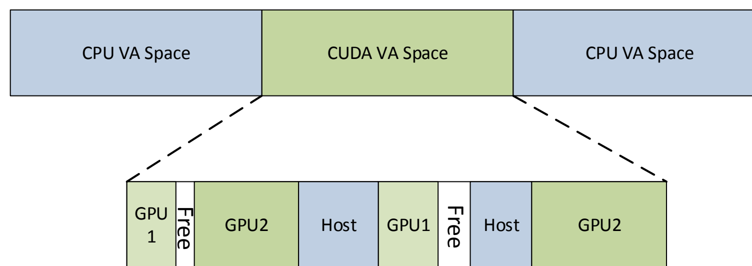


Figure 3.6.: Unified virtual address space (UVA)

the GPU and the network device. For this, a thread is required on both sides, so only two-sided communication is supported.

3.2.2. GPUDirect peer-to-peer

With CUDA 4.5, GPUDirect peer-to-peer was introduced. GPUDirect peer-to-peer allows a direct data transfer between two GPUs on the same host, without any copies in host memory, as shown in Figure 3.5. The data is directly routed through the PCIe-bus.

This technique also allows a GPU a direct access to the memory of another GPU with load and store instructions within a CUDA kernel. This development came simultaneously with the introduction of the *Unified Virtual Address Space (UVA)* in CUDA. The UVA creates one virtual address space for all GPUs in one compute node. Host memory that is allocated with `cudaMalloc` also lies in this virtual address space, as shown in Figure 3.6. A GPU kernel can use pointers to this virtual address space to directly access host memory or the memory of another GPU on the same host node. The memory controller of the GPU recognizes if a virtual address belongs to the local GPU, a remote GPU, or host memory and forwards the access to the right physical address.

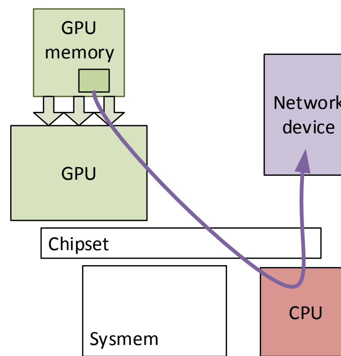


Figure 3.7.: Data transfer between GPU and the network device, using GPUDirect RDMA

This also allows GPU-controlled communication between GPUs on the same host by using load and store instructions. Before this, the only possibility to transfer data between two GPUs on the same host were `cudaMemcpy` operations with staged buffers in host memory, called from a CPU thread.

However, the Nvidia peer-to-peer technique is not open for third party devices, like network interface cards, since it uses a prohibitive protocol and requires special support in the network hardware.

The APEnet+ project [88] applied the protocol to the APEnet+ FPGA network device [89]. While the performance for RDMA-write accesses was quite good, the peer-to-peer reading protocol was difficult to implement. The work showed that using a direct protocol between GPUs minimizes the communication overhead on the CPU and provides better overlapping efficiency. The peer-to-peer protocol seems to be especially effective for latency-sensitive applications. In later work [90], they showed that for APEnet+, the protocol is useful for messages up to 128 kb. For larger messages, a staged protocol provides a lower latency and a higher bandwidth for inter-node GPU to GPU data transfers.

3.3. GPUDirect RDMA

To overcome the limitations of GPUDirect peer-to-peer GPUDirect RDMA was introduced, in 2012, with CUDA 5. This technique allows RDMA-capable network devices to directly read and write GPU device memory, as shown in Figure 3.7. This technique requires no changes in the network interface hardware but in the device drivers and user space libraries of the NIC. In the following, a more detailed description of this technique and the required adaptions in the device drivers is given.

Normally, GPU memory is only accessible with load and store instructions from the GPU. From host, the virtual GPU memory pointer can be used with CUDA related functions like `cudaMemcpy`, but direct read- or write-accesses result in a segmentation

fault. The GPU memory is not directly visible from the host, since it has no address in the physical address space of the host system.

Therefore, to allow a CPU thread or an RDMA capable device to access GPU memory, the memory must be accessible over an address in the physical address space of the host system. This is enabled with GPUDirect RDMA technology. It allows a translation from a *virtual* GPU device address to an address in the *physical address space* of the host system.

This is done by mapping a part of the GPU device memory to addresses within one of the Base Address Register (BAR) of the GPU. The BAR is a memory window in the physical address space of the host system which is normally used for communication between the CPU and the peripheral device.

From a peripheral device point of view, it makes no difference if a physical address points to host memory or to a BAR of another device. Therefore, to enable support for GPUDirect RDMA, the user space libraries and device drivers must support a virtual to physical address translation from a GPU device pointer to the physical addresses within the GPU BAR.

3.3.1. Nvidia GPUDirect Interface

In the first version, the NVIDIA-GPUDirect Interface consists of two parts, one in the user space and one in the kernel space. In the user space, the CUDA-function `cuPointerGetAttribute` is required to determine two tokens, *p2pToken* and *vaSpaceToken*, which are necessary to uniquely identify a GPU virtual address. The function requires a GPU device pointer as input. In the kernel space, the GPU kernel function `nvidia_p2p_get_pages`requires these two tokens and the virtual GPU memory address. The function pins the GPU memory and maps it to the BAR. The function returns a table with the BAR addresses belonging to the virtual GPU memory address.

The memory registration function of the NIC has to forward these two tokens to the kernel space. Since these tokens are not required for a normal host address, either the user space memory registration functions must be adapted or two different memory-registration functions, one for GPU memory and one for host memory, must be implemented. Therefore, changes in the user space and in the kernel space are required.

This was changed is the latest version of the GPUDirect RDMA. For the `p2p_get_pages` function-call, the tokens can be set to zero and the user space part is not required. Therefore, the latest version only requires changes in the device drivers of the network device; the user space part can remain unchanged.

3.3.2. Mellanox GPUDirect RDMA support

The GPUDirect technology was first officially supported on Mellanox Infiniband devices. To support GPUDirect RDMA technology, a special device driver, *nv_peer_mem* is required. In the following, the functionality of this driver is explained in more detail.

The latest version of the Mellanox Infiniband device drivers, part of the OFED-Stack 2.5, allows the registration of so-called *peer-memory agents*. A peer-memory agent is a spe-

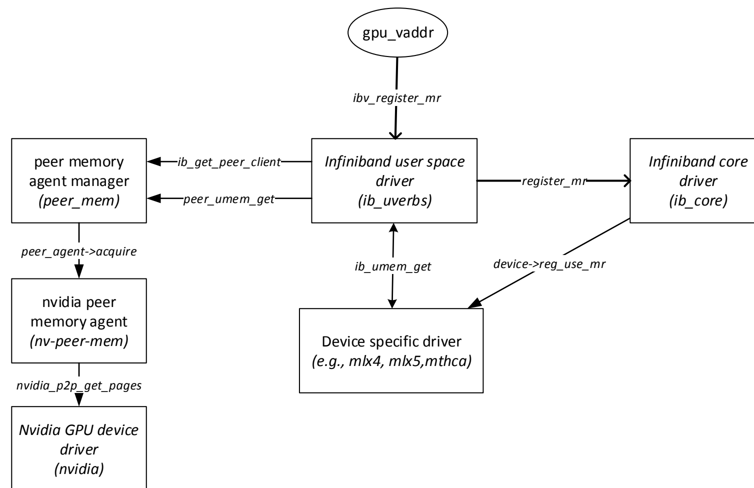


Figure 3.8.: Mellanox GPUDirect RDMA support, memory registration

cial device driver which allows the registration of so-called *peer-memory regions* for Infiniband devices. Peer-memory regions are regions that can be registered for an Infiniband HCA but are not located in the user space memory. A peer-memory agent must provide several functions to pin and unpin memory, to create page tables for peer-memory addresses andm to check if a virtual address belongs to this peer-memory agent. Figure 3.8 illustrates, how a virtual address, belonging to a GPU memory, is registered using the Mellanox peer-memory driver.

If a host process tries to register a user space memory region, the virtual address is forwarded to the Infiniband user space driver, *uverbs*. This driver is the interface for host processes to the Infiniband kernel stack. The user space driver forwards the address to the core diver, which forwards the registration request to the device specific driver. This driver eventually calls the function `ib_get_umem`, which is also part of the Infiniband user space driver. This function is responsible for handling user space memory registration.

The registration function forwards the virtual address to the registered peer-memory agents (`ib_peer_mem_get`). The peer-memory agent now checks if the virtual memory address belongs to this agent (*acquire*). If so, a pointer to the peer-memory agent is returned. If no fitting peer-memory agent is found, the memory is handled as normal user space memory.

The Nvidia peer-memory driver checks the affiliation of an address by trying to pin this memory region with `nvidia_p2p_get_pages`. If the pinning is successful, the virtual address belongs to GPU memory and, therefore, the function returns a success. However, before the affiliation testing function returns, the pinned GPU memory is released with `nvidia_p2p_put_pages`.

Later, the Infiniband user space driver pins the memory by calling the `peer_umem_get` function of the peer-memory agent. For the Nvidia drivers, again `nvidia_p2p_get_pages`

is called. The returned page table is now used to create the memory translation table (mtt) for the Infiniband HCA, with addresses pointing to the BAR of the GPU.

After registration, a GPU memory region can be used like a host memory region for Infiniband communication functions, with one exception. The Infiniband specification allows the flag `IBV_SEND_INLINE` for write and send-operations. Using this flag, the CPU copies the data payload directly to the Infiniband work request. In this case, the network device reads the data payload from the work request and not from the memory buffer.

Since GPU memory is not directly readable from host, using this flag results in a segmentation fault for GPU memory regions. Therefore, the inline flag should be avoided for GPU memory segments.

3.3.3. Host mapped GPUDirect RDMA support

To support GPUDirect technology for the Extoll device, another approach was implemented. This approach is more general and does not require special peer-memory agents. We describe the implementation for the Extoll device, however, this overall approach can be used to allow any kind of DMA capable devices access to GPU memory. It also allows direct access to GPU device memory from the host, which is a pleasant side effect.

The approach works in three steps:

1. Pin the GPU memory and map it to the GPU BAR using GPUDirect RDMA
2. Map the BAR addresses to the user space using memory mapped I/O (MMIO)
3. Register the MMIO address for the DMA capable device

The individual steps are shown in Figure 3.9 and now explained in more detail. To pin GPU device memory and map it to the user space a special device driver, *gpumap*, and a corresponding user space library, *gpumap-lib*, were developed.

The GPU memory pointer (`gpu_vaddr`) is handed over to the gpumap library function `map_gpu`. Here, the two tokens, *p2pToken* and *vaSpaceToken*, are determined. This step can be omitted in the newest version of GPUDirect RDMA. To pin the GPU memory, the tokens and the device pointer are forwarded to the *gpumap* driver, see Figure 3.9.

The *gpumap* driver calls the nvidia driver function `nvidia_p2p_get_pages.` This function pins the GPU memory and maps it to the BAR of the GPU.

In the next step, these BAR-addresses are mapped to the user space with Memory Mapped I/O (MMIO), using the mmap-function of the *gpumap driver*. This way, the GPU-memory is mapped into the user space (cf. Figure 3.9) with a new virtual address (`host_vaddr`) which allows the host to direct access to GPU memory with simple load and store instructions.

However, since the memory is now accessible with two different virtual addresses, with `gpu_vaddr` from the GPU and with `host_vaddr` from the host, this approach may require some address translation in the higher level communication libraries.

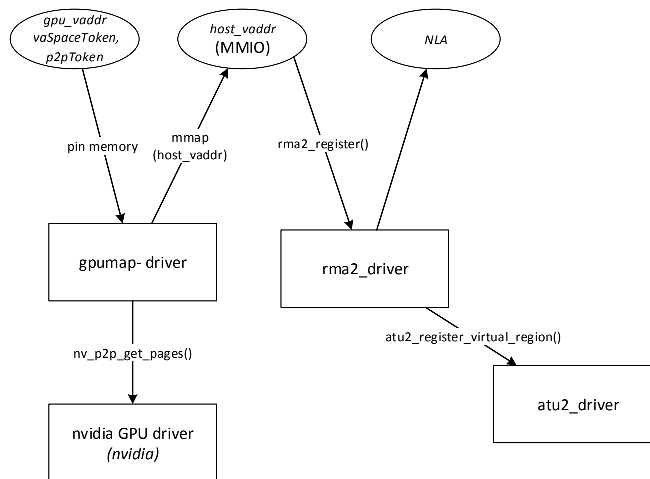


Figure 3.9.: Pinning and mapping of GPU memory using GPUDirect RDMA and memory mapped I/O

To allow one-sided communication operations with the RMA unit of the Extoll device, the mapped GPU-memory must be registered for the network device. Therefore, the MMIO address is handed over to the RMA memory registration function, which eventually forwards the address to the *atu driver*. To pin host memory and determine the physical page addresses to create a page table, the device driver normally calls the kernel function `get_user_pages`. This function locks the memory and returns a table with the physical memory pages.

However, this function fails for MMIO-addresses, as these addresses do not provide a page table. Therefore, a driver patch was developed that determines the physical addresses belonging to an MMIO address and creates a page table with these addresses. Figure 3.10 shows the functionaltiy of the patch.

1. Before `get_user_pages` is called, the virtual address is handed over to the the Linux kernel function `find_vma`. The virtual memory area (vma) is the structure that Linux uses to manage virtual memory regions. It provides the page tables and is required to perform virtual to physical address translations. If no area is found, an error is returned.

2. If a virtual memory area is found, it is checked if the `VM_IO` and `VM_PFNMAP` flags are set. These flags mark MMIO memory regions. If the flags are not set, the virtual address probably belongs to a normal user space memory area and the memory registration function is continued as usual by calling `get_user_pages`.

3. If the MMIO flags are set, the Linux kernel function `follow_pfn`is used to determine the physical addresses. To create a page table for MMIO addresses, the

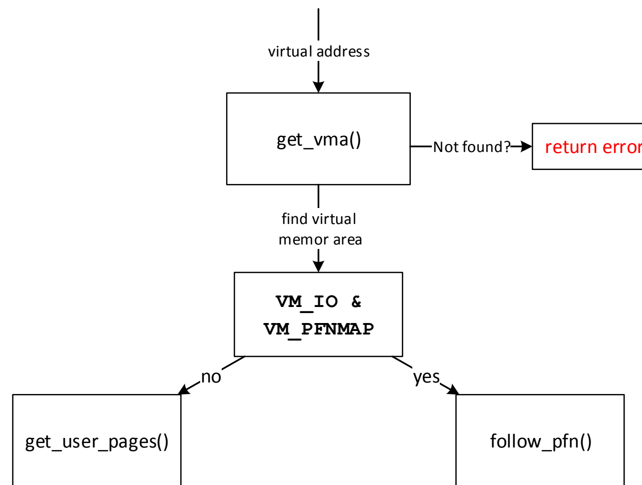


Figure 3.10.: Registration of MMIO addresses

Listing 3.1: Stepwise creation of the page table for MMIO addresses

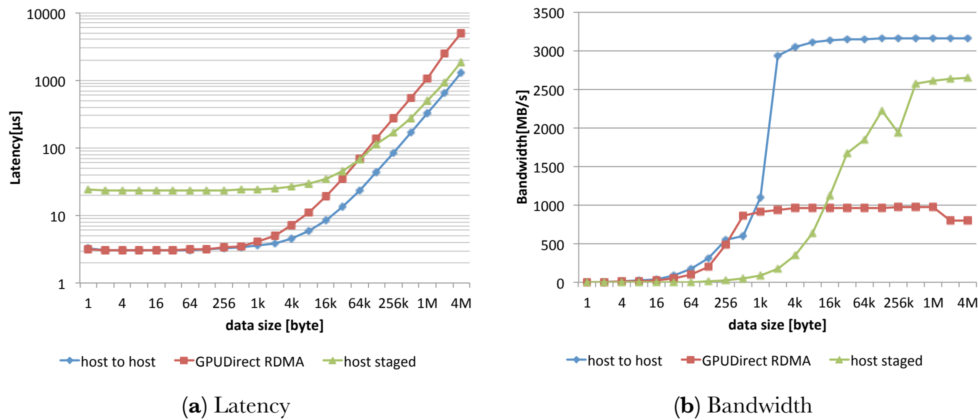
```
unsigned int mapped = 0;
int i =0;                                                      2
uint64_t physical_address = 0, curr_addr = virt_address;
while(mapped <length){                                         4
  follow_pfn(vma, curr_addr, &physical_address);
  pages[i++] =physical_address<<PAGE_SHIFT;                    6
  mapped+=PAGE_SIZE;
  curr_addr+=PAGE_SIZE;                                        8
}
```

physical addresses are determined, as shown in Listing 3.1. This has to be done in steps of the memory page size to *pretend* a real page table.

The result of this patch is the same as that of the Mellanox Infiniband patch: a page table with the GPU-BAR addresses. If the memory registration time is neglected, the latency and bandwidth results of GPUDirect RDMA are not affected by the way the support for GPUDirect is attached to the device drivers. The performance results in the following sections neglect the pinning time.

3.4. Performance Results of GPUDirect RDMA

In this section, we evaluate the performance results, using GPUDirect RDMA technology for Infiniband and the RMA unit of the Extoll device. We do not directly compare the Infiniband and the Extoll device, as the Infiniband device uses an ASIC with a peak



(**a**) Latency (**b**) Bandwidth

Figure 3.11.: GPUDirect RDMA performance of Infiniband

bandwidth of 32 Gbps while EXTOLL is implemented on an FPGA with a peak bandwidth of 9.8 Gbps.

3.4.1. Latency and bandwidth

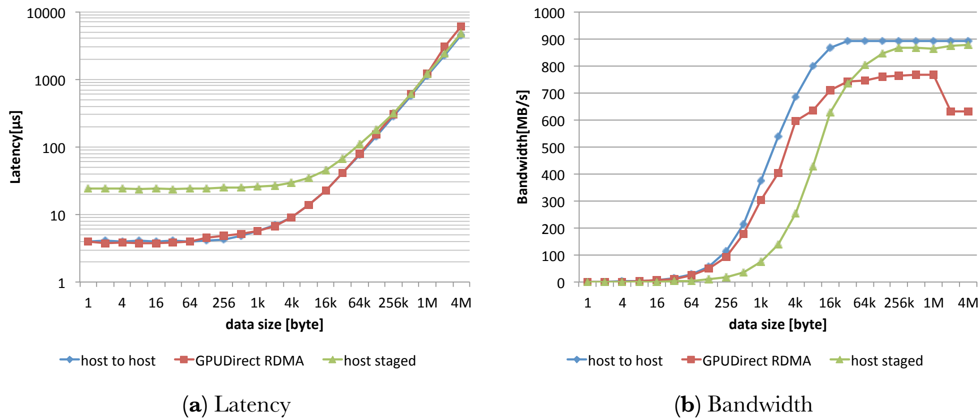
The latency test uses a simple pingpong benchmark. To synchronize the source and the target side, we use the `RDMA_WRITE_WITH_IMM` operation for Infiniband. On Extoll, we use the completer-notification for RMA put operations. By this, polling on GPU memory from host can be avoided. For the bandwidth tests, multiple remote write requests are posted subsequently to the hardware before they are synchronized. We use remote write operations for both benchmarks.

In order to assess the capabilities of the GPUDirect RDMA technology, we compare the results with previous methodologies. We use GPUDirect v1.0, so one host copy is required on both sides, but the network device and the GPU use the same pinned host memory buffer. To optimize the performance of this data transfer, we use a pipelined transfer protocol. On the sourcing side, the GPU copies the data to host memory in blocks with asynchronous CUDA copy operations. The network device starts to transfer the data as soon as the first block is completely copied to host memory. On the receiving side, the GPU starts to transfer the data from host to the GPU as soon as the first data block has arrived.

For Infiniband, we used two workstations, each equipped with two Xeon X5660 six core CPUs and one K20c Nvidia GPU, linked up with Mellanox Connect 3X QDR Infiniband.

For Extoll, we used two workstations with two Intel Xeon E5-2630-six core CPUs and also one K20c Nvidia GPU. We used the 32 GB Galibier Card, which provides a PCIe interface and runs at 157 MHz.

Figure 3.11 shows the results for Infiniband and Figure 3.12 for the Extoll device. For comparison, we also represent the latency for a host-to-host transfer with the used hardware.



(**a**) Latency (**b**) Bandwidth

Figure 3.12.: GPUDirect RDMA performance of the RMA unit of the Extoll device

For small messages, the use of GPUDirect RDMA clearly outperforms the staged data transfer. Bandwidth and latency are very close to the performance of a host-to-host data transfer. The result is different for larger data sizes.

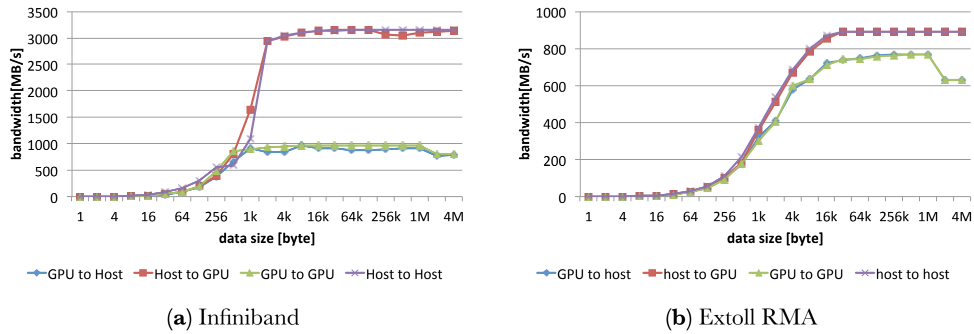
Here, the host-to-host data transfer clearly outperforms a GPU to GPU data transfer. Even a staged data transfer, using host memory buffers on both sides, outperforms the direct data transfer using GPUDirect RDMA. Another interesting observation is that for a data size larger than 1 MB, the bandwidth of a direct GPU to GPU data transfer slows down for both Extoll and Infiniband.

3.4.2. PCIe peer-to-peer performance

To get to the bottom of this, in the next step, the different directions for a data transfer between GPU and the NIC are examined. Thus, the bandwidth for data transfers between host and GPU over the RDMA interconnect are considered. In this case, the GPU memory can either be the source or the destination of the data transfer.

Figure 3.13a shows the results for the Infiniband device and Figure 3.13b for the Extoll device. The maximal bandwidths for the different directions are also summarized in Tables 3.2a and 3.2b. These results show that the bandwidth is slowed down if GPU memory is the data source. In this case, the network device has to read the data from the GPU. This effect is clearer for Infiniband since the maximal bandwidth of the Infiniband ASIC is higher. However, it is also noticable for the Extoll device.

The PCIe-bus causes this low performance. While posted peer-to-peer operations are well supported on many chip-sets, non-posted peer-to-peer operations are only poorly supported. This issue was also reported in other works, using GPUDirect RDMA technology [91] or using the XeonPhi accelerator card [92] and direct data transfer between the XeonPhi and an Infiniband card. Because of this limitation, the Nvidia GPUDirect peer-to-peer, which allows direct data transfer between GPUs on the same nodes, only uses peer-to-peer write operations[89].



(**a**) Infiniband (**b**) Extoll RMA

Figure 3.13.: Bandwidth using different transfer directions

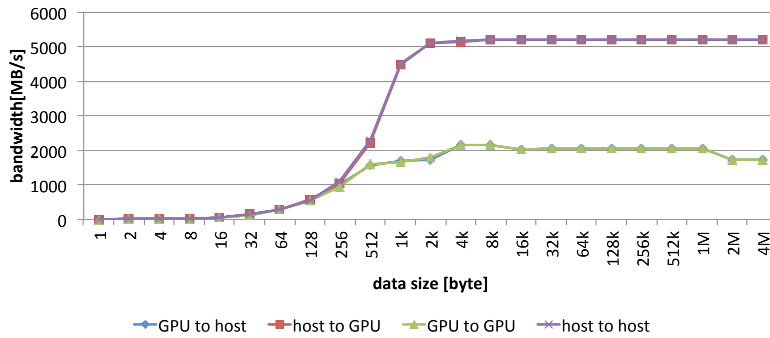


Figure 3.14.: Bandwidth using different transfer directions on an Ivy Bridge machine

3.4.3. Intel Ivy Bridge

The data in Figures 3.13a and 3.13b are measured on a machine with an Intel Sandy Bridge processor and chipset. On newer machines, using the Intel Ivy Bridge processor and chipset, the support for non-posted PCIe peer-to-peer accesses is better, though not ideal.

Figure 3.14 shows the bandwidth for GPU transfers on an Ivy Bridge machine. Again, the data is transferred between two K20c GPUs on different nodes. We use two machines with two 10-core Intel Xeon CPU E5-2690 v2 CPUs and Mellanox FDR Infiniband.

Table 3.1.: Maximal bandwidth in MB/s using GPUDirect RDMA

	Destination	Host	GPU
Source	Host	3212	3212
	GPU	968	969

(**a**) Infiniband

	Destination	Host	GPU
Source	Host	892	892
	GPU	769	767

(**b**) RMA

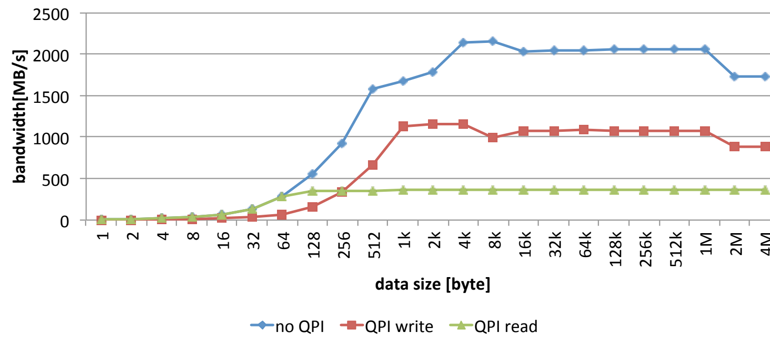


Figure 3.15.: Bandwidth on an Ivy Bridge machine over the QPI link for different directions

The results show that the maximal bandwidth for non-posted peer-to-peer data transfers on an Ivy bridge machine is approximatly 2.2 Gb/s. This is better than on Sandy Bridge machines. However, the performance is still much lower than the performance for host to host transfers using Infiniband FDR.

3.4.4. Inter I/O-hub data transfer

So far, all data were collected on machines where the GPU and the NIC are connected to the same socket and therefore use the same PCIe root complex. If the GPU and the network device are not located on the same socket, the data transfer have to be routed through the QPI (Intel) or HT (AMD) link.

On machines using a Sandy Bridge chipset, this is not supported. The I/O-hub does not forward peer-to-peer accesses over the QPI link [93]. Therefore, neither GPUDirect peer-to-peer nor GPUDirect RDMA is supported between devices located on different sockets using these chipsets.

On machines using the Intel Ivy Bridge chipsets, peer-to-peer request are forwarded, but the maximal bandwidth is very low. Figure 3.15 shows the reachable bandwidth if the data have to be routed through the QPI link.

In the *no QPI* case, both GPUs are located on the same socket as the Infiniband device. This result corresponds to the result in the previous section for a GPU to GPU data transfer. For the *QPI-write* graph, the destination GPU is located on a different socket than the Infiniband network device. In this case, the Infiniband device has to write the data to the destination GPU over the QPI link. For the *QPI-read* graph, the source GPU is located on a different socket than the Infiniband device. In this case, the Infiniband device has to read the data from the GPU over the QPI ink.

If the Infiniband device writes the data over the QPI link, the bandwidth is slowed down to maximal 1 GB/s. If the Infiniband device reads the data from the GPU over the QPI link, the bandwidth is even further slowed down to 350 MB/s. An interesting observation is that for small messages, up to 256 byte, the *QPI-write* performance is worse than the *QPI-read* performance.

Still, to reach maximal performance, the GPU and the network device should be located on the same socket. If this is not possible, staged copies for larger messages should be used to reach optimal performance.

3.5. Summary

In this chapter, we have shown that a direct data transfer between distributed GPUs is enabled by mapping a part of the GPU memory to one of the BARs of the GPU. Since these BARs are located in the physical address space of the host system, another peripheral device can handle these addresses like physical host memory addresses.

The performance results for GPUDirect RDMA show that a direct data transfer between GPUs brings performance benefits for small- and medium-sized data transfers. For larger data transfers, the benefits are limited due to limitations of the PCIe-bus. Since newer machines already show some improvements here, it can be expected that this will change in future systems. The same applies for current performance issues for the QPI link.

However, we should bear in mind that currently a direct data transfer has some performance limitations. For optimal performance, GPU and network device should be located on the same root complex, otherwise the performance is scaled down.

4. Host-controlled GPU-to-GPU Communication

In the previous chapter, different data transfer methods between distributed GPUs were discussed. This and the following chapters focus on communication control. A data transfer, whether staged or direct, has to be initiated and synchronized. To avoid race conditions and to guarantee data consistency, basic synchronization methods between source and destination of a data transfer must be provided. In the second chapter it was stated that on a heterogeneous system, either the GPU or the host CPU can control the communication. This chapter elaborates on CPU-controlled communication.

If the CPU controls the communication between distributed GPUs, a hybrid- programming model is required. In this case, the GPU is only used for computation while the CPU manages the communication and the data transfer. For this, a CPU-based communication library or language is needed. The data can be transferred directly or staged and the communication can either be one- or two-sided.

The advantage of a hybrid model is that it adds no overhead to the GPU due to communication functions. Also, often only a few or no changes in the GPU code are required to distribute a computation job from one GPU to a cluster of GPUs, because the CPU handles the communication and the data distribution.

A good communication API should allow overlapping of communication and computation. The communication time consists of the data transfer time and the communication overhead, for example, synchronization, tag matching, and data buffering. While the data transfer can be offloaded to the network device, the overhead blocks the CPU. Therefore, the communication cannot be completely overlapped for pure CPU-applications.

On GPU-accelerated applications, however, the communication overhead on the CPU can be overlapped with the computation on the GPU. If the communication time is smaller than the computation time, the communication can be overlapped completely. This is well implemented in many GPU-accelerated applications by using CUDA streams and asynchronous memory transfers [94, 7].

However, using more GPUs on the same job size (strong scaling) often means that the computation work for ever GPU declines while the communication overhead stays constant or rises. At a certain point, the communication time exceeds the computation time and can no longer be completely overlapped. Data transfer between distributed GPUs is more expensive than data transfer between distributed host memory regions, as shown in the previous chapter. Thus, GPU accelerated applications often show a worse strong scaling [7] than not accelerated applications.

Listing 4.1: Using MPI for data transfer between GPUs, no CUDA-aware MPI

```
/*on Sender side*/
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);             2
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
/*on Receiver side*/                                                  4
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice)               6
```

Listing 4.2: Using CUDA-aware MPI

```
/*on Sender side:*/
MPI_Send(s_buf_d, size,···);                                          2
/*one Receiver side*/
MPI_Recv(r_buf_d, size,···);                                          4
```

Therefore, the communication between two GPUs must be as efficient as possible, and a one-sided communication model can help to improve the strong scaling capabilities of GPU accelerated applications.

In this chapter, GPUs are integrated into the GASPI specification and the implementation of this specification, GPI-2. This is an example for one-sided, CPU-controlled communication for distributed GPUs. We evaluate the benefits of this model by comparing the performance to an optimized CUDA-aware MPI version as state-of-the-art. A part of this work was already published in [95].

4.1. Related work

Since GPU computing becomes more and more important for high-performance computing, the support of GPUs was integrated in many communication libraries and APIs. This section gives a short overview of exiting approaches.

4.1.1. CUDA-aware MPI

The most common way to utilize a hybrid cluster is a combination of CUDA and MPI and a lot of work has been done in optimizing *CUDA-aware* MPI versions [96].

CUDA-aware MPI means that the MPI version supports pointers to GPU device memory, as shown in listing 4.2. Otherwise, the data has to be copied explicitly between GPU device memory and host memory, as shown in listing 4.1. This can be optimized by using a pipelined protocol, but this requires extra effort from to the programmer.

However, using CUDA-aware MPI does not mean that the data is directly transferred between the GPUs, but that the MPI runtime system handles the underlying data transfer, which can be both direct or staged.

Even if the data are internally buffered, a CUDA-aware MPI version not only provides better programmability but also improves the performance, especially the latency of GPU to GPU data transfers [97].

If MPI is only used for host-to-host data transfers, the step of moving data back to the GPU may get delayed, even if the data has been received in host memory, as the MPI library may handle other incoming messages before the receive function returns.

Currently, two widely used CUDA-aware MPI-versions exist: OpenMPI[98] and Mvapich2 [99]. The Ohio State University developed a highly optimized MPI version for Infiniband clusters, Mvapich2 [100], and some work was done on integrating GPUs into this framework. In [101], Wang et al. introduce Mvapich2-GPU, a CUDA-aware MPI library. Internally, the data transfer is pipelined through the host and the features of GPUDirect 1.0 are used. In [102], the intra-node communication is optimized by using GPUDirect peer-to-peer and Nvidia inter-process communication. In further work, global all-to-all operations on GPU memory [103] and non-continuous data transfers[104] were optimized. The interesting point of the non-continuous data transfers is that GPUs are used to pack and unpack non-continuous data. Here, the GPU is used to *support* the communication.

In [91], the inter node communication in Mvapich2 1.9 was optimized by using GPUDirect RDMA technology. Normally, in MPI, the *eager* protocol is used for small messages. The sender transfers data to pre-allocated buffers. The receiver polls on flags, which are part of the transferred data. However, this technique is not usable for transfers to GPUs, because GPU memory is normally not directly accessible from host and therefore, polling on flags is expensive. Also, extra data copies between the buffer and the destination are required. These are much more expensive than host-to-host copies and significantly reduce the benefit of GPUDirect RDMA. Therefore, for a data transfer to a GPU memory buffer, the *rendezvous* protocol is used for all message sizes. This adds some overhead to the communication, but this overhead is small compared to data movement from and to the GPU. However, this shows that two-sided communication may not be the best communication method for host-controlled direct GPU to GPU data transfers.

Another framework that combines MPI with GPUs is MPI-ACC [105]. The main focus of MPI-ACC is portability, therefore it not only supports CUDA but also OpenCL. Also, it is independent of library versions and device families. Currently, it does not support GPU DirectRDMA.

4.1.2. GPUs in PGAS languages and libraries

Due to the growing interest in GPUs and the PGAS programming model, GPUs were also integrated in different PGAS languages and APIs. The idea of an *asynchronous partitioned global address space* was proposed in [106] by Sarawat et.al. They extend the PGAS model with two ideas: *places* and *asyncs*. A place is a collection of threads and data the threads operate on. These places must not be synchronous, so they do not have to have the same instruction set or number of cores. Activities are launched to places through *asyncs* and stay there for their lifetime. Asyncs can also be used for remote accesses and data transfers.

In [107], UPC is extended to support GPUs. A memory segment can either be allocated on GPU memory or host memory. UPC for GPUs can be used for data trans-

fer between the shared memory regions in UPC with `upc_memcpy(src,dest,nbytes)`, whereby both source and destination can either be located on host memory or on GPU memory. To allow this, GASNet was also modified to support GPUs.

In [108], Potluri et.al. extend openShmem to support GPUs. The openShmen standard provides no natural support for heterogeneous memory structures; therefore, some extensions were added. Potluri et.al propose three possibilities to extend openShmem for heterogeneous systems. The first one is a *static heap selection* whereby every process can create shared memory segments on either host memory or on GPU memory, but not on both. This approach requires only a few changes to the API, but does not work well with applications that require both shared buffers on host memory and on GPU memory.

Another possibility is the *dynamic heap selection* which allows a single process to create shared memory segments on both, GPU memory and host memory to overcome this limitation. However, this model has some interoperability issues with CUDA and OpenCL. The problem is caused by the openShmem interface which requires virtual pointers as input parameter for memory transfer operations.

CUDA provides several functions to detect if a virtual pointer points to GPU memory or to host memory; however, this is not supported for openCL. To support both libraries, a third method was chosen: *Symmetric mapping*. This solution allows the user to allocate GPU memory regions and then map them to the shared symmetric address space. The mapping and unmapping functions are collective blocking functions. The disadvantage of this solution is that for openShmem-related functions a different virtual pointer is required than for GPU related functions.

The current expansion of openShmem uses GPUDirect peer-to-peer for intra-node communication and buffered data transfers with GPUDirect 1.0 for inter-node communication. The support of GPUDirect RDMA is not yet implemented. However, in [109] was noted that it is planed for the future.

However, openShmem does not natively support heterogeneous memory structures. Another shortcoming is the lack of non-blocking one-sided communication functions and fault tolerance.

A further problem is that the openShmem standard 1.0 is not thread safe. This makes it harder to use with other approaches like openACC [110]. Some of these problems may be solved in future releases [111].

A one-sided communication library that provides these features already is GASPI. GAPSI seems to be the ideal candidate for a PGAS communication library to support GPUs. As GASPI is a relatively new specification and not yet wide spread, a short introduction to the standard and its first implementation, GPI-2, is given. GPI-2 currently only supports Infiniband and RDMA over Converged Ethernet (RoCe). The following description refers to the Infiniband implementation.

4.2. GASPI-Standard

The GASPI 1.0 specification [43] was released on February 14, 2013, together with the first reference implementation, GPI-2[112]. GPI-2 is the extension of the proprietary communication API GPI, which is also known as the Fraunhofer Virtual machine (FVM) [113]. The GASPI standard is based on the GPI-interface.

GASPI is a RDMA-based communication interface and all communication in GASPI is routed through the NIC. The basic idea of GASPI is to start one process per node or socket and to use thread-based parallelism on the node level. Therefore, all communication functions in GASPI are thread-save. GASPI should not be used for communication on shared memory machines, since the communication is routed through the NIC. This is one of the main differences to openSHMEM.

The processes of a GASPI application are organized in *groups*. These groups can be created dynamically during runtime. One initial group, `GASPI_GROUP_ALL` includes all participating processes. One of the main benefits of GASPI – compared to other PGAS communication libraries and languages – is *fault tolerance*. All non-local operations in GASPI come with a timeout parameter. A non-local operation is an operation that depends on a remote process. This prevents process blocking for an infinite amount of time, if a remote process does not react.

GASPI also maintains a failure vector with the current status of the nodes. If a node fails, GASPI allows continuing the work with a reduced node set. However, GASPI does not provide an automatic failure handling like check pointing or restarting.

4.2.1. Shared memory segments in GASPI

Figure 4.1 shows the GASPI memory model. Every process has its own, local memory which is not accessible for other processes. The memory in the *partitioned global address space* is also located on a specific node, but accessible for all other processes with special GASPI read and write functions. In GASPI, this global address space is made up from so-called *segments*. The segments are created dynamically during runtime. Every segment has a local, unique ID for communication functions. One of the goals of GASPI is to map the versatility of the memory hierarchies of modern high performance systems to the software layer, therefore heterogeneous memory structures are naturally supported. A segment can be created on individual nodes and can have a different size on every node, as shown in Figure 4.1.

In the implementation GPI-2, the creation of a shared memory segment means that the memory for a segment is allocated, locked, and registered for the network device. The memory is not unpinned until the segment is destroyed. To register the segment for a remote process, all required information about this segment is shared with the remote process.

For Infiniband, this information includes the virtual address, the size, and the r_key of the registered memory region. Segment creation and registration are very time consuming operations. However, once a segment is created and registered for remote pro-

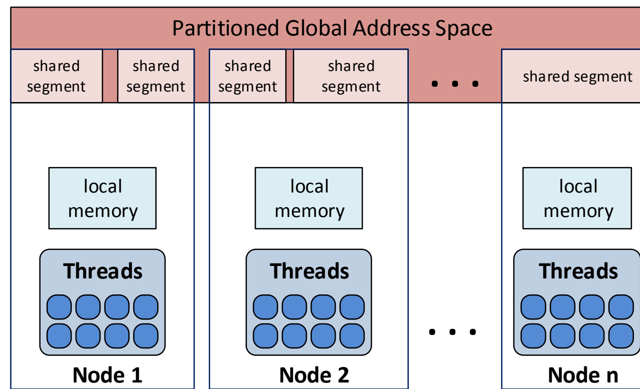


Figure 4.1.: GASPI memory model

cesses, it can be used for one-sided communication without sharing further information between the processes.

4.2.2. One-sided communication

The main communication functions in GASPI are one-sided read (`gaspi_read`) and write (`gaspi_write`) functions from and into the shared segments. All one-sided communication functions are non-blocking and asynchronous. Calling `gaspi_read` or `gaspi_write` adds the communication request to a queue and directly returns without completing the communication locally or remotely. The function still has a timeout parameter. This timeout parameter is required because the read and write functions are thread-save. If `GASPI_BLOCK` is used as timeout parameter, the function still returns without completing the communication. The function only blocks until the calling thread can submit a new communication request without interfering with another thread.

GASPI does not use the virtual addresses to identify a memory segment but the the remote segment IDs and the offsets within the segments. The GASPI runtime system determines the source and the destination address by using the segment IDs and these offsets.

Communication requests are submitted to so-called *queues*. GASPI provides an in-order execution of the requests in one queue, while requests in different queues are independent of each other and can be synchronized independently. The concept of queues is very similar to the concepts of streams for GPUs and the QPs in Infiniband.

The communication requests in one queue are synchronized with `gaspi_wait`. A queue has a maximum number of pending requests and posting a communication request to a full queue can result in undefined behavior. A `gaspi_wait` call only ensures the local completion of the communication requests. So for remote write requests, it is only guaranteed that all the data from the local buffer are read but not that the data is actually completely written to the remote memory segment.

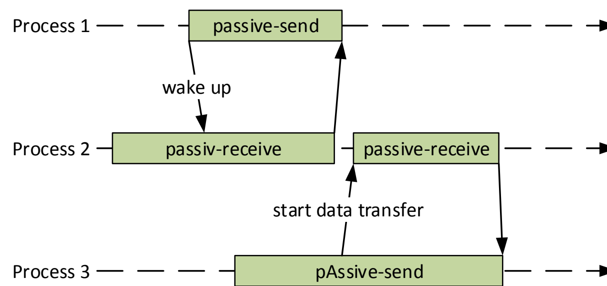


Figure 4.2.: Passive communication in GASPI

In GPI-2 for Infiniband, for every queue an RC-QP connection to every remote process is established. The QPs of one queue share a sinle completion queue.

A write or a read request adds a work request to the send queue of a QP, the wait routine polls for the completions of all requests on the respective completion queue.

4.2.3. Passive communication

Another concept in GASPI is *passive communication* which has a two-sided semantic with a *passive-send* operation, on the one side, and a matching *passive-receive* operation, on the other side. A *passive- receive* operation requires no knowledge about the sending process but waits for a *passive send* from all processes within a specified group.

A passive-receive operation blocks until the receive operation is completed, but should preferably consume no CPU-cycles. Therefore, at best, a blocked thread is woken up by the network hardware. Figure 4.2 shows how passive communication works. Process 2 posts a *passive-receive* request and then blocks. It now can accept *passive-send* requests from processes 1 and 3. Then, process 1 posts a *passive-send* request to process 2 and thereby process 2 is woken up. Now the data transfer starts and both processes block until the communication is completed.

Meanwhile, process 3 also posts a *passive-send* to process 2. The data transfer cannot be started until process 2 completes the first receive request and posts a new one. Therefore, process 3 blocks until the previous operation is completed and process 2 posts a new passive- receive request. Process 3 waits in a busy sate. Both passive- send and passive-receive operations come with a timeout parameter so the blocking is time-based.

Passive communication can be used to redistribute work, to log status messages, or to check the load status of a remote node. It should be used out of sync with the rest of the application, because the latency of passive communication is much higher than the latency of remote read and write requests.

In GPI-2, for the passive communication a further RC-QP connection to all remote processes is created. These QPs share one *shared receive queue*. A passive receive adds a receive request to this queue and waits on the completion channel for the completion of this requests. A completion channel is a file descriptor that is used to deliver completion notifications to user-space processes.

Waiting on a completion channel sets the thread to sleep and the thread blocks until a completion is created or a timeout occurs. A passive send request adds an Infiniband send request to the send queue of the QP and waits until the communication is completed or a timeout occurs. In this case, the remote send request is continued with the next call.

4.2.4. Collective Operations

A *collective operation* in GASPI is an operation that involves all processes in a specified group. Collective operations are time-based blocking. They block until the operation is completed or until they are interrupted by a timeout. In this case, the collective operation is continued with the next call.

GASPI supports only two kinds of *collective operations*: a barrier and an allreduce operation. The allreduce operation is not bound to a segment but can use any memory buffer for input or output.

Compared to other communication libraries, like MPI, the number of collective operations is limited. The reasoning for this is that a user-defined and application-specific collective operation to distribute or collect data is easy to implement and, in most cases, outperforms a generic solution. Furthermore, since the GASPI-interface should stay as small as possible, collective operations are deliberately omitted.

Collective operations should not interfere with one-sided communication requests or passive communication. Therefore, a further QP connection between all nodes is established for collective communication requests. Therefore, every GPI-2 process has to manage at least

$$N * (2 + Qn)$$

QPs, whereby N is the number of processes and Qn is the number of queues.

The collective operations are performed on a pre-allocated memory area, which is is also locked and registered for the Infiniband device. Internally, the collective operations use RDMA write requests and a binomial tree.

4.2.5. Atomic operations

GASPI also provides support for global atomic operations on shared memory segments. The two basic atomic operations, *compare-and-swap* and *fetch-and-add*, are supported. Global atomic operations can be used for global counter variables or global synchronization primitives.

GPI-2 uses the atomic operation support of Infiniband. Atomic work requests are submitted to the same RC-QP as the collective operations, therefore no additional QPs are required.

4.2.6. Weak synchronization

As a point-to-point synchronization mechanism between source and target of a one-sided communication process, GASPI provides a so-called *weak synchronization*. Weak

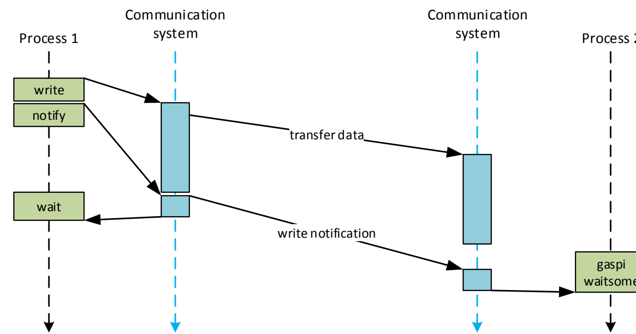


Figure 4.3.: Weak synchronization in GASPI

synchronization allows the origin process to update a notification flag for a shared memory segment on a remote process. On the target side, the flag can be used, for example, to check if a remote write or read operation is completed. To every shared memory segment, several flags are assigned for weak synchronization. Figure 4.3 shows the functionality of weak synchronization. Process 1 uses a write operation to transfer data to a remote memory segment. Next, a notification is sent to the remote side. Since both requests are posted to the same queue, they are guaranteed to be in order and process 1 synchronizes them locally with the same `gaspi_wait` call.

On the target side, the function `gaspi_notify_waitsome`is used to wait for notifications. Multiple threads can call this function in parallel and poll on one or more flags. If the notification flag is updated, it is guaranteed that the previous write operation is completed.

To reset a notification, a further local function call, `gaspi_notify_reset`is required. This function resets the flag in an atomic manner.

GASPI provides two kinds of operations with a notification: `gaspi_notify` updates a flag for a remote segment while `gaspi_write_notify` first performs a write operation on a remote memory segment and then directly updates a notification flag for this segment. Both operations are non-blocking and locally synchronized with `gaspi_wait`.

Writing a notification is still a one-sided operation, since the target side is not required to complete the communication function. The remote side can ignore a notification without further consequences for the communication process.

GPI-2 for Infiniband uses an RDMA write operation to update a notification flag on a remote segment. The request is submitted to the RC-QP that belongs the queue for remote write and read requests.

4.3. Integration of GPUs to the GASPI-specification

GPUDirect RDMA allows RDMA-capable devices direct access to GPU device memory, so this technique is best suitable for RDMA-based programming models like GASPI. As mentioned before, currently, GASPI supports only Infiniband network devices and

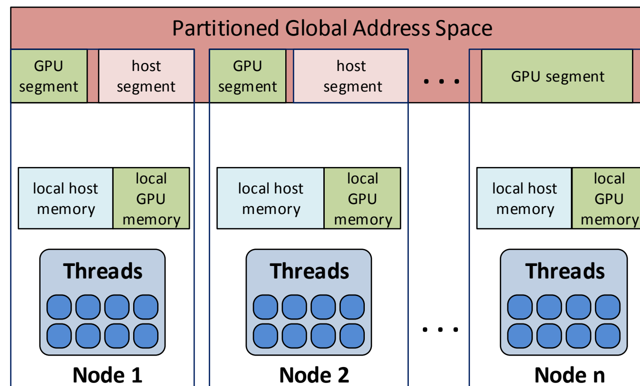


Figure 4.4.: GASPI support for GPUs

RDMA over Converged Ethernet (RoCe) but not Extoll. Therefore, the expansions to GASPI for GPUs only refers to the Infiniband version. However, the basic ideas can also be applied to Extoll.

4.3.1. GPU memory segments in GASPI

The GASPI specification was from the very beginning designed to support heterogeneous memory structures. This idea allows an easy integration of GPUs and GPU memory segments to that specification, as shown in Figure 4.4. The GPU-expansion allows not only the creation of shared memory segments on host memory but also on GPU memory. Thereby, a part of the memory of each GPU is made accessible for other processes with GASPI communication functions.

In GASPI two functions exist to create a shared memory segment: The function `gaspi_segment_create(ID,size,group,timeout,gaspi_alloc_policy)` creates a segment and registers this segment for all other processes in the specified group. The function `gaspi_segment_alloc(id,size,gaspi_alloc_policy)` only allocates a segment but does not register it for remote processes. It can be used on the local node as source or target for a write or read request, but a remote process cannot read or write these segments with RDMA requests. To allow this, the segment must be subsequently registered with `gaspi_segment_register`. This functionality can be used to create heterogeneous memory structures.

Both functions require the parameter `gaspi_alloc_policy` which defines the *allocation policy* for the memory segment. In the current specification, two values for this flag are defined: `GASPI_MEM_DEFAULT` and `GASPI_MEM_INITIALIZED`. The latter one initializes the segment memory with zero. The GPU extension uses this parameter to specify if a segment should be allocated on GPU device memory. If the flag `GASPI_MEM_GPU` is set, the segment will allocated on GPU memory, otherwise it will be allocated on host memory. The `GASPI_MEM_INITIALIZED` and `GASPI_MEM_GPU` flags can be connected with an OR-conjunction.

Listing 4.3: GASPI code example

```
gaspi_segment_create(segID,size,GASPI_GROUP_ALL,
                     GASPI_BLOCK,GASPI_MEM_INITIALIZED);        2
gaspi_segment_ptr(segID,
                 &(gaspi_pointet_t)HostMem)                     4
/*pointer to host memory*/
/*do something*/                                                6
gaspi_write(segID,locOff,remrRank,remSegID,remOff,
            size, queue, GASPI_BLOCK);                          8
/*do something*/
gaspi_wait(queue, GASPI_BLOCK);                                 10
```

Listing 4.4: GASPI for GPUs code example

```
gaspi_segment_create(segID,size,GASPI_GROUP_ALL,
                     GASPI_BLOCK,                               2
                     GASPI_MEM_INITIALIZED|GASPI_MEM_GPU);
gaspi_segment_ptr(segID,                                        4
                 &(gaspi_pointet_t)DeviceMem)
/*pointer to GPU memory*/                                       6
cudakernel<<< ··· >>>(DeviceMem,···);
gaspi_write(segID,locOff,remRank,remSegID,remOff,               8
            size,queue,GASPI_BLOCK);
cudakernel<<< ··· >>>(DeviceMem,···);                           10
gaspi_wait(queue, GASPI_BLOCK);
```

Once a segment is created, it makes no difference for GASPI-related operations where the segment is allocated. All GASPI functions operating on shared host memory segments are also supported on GPU memory segments.

Listings 4.3 4.4 show short GASPI code examples. In Listing 4.3, a host memory segment is created and `gaspi_write` initiates a data transfer between two host memory segments. In Listing 4.4, a GPU memory segment is created, therefore `gaspi_write` initiates a data transfer between two GPU memory segments. The only difference between the two code examples is the use of the `GASPI_MEM_GPU` flag. Of course, GASPI can also be used to transfer data between a host and a GPU segment. It is not required that both segments are of the same type. The following sections describe how this integration of GPUs to GPI-2 is realized.

4.3.2. Initialization

Before GASPI can be used with GPU memory segments, the function `gaspi_init_GPUs` must be called. This function detects all GPUs on the same root complex as the Infiniband device that is used by the GASPI process.

To detect these GPUs, the Linux *sysfs* filesystem is used. The *sysfs* provides information about the location of every PCIe device. The initialization function reads this information for the GPUs and compares it with the information of the Infiniband device that is associated with the GASPI process.

Only GPUs on the same root complex can be used for direct, one-sided communication, as shown in section 3.4.4, since, otherwise, GPUDirect RDMA is not supported. This case requires a work-around with helper threads on both sides. However, this is not the topic of this work and will therefore be addressed in future work. Also, before GPUDirect was introduced, staged protocols and helper threads were necessary to transfer data between distributed GPUs and, therefore, this has been discussed in great detail before, for example, in[101] or [108]. For the further description and discussion in this work, we assume that the GPU and the network device are located on the same root complex and GPUDirect RDMA is supported.

4.3.3. GPU memory segment creation

If a shared GPU memory segment is created, the GASPI library allocates the device memory, locks it, and registers it for the Infiniband device using GPUDirect RDMA. Locking and unlocking of GPU memory are expensive operations. Therefore, the most straightforward way, pin memory before each transfer and unpin it right after the transfer is completed, would perform poorly in general which means lazy pinning and unpinning strategies are required. The memory management in GASPI is simple: A memory area is pinned when it is created and will not be unpinned until the segment is destroyed.

By this, unnecessary locking and unlocking operations can be avoided and communication can be performed without entering the operation system. However, this means that in GASPI for GPUs, the size of GPU memory segments is limited to the BAR size, which is 256 MB on most systems today. This problem is addressed later in more detail.

On a system with more than one GPU, the memory is allocated on the currently active GPU device, which can be defined with *cudaSetDevice*. If the active device does not support GPUDirect RDMA or is located on a different root complex, an error is returned.

Host memory segment allocation

If GPUs are initialized for the current GASPI process, new host memory segments are allocated with `cudaMallocHost`. Thereby, the memory is directly locked and registered for the GPU. Host memory regions are registered for the network device anyway, which also includes locking, so by this, no additional host memory is locked. The advantage of this is that for internal copies between a host and a GPU memory segment, the memory engine of the GPU can be used. This can help to relieve the network device from internal copies.

4.3.4. Remote write and read operations

Once a GPU memory segment is allocated and registered, it makes no difference for the Infiniband network device if the memory segment is allocated on host memory or on device memory. Therefore, remote read and write operations are natively supported on GPU device memory. However, for small messages, GASPI uses the Infiniband inline-

flag which is not supported for GPU memory segments and, therefore, this must be avoided for GPU memory segments.

4.3.5. Passive communication and atomic operations

Passive communication uses the two-sided *send/receive* communication supported by Infiniband. Since this works on GPU memory segments as well as on host memory segments, no further changes in the code are required to support this.

The same holds true for atomic operations, which also use the native support of Infiniband. It should be noted that the atomic operations are only atomic regarding GASPI accesses. If a CUDA thread accesses a value simultaneously, the atomic access is not guaranteed.

4.3.6. Weak synchronization for GPU segments

In GASPI, for every shared memory segment, an additional area of flags, which are used for weak synchronization, is allocated, locked, and registered. Normally, this segment is located at the beginning of the shared memory area. However, for GPU memory segments, this is not recommended, as, in this case, the flags would be allocated on GPU device memory.

Since the host controls the communication and synchronization, the flags are only accessed from host. Waiting for a notification means polling on a flag. To allow the CPU polling on flags in GPU memory, the GPU memory must be mapped to the user space with MMIO, as described in section 3.3.3 or CUDA memory copy operations must be used.

This would result in many additional read accesses through the PCIe-bus, which would result in performance losses. Also, as mentioned before, lockable GPU memory is a very scarce resource and should be used sparsely.

Therefore, for every GPU memory segment, a smaller host memory segment for the flags is allocated and registered for the network device. This host memory area is also registered for all other nodes in a specified group but invisible to the user.

A `gaspi_write_notify` operation, which targets a remote GPU memory segment, first initiates a data transfer between the local segment and the remote GPU segment. Then, the flag is transferred to the remote flag area in host memory. This allows an easy synchronization between the source and the target side. By this, some of the issues which we have seen for two-sided communication in section 4.1.1 can be avoided.

4.3.7. Allreduce

The allreduce operation is not bound to a shared memory segment, but it can use any buffer as input and output. GPI-2 for GPUs also supports allreduce operations with any GPU buffers for input or output.

For host memory input or output buffers, GASPI copies the input data to a shared memory segment. This shared memory segment is invisible to the user and only used

for collective operations: the allreduce operation and the barrier. We call this segment the *shared group segment*.

This shared group segment has a limited size and, therefore, allreduce operations are only supported up to a limited number of input elements, which in the current version, amount to 255 input elements.

A memory copy between two host memory buffers is much faster than a copy operation between GPU and host memory and therefore slows down the performance of allreduce operations with GPU memory buffers.

However, if the data are not copied to host memory but remain on the GPU, the GPU has to perform the reduction operation since the CPU is not able to access data on the GPU directly. In principle, GPUs are highly suitable for this kind of workload. Still, the time-consuming part of the allreduce operation is not the reduction but the data transfer. First, the input data are collected from all processes. After the reduction, the result is broadcasted back to all processes.

To optimize this, often tree-based approaches are used. The same is true for GPI-2. If the GPU performs the reduction, a reduction kernel has to be started for every branch of this tree. Due to the nature of GASPI, the GPU then can perform at least 255 reduction operations in parallel. The overhead of starting and stopping the kernel outperforms the benefit of the GPU in this case. Another problem is that a reduction kernel may interfere with a computation kernel on the GPU. Although concurrent kernel execution is possible, a large compute kernel may block the GPU so that the reduction kernel is not scheduled. This may result in a large delay for the allreduce operation.

Besides this overhead, the input or output buffers do not have to be located in shared memory segments. Therefore, either the buffers have to be registered dynamically with GPUDirect RDMA or the data have to be copied to a registered shared group segment in GPU memory. This overhead may be smaller than copying the data to host memory, but it still cannot be neglected.

Because of this, we decide to copy the data to the shared group segment on host memory. To do this as fast as possible and to avoid additional copies, the shared group segment is registered for the GPU during GPU initialization.

The allreduce function first checks if the input or output pointer belongs to GPU memory or to host memory. For this, the function `cudaGetDeviceProberties` is used. If the pointer belongs to GPU memory, `cudaMemcpy` is used to copy the data. Except for this modification, no further changes are required to support the allreduce function with GPU memory buffers as input or output.

4.3.8. Additional functions for GPUs

To allow a more flexible use of GPUs and to facilitate the use of GPUs in GASPI, we defined some additional, GPU-related, functions. These functions will now be explained in more detail.

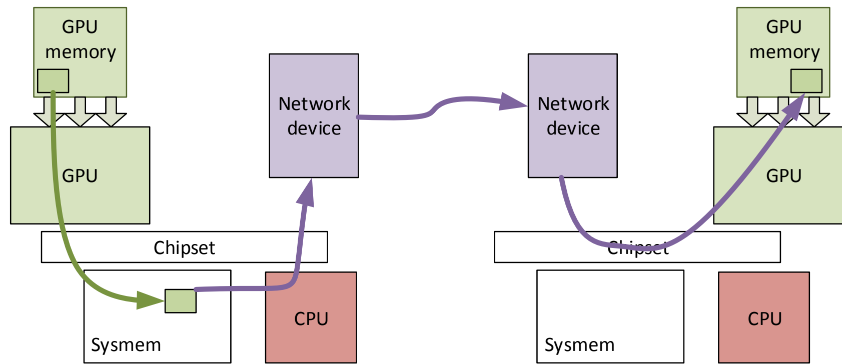


Figure 4.5.: Data flow for the hybrid data transfer protocol, CPU on the left system is required to control the data flow between host memory and network device

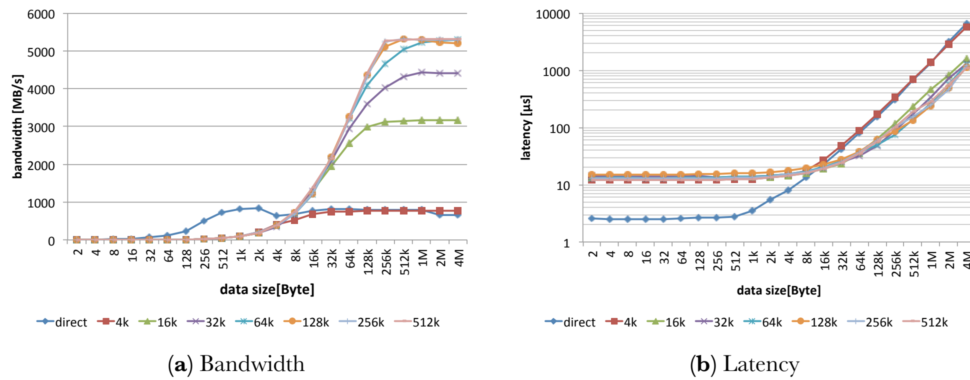
Hybrid transfer protocol

As shown in the pervious chapter, the maximal bandwidth for a GPU to GPU data transfer is limited due to the insufficient support for non-posted device-to device operations through the PCIe-bus. We also showed in section 3.4 that better latency and bandwidth can be reached by using staged copies in host memory.

Therefore, we developed a hybrid protocol for remote write operations which uses GPUDirect 1.0 for large data transfer sizes and GPUDirect RDMA for small data transfer sizes.

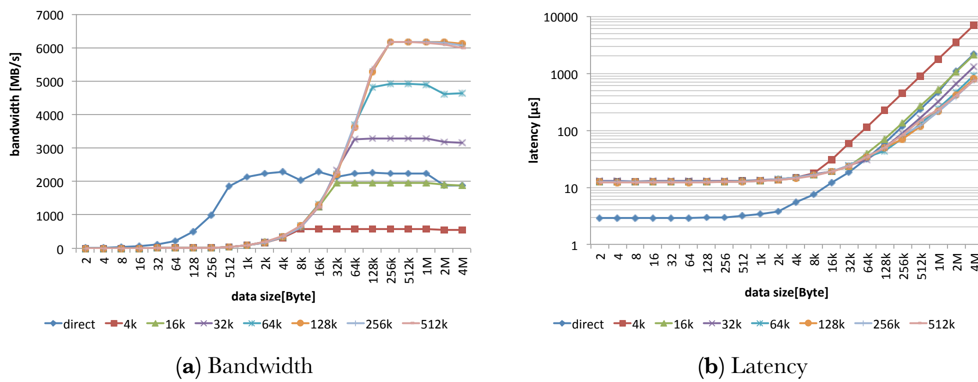
Small messages are still directly transferred using GPUDirect RDMA. Larger messages are first copied to a buffer in the host memory on the sourcing side, as shown in Figure 4.5. Therefrom, the data is directly transferred to the GPU memory on the target side. This operation is still one-sided, as the target side is not actively involved in the communication process. However, on the sourcing side, a CPU thread is required to control the data flow between host and device. Thus this protocol is not fully asynchronous, which dissents the GASPI standard. Therefore, we define a new command: `gaspi_gpu_write`, so `gaspi_write` still can be used for fully asynchronous but slower remote write operations.

Since the concept of queues in GASPI is very similar to the concept of streams in CUDA, we create one CUDA stream for every GASPI-queue which is used to transfer the data from GPU to host. To achieve the maximal performance for larger messages, we use a *pipelined* protocol to transfer the data, so data is transferred in chunks. The chunks are copied between the GPU and the host buffer with an asynchronous CUDA copy operation and every copy operation is followed by a CUDA event. If all GPU to host copy operations are added to the stream, the process starts to poll for the completion of the *events* and, thereby, for the completion of the asynchronous copy operations. If such an asynchronous CUDA copy operation is completed, the network device is initialized to transfer the data between the host memory buffer and the remote memory segment by adding a new work request to the Infiniband queue.



(**a**) Bandwidth (**b**) Latency

Figure 4.6.: One-sided staged data transfers with different block sizes on a Sandy Bridge machine with K20 GPUs and QDR Infiniband



(**a**) Bandwidth (**b**) Latency

Figure 4.7.: One-sided staged data transfers with different block sizes on a Ivy Bridge machine with K40 GPUs and FDR Infiniband

To allow efficient copy operations between GPU and host, we use pre-allocated host buffers. These buffers are registered for the network device and for the GPU, thus asynchronous copy-operations are supported. These segments do not have to be registered for remote processes since they are only required for local transfers and not remotely accessed.

Optimization

The data transfer using `gaspi_gpu_write`can be optimized by two factors: the maximal data size that is transferred directly and the chunk size for the pipelined transfer.

Figures 4.6 and 4.7 show the latency and the bandwidth for a direct data transfer and a one-sided staged transfer with different block sizes.

Figure 4.6 shows the results for two Kepler K20 GPUs, each on a machine with two Intel Xeon E5-2609 four core CPUs (Sandy Bridge), connected with Mellanox QDR Infiniband. Figure 4.7 shows the results for two Nvidia K40 GPUs on a dual socket

machine with two 10 core Intel Xeon CPU E5-2690 v2 CPUs (Ivy Bridge), connected with Infiniband FDR.

On the Sandy Bridge machine with K20 GPUs, a direct data transfer of up to $4-8$ kB shows the best performance, on the Ivy Bridge machine, this is the case for a transfer size of up to 32 kB.

On both machines, the best bandwidth is reached for a block size between 256 kB and 512 kB. The latency is slightly better for a block size of 256 kB. However, since the perfect choice of these parameters strongly depends on the used hardware, they are defined as runtime parameters.

Dynamic segment mapping

On most devices, the GPU BAR size is limited to 256 MB. Therefore, the maximal size of all GPU segments on one GPU also corresponds to 256 MB. For some Tesla Cards, a larger BAR size is supported,or example, 6 GB. This allows the mapping of the complete GPU memory to the BAR. However, a larger BAR size can cause problems for older BIOS due to compatibility support for 32 bit operating systems. On these systems, the bootstrap can fail, or the GPU may be unusable due to misconfiguration. However, the limit of 256 MB shared memory can limit the problem size or induce additional memory copies.

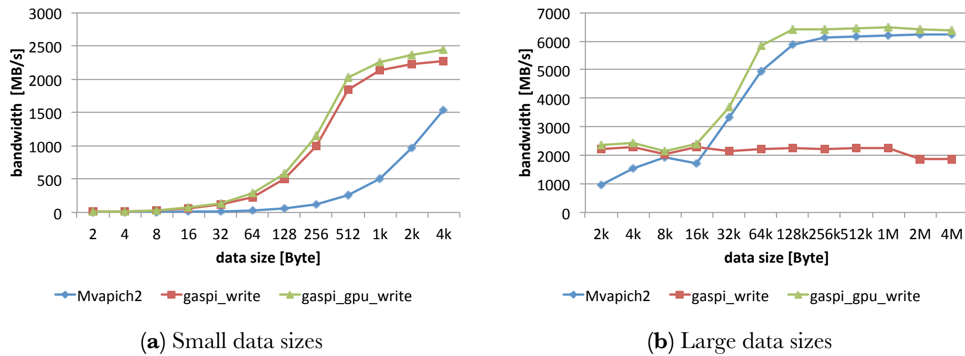
Therefore, for GPU segments, a more flexible usage is allowed. Similar to open-SHMEM, GASPI allows the dynamic mapping of GPU memory to shared memory segments.

This means that an already allocated GPU memory region can subsequently be mapped to the shared address space. Since GASPI does not use virtual addresses but segment IDs for identification, a subsequently registered segment can be used like a normally allocated segment, without further address translation. Similar to the segment allocation functions, GPI-2 for GPUs provides two different functions for mapping registered memory to the global address space. The function `gaspi_map_create` is a collective operation that maps the memory to the global address space and registers it for all processes in the specified group, whereas `gaspi_map_seg` only creates the segment locally but does not register it for other processes.

However, pinning of these memory regions is a very time- consuming function and therefore this feature should not be used to frequently map and unmap shared memory regions.

4.4. Performance results

In this section, the performance results for GPI-2 GPU support are presented and discussed. The results are compared with the CUDA-aware MPI version Mvapich2, with GPUDirect RDMA support enabled. Unless otherwise noted, the following machines are used for testing: one node of these systems is a dual socket machine with two 10-core Intel Xeon CPU E5-2690 v2 CPUs (Ivy Bridge), connected with Infiniband FDR. Every



(**a**) Small data sizes (**b**) Large data sizes

Figure 4.8.: Bandwidth in GPI-2 for a GPU to GPU inter node data transfer, compared with Mvapich2

machine has six Nvidia K40c GPUs, of which four are connected to the same socket as the Infiniband device and therefore can be used with GPUDirect RDMA.

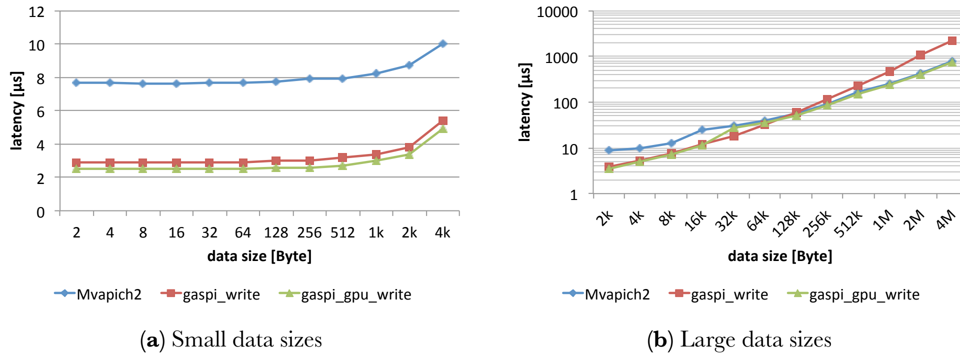
4.4.1. Bandwidth

Figure 4.8 shows the results for bandwidth for small and large data transfer sizes. The results show the bandwidth of a remote write data transfer between two GPUs,located on different nodes. The GASPI results are compared with the results of the bandwidth benchmark of the osu-micro-benchmark suite for MPI [114]. This benchmark uses asynchronous `MPI_Isend` and `MPI_Irecv` operations to transfer the data.

For small transfer sizes, `gaspi_write` and `gaspi_gpu_write` clearly outperform MPI. For larger messages, `gaspi_write` is limited due to the PCIe issues described in the previous chapter, while `gaspi_gpu_write` still outperforms the MPI version, if only marginally. For lager messages, the overhead that is caused by the two-sided communication scheme becomes smaller compared to the data transfer latency. Both the MPI version and `gaspi_gpu_write` show a clear bend between 16 kB and 32 kB. Here, the data transfer switches from direct to staged for both communication libraries.

4.4.2. Latency

Figure 4.9 on the next page shows the results for the latency benchmarks. For GASPI a pingpong benchmark with `gaspi_write_notify` and `gaspi_gpu_write_notify` and `gaspi_waitsome` on the remote site is used. The use of weak synchronization avoids polling on GPU memory from host. For MPI, the latency benchmark of the osu-micro-benchmark suite is used. This benchmark uses `MPI_Send` and `MPI_Recv` for the pingpong benchmark. For small messages, `gaspi_write_notify` and `gaspi_gpu_write_notify` show the same low latency, since both use direct copies. Here again GASPI clearly outperforms MPI. For large messages, the results for MPI and GASPI using staged copies do not differ by much. Here, both communication libraries use staged copies and the



(**a**) Small data sizes (**b**) Large data sizes

Figure 4.9.: Latency GPI-2 for a GPU to GPU inter node data transfer, compared with MPI, Mvapich2

overhead of the communication becomes smaller compared to the data transfer time. The latency of the direct data transfer, however, is much larger.

4.4.3. CPU-communication overhead

If the communication is controlled by the CPU, the communication overhead on the GPU is equal to zero. Still, a GPU-to-GPU data transfer causes some overhead on the CPU, which is determined with the following benchmark.

The smaller the communication overhead, the better communication and computation can be overlapped on the CPU. The most common way to overlap communication and computation is the *post-work-wait* method [115]. This means, an asynchronous communication is started, in GASPI, for example, with `gaspi_write`, in MPI with `MPI_Isend`,then some work is performed before communication is synchronized, in GASPI with `gaspi_wait`, and in MPI with `MPI_Wait`. Listings 4.5 and 4.6 show short code examples for this.

Listing 4.5: Post-work-wait loop for GASPI

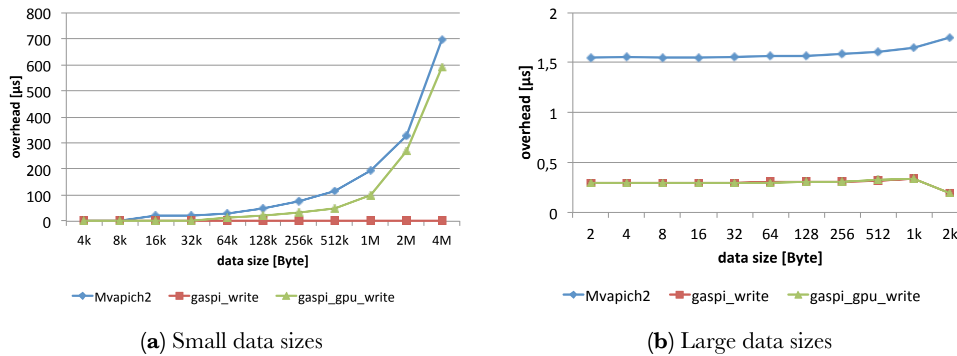
```
gaspi_write(···);
do_work(ms);                    2
gaspi_wait(···);
```

Listing 4.6: Post-work-wait loop for MPI

```
MPI_Isend(···);                 1
do_work(ms);
MPI_Wait(···);                  3
```

To measure the overhead, a method similar to the method described in [115] is used. The codes shown in Listings 4.5 and 4.6 are executed for several iterations. With every iteration, the runtime of the workload is increased and the runtime of the complete loop is measured. At one point, the runtime becomes larger than the communication time. At this point, the work time is equal to the data transfer time and the communication overhead can be determined with:

$$t_{overhead} = t_{runtime} - t_{workload=0}$$



(**a**) Small data sizes (**b**) Large data sizes

Figure 4.10.: Communication overhead in GASPI and MPI for am inter node GPU-to-GPU data transfer

We run this benchmark for `gaspi_write`, `gaspi_gpu_write` and asynchronous MPI-communication, using `MPI_Isend` and `MPI_Irecv`, to transfer the data between two GPUs on different nodes. Figure 4.10 shows the results for the overhead. The overhead of `gaspi_write`is hardly measurable and is in the area of measurement uncertainty. The same is true for small messages with `gaspi_gpu_write` since here also a direct data transfer is used.

This is different for larger messages using `gaspi_gpu_write` and MPI. The staged protocol adds a lot of overhead to the CPU due to synchronizing of the data streams between network device, GPU, and host memory. For MPI, the message passing overhead is added on top, which includes the synchronization between the sending and the receiving side as well tag matching. Note that for MPI only the overhead on the sending side is considered, on the receiving side additional overhead may be added. This is not the case for one-sided communication in GASPI.

Collective allreduce

Figure 4.11 shows the results for the total runtime of the allreduce operation in GASPI, using buffers in GPU memory as input and output. Additionally, Figure 4.12 shows the performance of the GPU allreduce operation, relative to the performance of the allreduce operation with input and output buffers in GPU memory. This benchmark was started on four nodes, each using the four GPUs connected to the same root complex as the Infiniband network device. For every GPU, a single GASPI process is started. However, the data transfer is routed through the network hardware, even if the GPUs are located on the same node. The results are not compared to MPI since MPI currently does not support allreduce operations on GPU memory.

The results show that an allreduce operation with GPU memory as input and output has a much lower performance then the operation with host buffers as input and output. For two GPUs, the latency of an allreduce operation is over 20 μs, for all numbers of input elements; for sixteen GPUs is it even more then 30 μs. This is due to the double copies between host memory and GPU memory from and to the shared buffer. For this

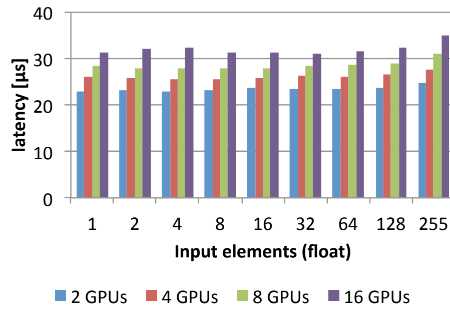


Figure 4.11.: GASPI total runtime of the allreduce operation

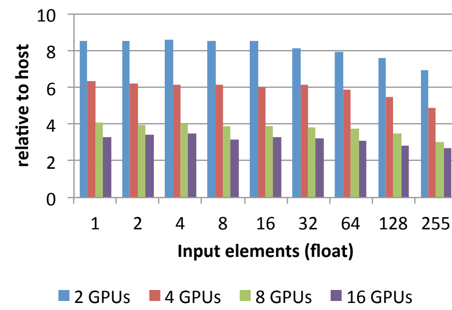


Figure 4.12.: GPU-allreduce, relative to host-allreduce operation

relatively small number of GPUs, this is the dominating factor for the allreduce operation. However, this overhead stays constant for larger numbers of GPUs. Therefore, for more GPUs the difference between host and GPU memory buffers becomes smaller and the performance relative to the host-allreduce version, becomes better. While for two GPUs the allreduce operation with GPU memory buffers takes about eight times longer than the host allreduce operation, for sixteen GPUs, it only takes twice as long.

4.5. Application level performance

The previous sections presented the results for several micro-benchmarks for GPU-to-GPU data transfers. All these benchmarks use GPU memory as source and target of a data transfer, while the GPU itself is not used for computation. However, on an application level, the GPU is used for computation and, therefore, synchronization between GPU and CPU is required. The GPU computes the data and if a new message has to be sent or if remote data are required to resume the computation, a context switch between GPU and CPU is required. Therefore, synchronization methods for GPU and CPU have to be considered first.

4.5.1. Synchronization between GPU and host

The GPU is controlled by the CPU and requires a host thread for this. This host thread starts computation kernels and waits for the completion of these kernels. Therefore, synchronization between host and GPU is synonymous with starting and synchronizing a computation kernel. However, there are several ways to handle this, which add different overhead to the application. Figure 4.13 on the following page shows the design space for Cuda GPU-CPU synchronization.

The simplest way to synchronize GPU and CPU is the device synchronization, using `cudaDeviceSynchronize`. This host function waits until all computation kernels and all data transfers between GPU and host are completed.

The function `cudaStreamSynchronize` does not return until all kernels and data GPU-host transfer operations on a specified stream are completed. The function

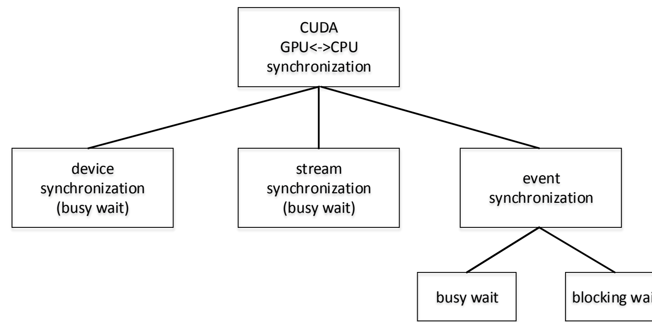


Figure 4.13.: Synchronization methods for GPU/host synchronization

`cudaStreamQuery` only tests,if all operations on a stream are completed and then directly returns. If all operations are completed, a success is returned, otherwise it will be a *not-ready* notification.

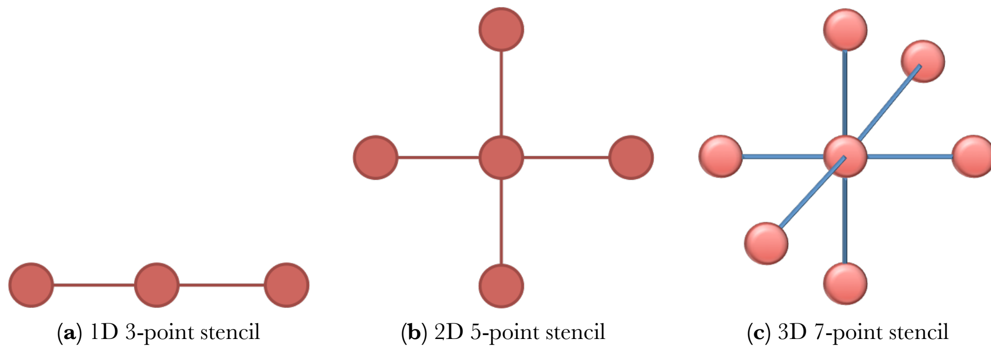
A more flexible way to synchronize CPU and GPU are *events*. Events can be submitted to a specified stream or the null-stream. The null-stream is the stream that is used if no specific stream is indicated. The function `cudaEventSynchronize` waits until all operations that a submitted before the event are completed. The function `cudaEventQuery` queries for the completion of an event, analogous to `cudaStreamQuery` for streams. These synchronization functions normally wait in a *busy state*, polling for the completion of the submitted operations. Events are an exception if they are created with the flag `cudaEventBlockingSync`. A host thread calling `cudaEventSynchronize` on an event that is created with this flag stays in a blocked state until the event is completed. The GPU wakes the thread with an interrupt.

To quantify the overhead of these different synchronization methods, a simple kernel is started on the GPU and then synchronized with one of these methods. The kernel is started with 64 blocks, each with 128 threads. The kernel only calls the function `clock` which returns the value of a per-multiprocessor counter and then directly returns. We run the test on a K20c GPU, which was linked to a workstation with two Intel Xeon E5-2630 six core CPUs. Table 4.1 shows the results.

These results show that the synchronization and context-switching overhead cannot be neglected. The fastest synchronization is the polling event synchronization . The only method in which the CPU does not poll is the blocking event synchronization. However, since an interrupt is required to wake the host thread, the latency of this synchronization

Table 4.1.: Synchronization time for different host-GPU synchronization methods

device synchronization	stream synchronization	event synchronization	blocking event synchronization
15.24μs	12.25μs	11.98μs	65.71μs



(**a**) 1D 3-point stencil (**b**) 2D 5-point stencil (**c**) 3D 7-point stencil

Figure 4.14.: Stencil environments

method is much higher. For an efficient synchronization between host and GPU, the event or stream synchronization should be used.

4.5.2. Stencil codes

For a benchmark that is targeted more at the application level, we use a stencil code. Stencil codes are iterative kernels which update an element according to some fixed pattern, called stencil.

A stencil is usually performed on a two- or three-dimensional grid. In each iteration, the stencil code updates all elements within this grid using neighboring elements, as shown in Figure 4.14, for different stencil configurations. Stencil codes are very common in computer simulations like computational fluid dynamics or for solving partial differential equations.

On distributed memory systems (for example, distributed GPUs), the grid is decomposed and distributed between all participating processes. Before a new value is calculated, it must be ensured that all neighboring elements are up to date.

On a single GPU, this means that after an iteration all threads have to be synchronized before a new iteration is started. This is normally done by starting a new computation kernel for every iteration and putting all these kernels to the same stream.

If the grid is distributed over multiple GPUs, the boundary points of each iteration have to be exchanged with the remote GPUs, as shown in Figure 4.15 for a two dimensional grid. In this simple case, the domain is only decomposed along the slowest direction. In every iteration, the left and the right borders have to be exchanged with the neighbors. The advantage of this domain decomposition is that only continous memory areas have to be transferred.

Since the left part of a grid can be updated independently from the right part of the grid, the computation of the right part can be overlapped with the boundary exchange of the left part and the other way around. There are a lot of examples, e.g. [116, 117], of optimizing this for GPUs with CUDA and MPI.

In this work, we use the himeno code [118]. It was developed in 1996 at the RIKEN Institute in Japan and it focuses on the solution of 3D Poisson equations in generalized

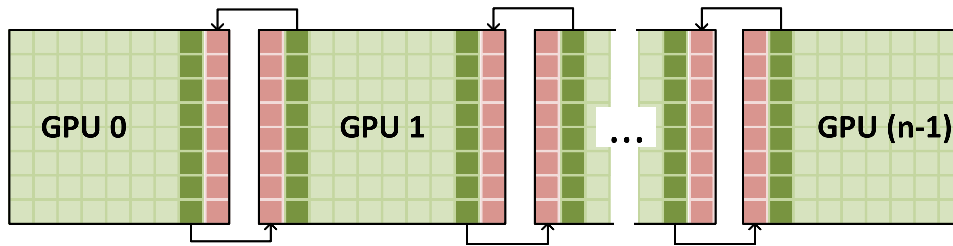


Figure 4.15.: Boarder exchange for a 2-D stencil code

coordinates on a structured curvilinear mesh. Using finite differences, the Poisson equation is discretized in space. The discretized Poisson equation is solved iteratively, using Jacobi relaxation, which yields a 19-point stencil.

A recent and well performing multi-GPU version with MPI for inter-GPU communication is described in [8]. The domain is sliced along the z-direction and distributed over the GPUs.

The benchmark uses two kernels, one for the upper part of the grid, one for the lower part. In our version, a single stream is used for both compute kernels to guarantee the local data consistency. The version in [8] uses two streams, one for the upper part and one for the lower part of the grid. Since this version does not use a CUDA-aware MPI version, two streams are required to overlap the computation and the data transfer between GPU and host. Using CUDA-aware MPI or GASPI for communication, this overlapping is handled by the communication library. Therefore, a single stream is sufficient.

To synchronize the kernels and the CPU for data transfer, we use *events*, because this is the most efficient synchronization method, as shown in the table 4.1. In Figure 4.16, the control flow of this hybrid application is shown.

In every iteration, a host thread first starts the data transfer of the bottom boundary (*send_top*) and then the compute kernel for the top part of the grid (*top_kernel_start*). The data transfer itself is handled by the network device or the underlying communication library and overlapped with the computation on the GPU. Theoretically, the CPU can now be used for other work or set to sleep while the compute kernel is running on the GPU and the network device handles the data transfer. Before the next compute kernel can be started, the CPU has to ensure that the remote boundaries are updated (*wait_btm*). Similarly, before the CPU can start the data transfer of the top boundaries (*send_top*), it has to ensure that the top kernel is completed. Therefore, usually a host thread is delegated to control the flow between network device and GPU for such multi-GPU applications.

The MPI-version uses asynchronous `MPI_Isend`to send the data and `MPI_Ireceive` to wait for the remote data. The GASPI-version uses `gaspi_gpu_write_notify`to send the data and `gaspi_notify_waitsome`to wait for the remote data. The MPI- and the GASPI-version only differ in the communication routines.

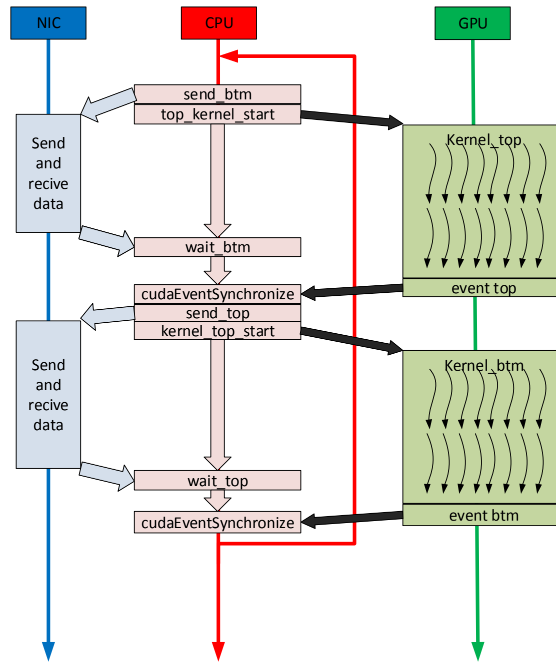


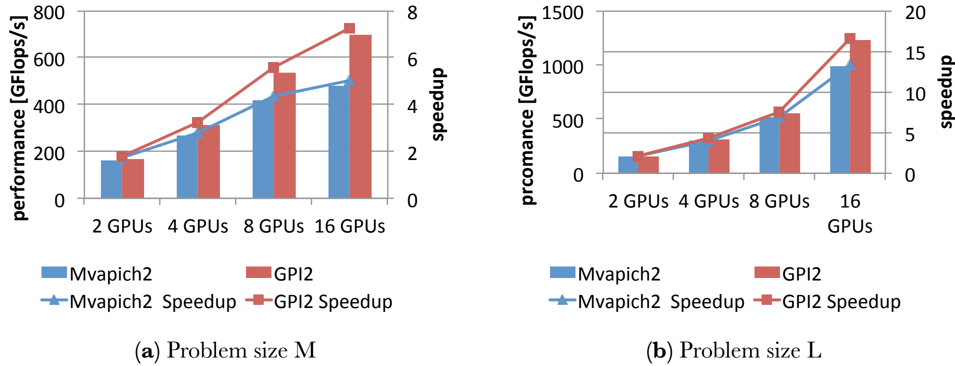
Figure 4.16.: Control flow of a multi-GPU application, using events for synchronization; communication is controlled by the CPU

This version of tje benchmark only uses the GPU for computation while the CPU is only used for the communication. Therefore, this benchmark does not count in the CPU overhead. We run the benchmark with two fixed problem sizes, as shown in table 4.2. The performance results for strong scaling are shown in Figure 4.17a for the M-problem size and in 4.17b for the L-problem size. The graphs also show the speedup relating the single GPU- version.

For two GPUs, GASPI and MPI show the same results. For more GPUs, the GASPI-version shows progressively a better performance and scaling than the MPI-version. For the larger problem size, the scaling for GASPI is almost linear and using sixteen GPUs brings a speedup of 16, whereas for MPI, only a speedup of 13 can be reached. For the smaller problem size, the scaling is not as good. The GASPI version provides a speedup of 7.3, while the MPI-version only shows a speedup of 5 for sixteen GPUs.

Table 4.2.: Himeno problem size

	X	Y	Z	MFlops per iteration	data transfer size (kByte)
M	128	128	256	137.1	64
L	256	256	512	1118.7	256



(**a**) Problem size M (**b**) Problem size L

Figure 4.17.: Strong scaling of the Himeno benchmark using GPI-2 and MPI

The medium problem size provides 256 points along the z-direction. This means that for 16 GPUs, only a grid size of $128 \times 128 \times 16$ is processed on a single GPU. Here, the computation cannot be completely overlapped with the computation and the communication time becomes the dominating factor. Since the communication latency in GASPI is lower, the GASPI-version provides the better scalability.

4.6. Summary

In this chapter, we have shown how GPUDirect RDMA can be used to integrate GPUs into the one-sided communication library GASPI. A one-sided communication pattern can benefit from these new techniques, in particular for small message sizes. For larger messages, staged protocols, in which the data are buffered on one side of the communication, show a better performance but add a lot of overhead to the CPU.

However, the performance results show that GASPI outperforms MPI as state-of-the-art for small and medium- sized messages and on the application-level benchmark. The lower latency – caused by a lower communication overhead– results in a better scalability.

Nevertheless, a hybrid model requires a lot of context switches between host and GPU which adds a lot of overhead to the CPU. Event synchronization is the most efficient way to synchronize host and GPU, but still requires approximately $12 \mu s$. Still, on an-application level benchmark, this overhead can completely be overlapped with computations on the GPU. Therefore, almost a linear speedup can be reached if the problem size is large enough. However, using this hybrid communication model, a host thread is always required to control the communication.

If an application is running on the GPU that requires frequent communication, this thread can hardly be used for other issues or set to sleep. The only way to achieve this is to allow the GPU to control the communication and completely bypass the host CPU. This will be addressed in the folowing chapters.

5. GPU-Controlled Put/Get Communication

In the previous chapter, GPI-2 for GPUs, a host-controlled, one-sided communication library for GPUs was implemented and evaluated. This model allows good overlapping of communication and computation since the complete communication overhead is offloaded to the CPU. However, this model requires context switches between the GPU and the CPU, which add overhead to the application.

Besides this overhead, a CPU thread is always required to control GPU-related communication. This CPU thread consumes additional cycles and thereby power. Furthermore, it can hardly be used for other work without performance losses. Therefore, in this chapter, we analyze *GPU-controlled* one-sided put/get communication.

To allow the GPU to control inter-node communication, the GPU must be able to source communication requests to the network interface card. In this chapter, it will be described which steps are necessary to enable this, withe the example of the Infiniband host channel adapter and the RMA unit of the Extoll device.

The only way to utilize a GPU is launching compute kernels. Therefore, all GPU-issued communication must happen within such a GPU kernel. A GPU-communication library must be implemented in a GPU programming language like CUDA or openCl. The GPU programming and thread execution model differs in some points strongly from the execution model of the host system. This requires a discussion on how a put/get API can be reconciled with the GPU thread execution model.

In this chapter, the implementation of a communication library for GPUs is discussed and described. Last but not least, it is necessary to compare the possible performance of GPU-Initiated communication with the hybrid communication/computation model regarding power and energy. Parts of this work were already published in [119, 120] and [121].

5.1. Related Work

The idea of controlling communication from the GPU was first introduced by Owens et.al in [122] with DCGN (Distributed Computing on GPU Networks), a framework that allows GPU threads to send and receive data with commands similar to MPI. An underlying MPI framework on the host actually performs the communication, but the communication calls are made within a GPU kernel. The performance of this communication library was much lower than the performance of the hybrid model. However, direct GPU- to-GPU data transfers were not supported yet and the host was required.

Still, the authors clearly state that support for direct communication among GPUs without CPU involvement is desirable. They repeat this in [123] where they claim their requirements for an MPI-version running on a GPU. However, as far as we know, currently no library exists which supports direct communication from the GPU without CPU involvement. Therefore, this is the first work implementing and analyzing the performance of GPU-controlled put/get communication, whereby the GPU controls the communication and the CPU is completely bypassed.

5.1.1. Communication libraries for the Intel Xeon Phi

For the Intel Xeon Phi architecture, several projects in this area exist. In contrast to GPUs, the Xeon Phi provides a Linux operating system, thus genuine device drivers can be developed. In [124] Si et.al. design DCFA (direct communication facility for manycore-based accelerators) which provides an Infiniband verbs interface for the Intel Xeon Phi. This interface allows the Xeon Phi to control the Infiniband network device. The host CPU is required to initialize the Infiniband HCA, but then the Xeon Phi Co-processor is enabled to directly communicate with the HCA without any further CPU involvement. For this, a special Infiniband device driver for the Xeon Phi was developed. In [125], this interface is used to create an MPI library for inter-node communication between distributed Xeon Phis and the host systems. In [126], Potluri et.al. introduce the Xeon Phi support for the MPI implementation Mvapich, optimized for Infiniband clusters. Since the PCIe peer-to-peer performance is also a bottleneck for the data transfer between the Xeon Phi and the network device, a proxy-based architecture was introduced in [92], which uses staged copies in host memory and a proxy server to perform these copies.

For GPI-2, the native support for the intel Xeon Phi was added. This GPI-2 version also uses the Infiniband verbs API of the Xeon-Phi device [127].

However, these approaches for the Intel Xeon Phi can only be partly transferred to the GPU. A GPU does not have a Linux-like operating system, so no genuine device drivers can be developed. Also, in contrast to the Xeon Phi, the GPU does not support X86 code, so the source code for a GPU communication library cannot just be compiled for the GPU.

5.1.2. Libraries for GPU computing

To allow the GPU to control the communication, a communication library is required to define a set of functions that can be called within the GPU. Most existing GPU libraries provide a CPU interface to offload special tasks to the GPU. Examples for these GPU/CPU-libraries are Thrust [128] or GPU-accelerated libraries like cuBLAS-XT[129], which support the execution of linear algebra subroutines on multiple GPUs, or cuFFT [130], a library for Fast Fourier Transformations on the GPU.

In contrast to the wide support of GPU-accelerated libraries, libraries that provide an interface with functions that can be called within a GPU kernel are hard to find. One reason for this is that CUDA supports a separate compiling and linking of device code

only since CUDA 5 [131]. This feature enables the separate development of libraries and APIs for GPU kernel which was not possible before. An example for a GPU-kernel library is CUB [132], which provides an abstraction layer for complex block-level, wrap-level, and thread-level operations on the GPU.

In [133], Stuart et.al. provide an interface for GPU-to-CPU callbacks. This interface allows forwarding requests from the GPU to the CPU. They implement three sample applications on top of this interface: a TCP/IP client and server, a memory manager for the GPU, and a command-line debugging tool that uses *printf*.

Another example of a library is GPUfs [134] which allows file system calls on the GPU. It also requires a host thread to handle request from the GPU and for system calls.

5.2. Sourcing communication requests to RDMA-capable hardware

If the GPU controls the communication, the GPU must be able to communicate with the network interface. Even though the CPU can set up the device, at least the sourcing and synchronizing of communication requests must be handled directly by the GPU, completely bypassing the CPU. However, to enable the GPU to source and synchronize communication requests, it must, first be understood how this is handled on the CPU.

Communication request are submitted to the hardware via to so-called *work requests* or *descriptors*. A work request includes all necessary information for the communication like the local and the remote memory address, the destination node, and the payload size. The network device uses this information to perform the communication while the CPU can be used for other tasks. If the communication is completed, the network device may send a completion notification to the host. To allow direct communication from user space, the user space process must be able to directly access the network device. Normally, this is done by mapping a device register to the user space with memory mapped I/O (MMIO). These registers have physical addresses within a base address R register (BAR) of the device.

How exactly a work request is submitted to the hardware is different for the various types of hardware. Here, we analyze it in more detail for Infiniband and the Extoll RMA unit.

5.2.1. Work processing on Infiniband

As explained in section 2.4.6, communication in Infiniband is handled between so-called queues. The descriptor for a communication request is submitted to a send or receive queue, a ring buffer in host memory. In a further step, the Infiniband HCA has to be notified about this new communication request. Therefore, the initiating process writes a notification to the so-called *doorbell register*. The doorbell register is a register on the Infiniband device. For user space processes, the doorbell register is mapped to the user space with memory mapped I/O. If the communication is completed, the Infiniband HCA may write a completion notification to the completion queue which can be used for local synchronization.

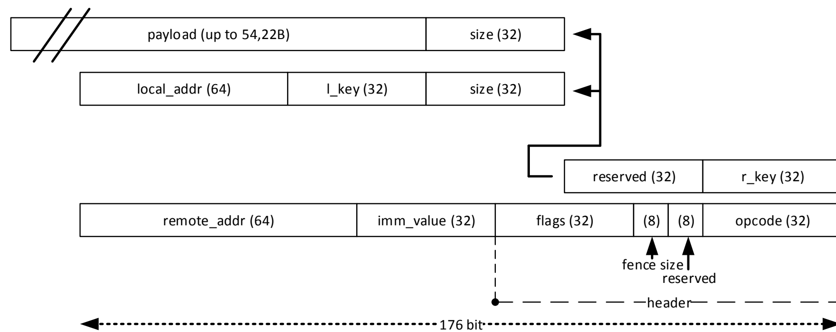


Figure 5.1.: Sample work request for Infiniband

The actual format and size of completion element and work request are device specific and also depend on the kind of work request. A remote write request, for example, requires information about the remote memory segment which is not required for a send request.

Figure 5.1 shows the structure of the descriptor for an RDMA read or write request for Mellanox Infiniband Connect-X2 devices. The first 80 bits build the header of the work request. This header is comprised of the command code (*opcode*), the *flags*, the actual size of the work request (*fence size*), and eight reserved bits. The immediate value is used for write work requests with immediate data and can be ignored otherwise. The *remote address* and the *r_key* describe the remote memory segment.

If the *inline* flag is set, the payload of a write request is directly copied to the work request. The payload is split up into segments of up to 54,228 bits (6,788 bytes). Such a *payload* segment starts with the data size followed by the data. Infiniband allows the use of multiple payload segments within a single communication request. However, the complete payload size for an inline operation is limited, normally to 64 kbytes.

If the inline flag is not set, the work request requires one or more local *segment descriptors*. Such a descriptor consists of the *size*, the *l_key*, and the *local_addr*. If more than one local segment is used, the transferred data is assembled on the remote side for a remote write and distributed between the local segments for a remote read.

Therefore, an RDMA request has to be at least the size of 352 bit but can be significantly larger if the inline operation or multiple local segments are used.

Figure 5.2 shows the format of a completion element that a Mellanox Connect2X Infiniband HCA writes to the completion queue. This completion element should not be mixed up with the `ibv_wc` structure that is returned by the `ibv_poll_cq` function. The `ibv_wc` is created after a successful polling on a completion element, using the information provided by the completion element.

Since multiple queues can share one completion queue, the first 32 bits code the QP number. The *mlpath_rqn* field is only required for UD-QPs; the *service level flag* distinguish between Ethernet and Infiniband completion elements. The *rlid* describes the remote process. The *wqe* flag is required to match the completion element with a corresponding

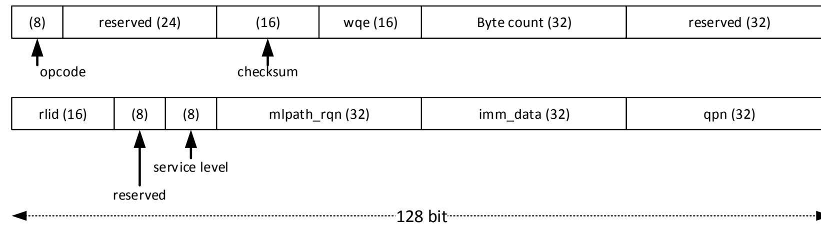


Figure 5.2.: Sample completion element for Infiniband

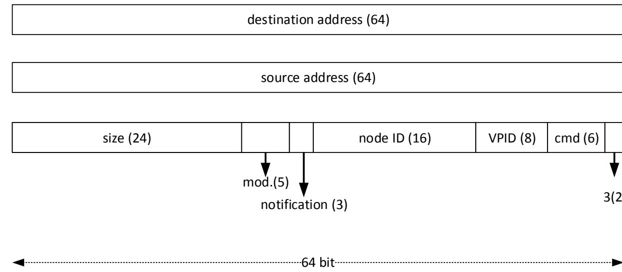


Figure 5.3.: Structure of an RMA descriptor for remote put/get commands

work request. If an error occurs, this is coded in the *opcode* field and all other bits get a new meaning. All in all, the completion element has a size of 256 bits. The Infiniband poll function polls for a new completion element and evaluates it.

5.2.2. Work request generation for the Extoll RMA unit

The work processing for the RMA unit works in a slightly different way than for Infiniband. Instead of first writing the descriptor to a queue in host memory and then writing a notification to the network interface, the descriptor is directly written into one of the registers of the Extoll device, called *requester page*. This requester page is accessible over a BAR of the Extoll device. To allow communication from the user space without entering the operating systems, this requester page is mapped to the user space with MMIO. The Extoll device provides more than one requester page. To every port that is opened, a new page is assigned.

The descriptor has the size of 192 bits (24 bytes), the structure is shown in Figure 5.3. The descriptor includes all necessary information to perform the communication. The source and the destination address can either be the physical addresses or the *network logical address* (NLA) which is created during memory registration. The physical address can only be used for physical continuous memory regions, while the use of NLAs allows the data transfer between larger virtual memory regions. Depending on the address (physical or NLA), the descriptor is written to another address within the requester page so the hardware can recognize which addressing is used.

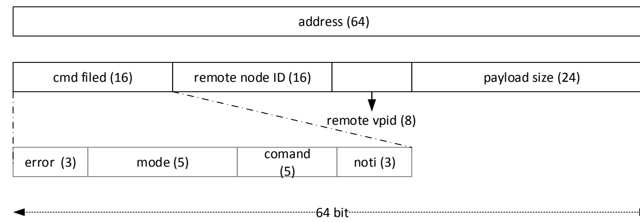


Figure 5.4.: Structure of a RMA notification for put/get commands

The *Node ID* and the *VPID* describe the destination node and port. The notification flag specifies which notifications should be created for this communication request (cf. section 2.4.7).

Notifications are used to get information about the current state of a communication request. Notifications are created by the hardware and written to pre-allocated buffers in host memory. These buffers are located in kernel space and allocated when the device driver is loaded. For user space processes, they are mapped to the user space with MMIO. Every port (or endpoint) gets its own notification queue.

A notification has the size of 128 bits. Figure 5.4 shows the structure of an RMA put notification. The address field returns the NLA or physical address that is involved with this node. The 16 bits of the command field are split up into 3 bits for errors, 5 bits, each for the modification and the command, and 4 bits that code the kind of notification. Consuming a notification does not automatically free the notification. This has to be done by the programmer. To avoid an overflow, the notifications should be freed as soon as possible.

5.2.3. Conclusion for GPU-controlled communication

Although work request processing for Infiniband and for the Extoll RMA unit differ, there are still a lot of commonalities which have to be observed to allow a GPU to use this hardware. First of all, both allow a user space process to direct access the network device. This is done by mapping a device register to the user space with memory mapped I/O. In the setup-phase, however, for both several accesses to the kernel space are required.

For both, the data transfer is handled by the hardware after a work request is submitted. The network hardware can write a notification to a queue in host memory if the communication is completed. For Extoll, the queues are located in kernel space, for Infiniband in user space. However, a user space process can consume this notifications for local synchronization for both network devices.

This means that both devices allow sourcing and synchronizing of communication requests from user space once a connection is set up. This principle also allows the GPU to directly source and synchronize communication request, as we will show in the upcomming sections.

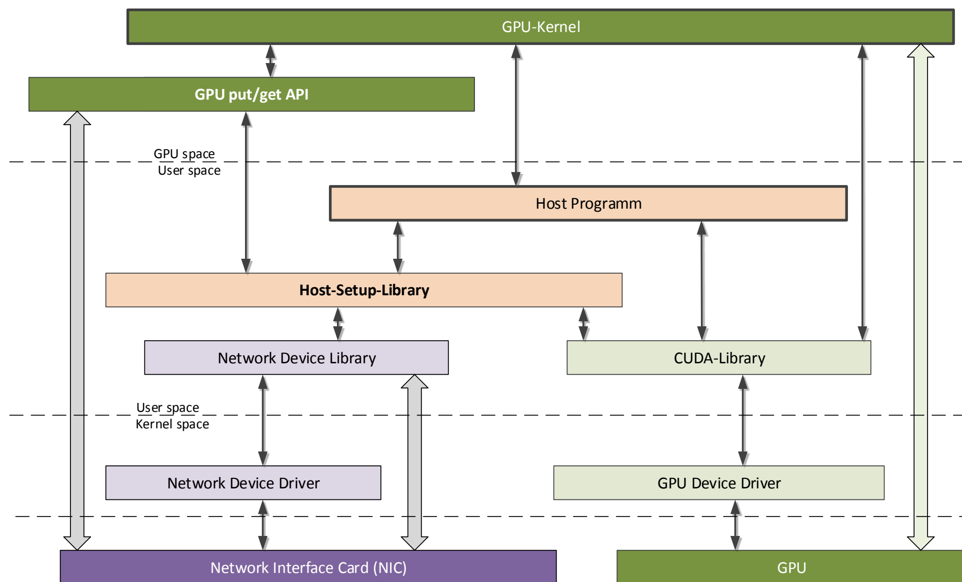


Figure 5.5.: Software stack to allow a GPU to communicate

5.3. GPU-controlled communication

The main problems for GPU-controlled communication are the GPU programming model and runtime system which are not designed for this kind of workload.

GPUs neither have a Linux-like operating system nor are they accessible in another way than launching kernels or transferring data between host and GPU. This is different on the Intel Xeon Phi, which also allows the starting of applications directly on the Xeon Phi in the so-called native-mode. A GPU always requires a host process to control it. Everything on the GPU has to be implemented in the form of a GPU kernel.

Another problem is that the network device is shared between the GPU and the CPU and the device drivers of the network device are required to manage the network device resources. To register memory and to create a connection to a remote node, oftentimes operating system accesses are required. However, the GPU cannot directly access the operating system. This has to be done by the CPU.

The Intel Xeon Phi Infiniband drivers, which are described, for example, in [125], use a proxy model. The device driver of the Xeon Phi communicates via a proxy system with the device drivers of the host. The Xeon Phi device driver forwards requests to the host device drivers. This model allows a sharing of the resources between the Xeon Phi and the host system.

However, this model cannot be used for GPUs due to the above mentioned reasons. Therefore, a GPU-initiated communication requires a host process to set up the network device, register the memory, and connect to remote nodes. Figure 5.5 shows the principle software stack for such an application.

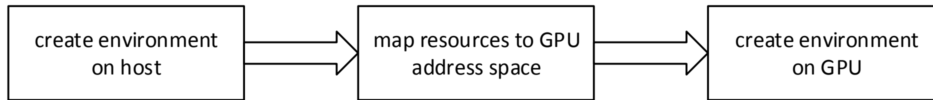


Figure 5.6.: Steps to create an communication context on the GPU

The GPU-communication library consists of two parts: one library for the GPU (*GPU put/get API*) and one for the host (*Host-Setup Library*). The host library is required to set up connections, register memory, and to manage the network device resources. It accesses the NIC library and thereby the NIC device drivers.

The host library is also required to port resources to the GPU. The functions of the API are called by a *host application* which also starts the *GPU kernel*. This kernel, however, now can use the GPU put/get API for communication with remote GPUs. This API provides direct access to the NIC by completely bypassing the host.

The next sections describe, how the individual parts of this software stack are implemented and cooperate to allow put/get communication from the GPU.

5.4. Creating a communication environment on the GPU

This section describes, how a communication environment is created on the GPU. First, we describe in general which steps are required and what must be taken into account. Then, we describe how these steps are implemented for Infiniband and for the Extoll RMA unit.

Figure 5.6 illustrates which individual steps are required to create a communication environment on a GPU. The NIC has to be initialized and connections to remote processes have to be established. To allow direct and one-sided communication, communication buffers have to be allocated and registered for the NIC and the information about the registered memory regions have to be shared between the processes. Only the CPU can do this work since it requires several accesses to the operating system.

In establishing such a communication environment, several *communication resources* are created or mapped to the user space. These resources are, for instance, the device registers, the queue buffers, or runtime variables, which describe, for example, the number of open communication requests. Parts of these resources have to be ported to the GPU to allow communication and it has to be considered which of them are required and where these resources are located.

If the CPU controls the communication, resources can either be located on host memory or on the network device. If resources are located in host memory, they can either be found in kernel space or in user space. If they are located in kernel space, they are either also only accessible in kernel space or they are mapped to the user space with MMIO. The buffers for RMA notifications, for example, are located in kernel space and mapped to the user space with MMIO.

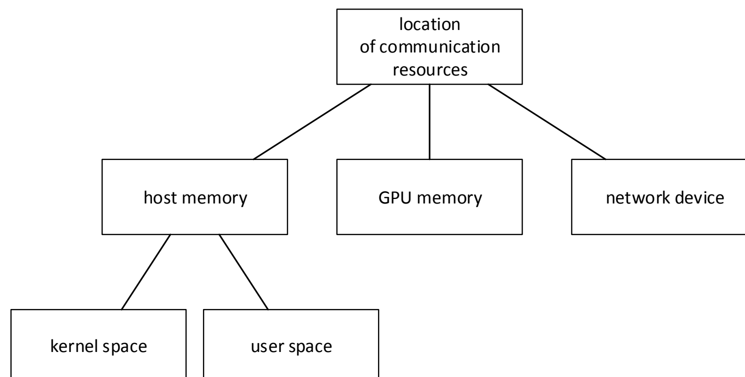


Figure 5.7.: Location of communication resources

Resources on the NIC that are required for user space communication are mapped to the user space with MMIO. This applies, for example, to the doorbell register for Infiniband or the requester page of the RMA unit.

However, if communication is controlled by the GPU, there are three different possibilities for the location of the resources, as shown in Figure 5.7.If the resources are located in GPU device memory, then they must be located in the global device memory to be visible and usable for all GPU threads.

5.4.1. Porting resources to the GPU

Bearing the possible locations for communication in mind, there are three different ways to port communication resources to the GPU, as shown in Figure 5.8.

Resources that are only accessible in kernel space can only be accessed by the CPU, because the GPU cannot access the operating system. During memory registration, for example, a host thread creates the page table for the registered memory area. After this, the page tables are not touched from the host until the memory is deregistered. Therefore, the GPU does not need a direct access to this resource. Furthermore, it is also useful to allow only the CPU to access the page table, since it is not possible to prevent simultaneous accesses from the CPU and GPU. Current GPUs do not support atomic accesses through the PCIe-bus to the host memory, which are atomic with respect to the host CPU. This may be different in the future, because atomic accesses are supported with PCIe 3. However, at the moment it is not supported and, therefore, the GPU should not be allowed to directly access resources that can only be accessed in kernel space.

Other resources can be copied to the GPU. That applies to runtime variables like identification numbers, communication properties and internal counters. In short, all data that is required for communication but neither read nor written directly from the network device.

For resources that are located on the network device or on registered host memory, the third method must be used: mapping the resource to the virtual GPU address space.

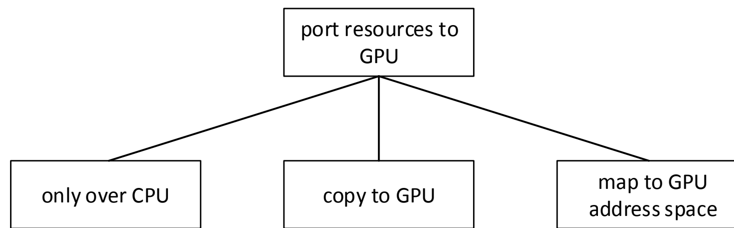


Figure 5.8.: Access to remote resources

This applies to all host-memory variables and buffers that are accessed from both, the network device and the GPU/CPU during work request generation and processing. This also applies to device-registers that are mapped to the user space with MMIO. However, as long as these resources are owned by the GPU, they should not be used by a CPU process, since it is not possible to avoid race conditions between CPU and GPU.

User-space host buffers and variables can be registered for the GPU with the CUDA function `cudaHostMemRegister`. This function registers already allocated memory for the GPU. If the registration is successful, the function `cudaHostGetDevicePointer` returns a pointer with an address in the virtual GPU address space. A GPU kernel now can access the host memory buffers directly, using this address.

However, the memory registration function of the GPU driver normally would fail trying to register an MMIO address. This applies to both, the MMIO device registers and to MMIO kernel-space buffers. This problem can be solved by using the patch described in section 3.3.3. This patch allows the registration of MMIO addresses for the network device and can also be used for the GPU. Although most of the GPU device driver is prohibitive and cannot be patched, the memory registration function must be linked against the operation system. Therefore, this part of the Nvidia GPU device driver can be patched with an appropriate patch.

If these steps are done, the mapped addresses and the replicated elements on the GPU can be used to create the communication interface on the GPU. For this, the required information has to be copied to the GPU, using CUDA kernels.

The next section describes in more detail, how an Infiniband communication environment is created for the GPU under the above-mentioned conditions. The subsequent section describes this for the RMA unit of the Extoll device.

5.5. Creating an Infiniband communication environment on the GPU

This section describes how an Infiniband communication environment is created on the GPU. Furthermore, the performance of this interface is evaluated and potential bottlenecks are identified and – if possible – solutions for these bottlenecks are developed.

Table 5.1.: List of resources that are required to create an Infiniband connection

element	description	resources
`ibv_context`	Infiniband user space context	doorbell register, QP-table, run-time variables
`ibv_pd`	protection domain	–
`ibv_cq`	completion queue	queue buffer, queue counter, run-time variables (e.g., queue size, ID,...)
`ibv_qp`	queue pair	send and receive queue buffer, counter, property variables (e.g., queue size, ID, ...)
`ibv_mr`	registered memory region	property variables (address, size, keys)

5.5.1. Context Setup on the Host

In the first step, the Infiniband communication context is set up on the host. Here, only a short introduction to this is given. A more detailed description of the setup routines for Infiniband connections can be found in, for example, [135].

Table 5.1 shows the list of Infiniband resources that are required to set up a connection and to allow one-sided communication.

First, the Infiniband device is set up and an *Infiniband user space context (ibv_contex)* is created. For our purposes, the most important part is the doorbell register that is assigned to this user space context and mapped to the user space with MMIO. Every user space context gets its own doorbell register.

The next required element is the *protection domain (pd)*. Protection domains allow the association of multiple resources like completion queues and queue pairs within a single domain of trust. To avoid race conditions, only one protection domain is created for all GPU communication resources and the same protection domain should not be used for host-controlled communication.

However, the *protection domain* itself mostly exists on the Infiniband hardware. The user space data structure only contains a pointer to the user space context and a handle. This handle is only required for initialization and the allocation of other Infiniband elements. Therefore, it does not have to be ported to the GPU.

For this protection domain, then the *completion queues (ib_cq)* and *queue pairs (ibv_qp)* are created. The creation includes the allocation and registration of the queue buffers.

We use reliable connection (RC-)QPs for GPU-controlled communication. Unreliable connection QPS would add overhead to the GPU due to software based reliability checks. The lacking support for one-sided communication types of the datagram QP types led us to the use of RC-QPs.

An RC-QP has to be connected to another RC-QP since the communication is handled between these two QPs. Currently, QPs have no means to locate each other. Therefore, a TCP/IP connection is opened to exchange the information that is required to

connect the QPs. This information includes the LID, QPN, and PSN. The LID is the unique *local identifier* for an active Infiniband port. The QPN is the queue pair identification number. The PSN (*Package Sequence Number*) is required for reliable connection QPs to verify that packages arrive in the correct order and that no packages are missing. To establish a virtual connection between two QPs, these variables are exchanged and forwarded to the kernel space where the device driver forwards them to the Infiniband hardware.

Last, to allow communication, communication buffers have to be allocated and registered for the network device. This means that an Infiniband memory region *(ibv_mr)* is created. For GPU-controlled communication, these communication buffers are allocated in GPU memory, therefore the support of GPUDirect RDMA is required.

To allow one-sided communication, the information about these memory segments (*virtual address, size* and *memory key*) has do be exchanged between the processes. For this, the TCP/IP connection is used.

After the setup-phase on the host, communication in Infiniband can be handled without entering the operation system kernel. Therefore, it is possible to port existing Infiniband resources to the GPU and source and synchronize network traffic by completely bypassing the host CPU. However, it must be ensured that communication resources like QPs or the doorbell registers are not used by the host system as long as they are used by the GPU, otherwise a simultaneous access cannot be avoided.

In the next step, communication resources that are located in host memory or on the device and are accessed from both the GPU and the Infiniband device are mapped to the GPU address space. These resources are the doorbell register, the queue buffers, and some counter variables that indicate the position in the queue buffers for the network device and the GPU.

5.5.2. Creating Infiniband elements for the GPU

In the last step, the Infiniband elements are created on the GPU and the information about these resources is copied from the CPU to the GPU. To allow the GPU to source and synchronize communication requests, the user space context `(ibv_context)`, the queue pairs `(ibv_qp)`, and the completion queue `(ibv_cq)` require a counterpart on the GPU.

To allow RDMA communication, also a structure for local and remote memory regions is required to manage the virtual addresses, the memory keys, and the size of the segments. For local memory regions, these regions correspond to the Infiniband memory regions `(ibv_mr)`.

These last steps also includes the copying of variables and pointers to the GPU. Listings 5.1 and 5.2 show, in a simplified example, how this is realized for a queue pair on host and GPU.

The GPU device memory is managed from host with `cudaMalloc`. Therefore, first the memory for the new GPU element has to be allocated, as shown in line 4 of Listing 5.1. Next, a GPU kernel is launched to set the variables on the GPU.

Listing 5.1: Creation of a QP for the GPU, host function

```
struct gpu_qp* export_qp_to_gpu(struct ibv_qp *qp,
                                struct ibv_gpu_ctx *ctx){          2
  ...
  cuErr= cudaMalloc((void**)&new_gpu_qp, sizeof(struct gpu_qp));   4
  /*allocate memory for the new GPU QP*/
  if(cuErr!=cudaSuccess){                                          6
    ··· /*error handling*/
  }                                                                8
  init_qp<<<1,1,0,0>>>(new_gpu_qp, qp->buf_size, qp->qpn,
                     m_send_queue, m_recv_queue ctx);              10
  /*port resources the QP*/
  cudaErr = gutaGetLastError();                                    12
  if(cuErr!=cudaSuccess){
    ···  /*error handling*/                                        14
  }
  cudaDeviceSynchronize();                                         16
  ...
  return new_gpu_qp;                                               18
}
```

Listing 5.2: Creation of a QP for the GPU, GPU kernel

```
__global__ void init_qp(struct gpu_qp *new_qp, int buf_size,       1
                        int qp_num, void* send_q, void *recv_q,
                        struct ibv_gpu_ctx* ctx){                  3
  new_qp->buf_size = buf_size;
  new_qp->qp_num = qp_num;                                         5
  new_qp->send_queue = send_q:
  new_qp->recv_queue= recv_q;                                      7
  new_qp->ctx = ctx;
}                                                                  9
```

The kernel launching is shown in line 7 of Listing 5.1. The parameters `m_send_queue` and `m_recv_queue` describe the mapped addresses of the queue buffers. The GPU kernel itself is shown in Listing 5.2. For every Infiniband GPU element, at least one kernel has to be called and synchronized to set the values on the GPU. Therefore, setting up the Infiniband context on the GPU is a very time-consuming operation.

5.5.3. Infiniband interface on the GPU

To use these Infiniband elements on the GPU, a part of the Infiniband verbs API is ported to the GPU: `ibv_post_send`, `ibv_post_recive`, and `ibv_poll_cq`. These three functions are sufficient to source and synchronize communication requests.

The CUDA C language allows an easy porting of these library functions to the GPU, since most of host code can be reused. Still, for better performance, some GPU-specific optimization is required. The creation of the work request is a sequential workload,

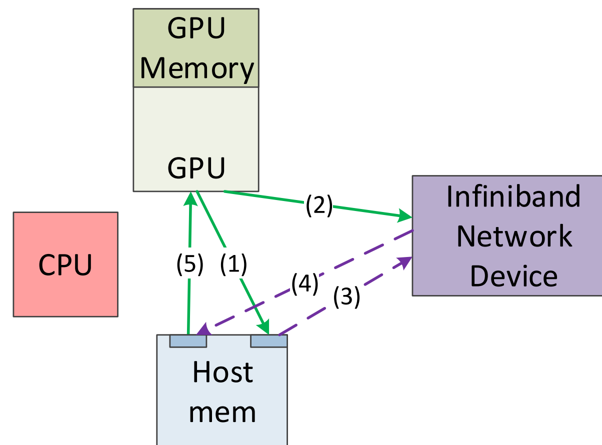


Figure 5.9.: PCIe accesses to source and synchronize communication requests on the GPU with queue buffers on host

and GPUs are optimized for data-parallel code execution. Therefore, most of the work request generation can only be performed by a single thread and there is virtually no optimization strategy for single-threaded workloads on the GPU, expect avoiding them.

However, the host version of `ibv_post_send` handles a lot of different cases, depending on the work requests and the used QP, which results in a lot of `if` and `switch/case` branches. This is not very efficient on a GPU. Therefore, we reduced the functionality on the GPU to a minimum, only supporting RC-QPs to get a more efficient version.

We also decided not to support inline operations on GPUs. For inline operations, the data are directly copied to the work request so the network device does not have to read them with further DMA access from the communication buffer. This works for small messages on host very well. On the GPU, memory copy operations that are performed by a single thread are not very efficient. Also, since the send queue is located on host memory, this would require further memory accesses through the PCIe-bus and, therefore, nothing would be gained by using the inline-operations on the GPU. On the other hand, not supporting inline operations avoids further branch divergences. A further optimization is the location of the queue buffers.

Allocation of the queue buffers

Normally, the queue buffers are allocated in user space memory on the host and registered for the Infiniband device during QP and CQ creation. Then, they are mapped to the GPU address space to allow access within a GPU kernel. Figure 5.9 shows how many accesses via the PCIe-bus are at least required to source and synchronize a communication request if the queues are located on host memory.

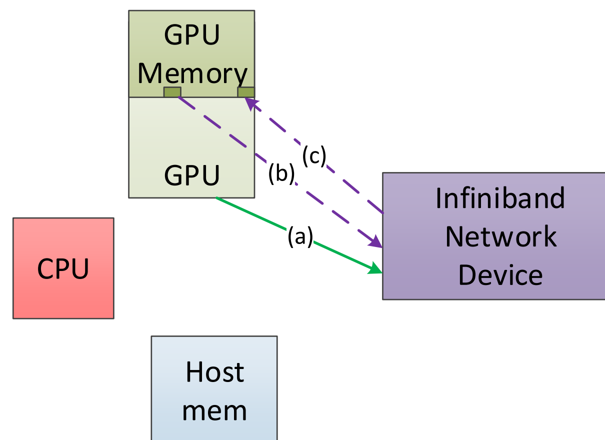


Figure 5.10.: PCIe accesses to source and synchronize communication requests on the GPU with queue buffers on GPU

Note that this is only a simplified illustration of the process. For the actual data transfer, further accesses through the PCIe-bus are required, for example, to incremented a special counter that marks the position in the queue. Also, the submission of work requests requires more than one access to host memory. However, for overview purposes, we omit these accesses here. The actual data transfer is also omitted.

First, the GPU writes the work request to host memory (1). This requires at least one write access through the PCIe-bus. Then, the GPU writes to the doorbell register to trigger the data transfer (2). Next, the network device reads the work request from the host (3) and starts the data transfer by reading or writing the data directly from the GPU memory. If the communication is completed, the Infiniband device writes a completion notification to host memory (4). To verify the completion, the GPU reads the completion from host memory (5).

Another possibility is to allocate the ring buffers directly in GPU memory. Figure 5.10 shows how many accesses through the PCIe-bus are required in this case. The GPU writes the work request to the local device memory, so here, no accesses via the PCIe-bus are required. In order to inform the network device of the new work request, a GPU thread writes to the doorbell register (a) and the network device reads the work request from GPU memory (b). If the data transfer is completed, the Infiniband device writes the completion directly to GPU memory (c), where a GPU thread can consume the completion from device memory without further accesses via the PCIe-bus.

Therefore, the allocation of the queue buffers in device memory reduces the accesses through the PCIe-bus. We implemented new versions of the Infiniband user space library functions `ibv_create_cq` and `ibv_create_qp` which allocate the buffers on GPU memory instead of host memory.

The queue buffers in GPU memory have to be registered for the Infiniband device. This requires a small modification of the peer-memory device driver described in section 3.3.2. The peer-memory driver allows only the registration of GPU memory regions, which are used as communication buffers, but not for queue buffers. However, the device drivers forward the virtual address to the same user-space memory registration function in kernel space for both cases, but this function normally does not check, if the address may belong to a peer client for queue buffers . Apart from this, the registration process is the same. Therefore, the Infiniband device drivers were adapted to allow also the registration of queue buffers in GPU memory.

The modification enables the registration peer-memory clients for Infiniband user-space contexts. If a new user space context is created, the device driver checks if a GPU peer-memory client exists. If so, the peer memory-client is registered for this user-space context. If the device driver now tries to register a queue buffer, the peer-memory client checks, if the memory address belongs to this peer-memory client and registers it in the same way as communication buffers.

However, GPU memory is a scarce resource and every QP requires at least one additional ring buffer. Furthermore, for every connection to a remote GPU, at least one new QP is required. The memory footprint of the QPs is already a known problem for host processes in Infiniband [73]. Since GPUs have less memory, this problem is compounded for queues in GPU memory. Therefore, it may be unavoidable for larger systems to allocate the queue buffers on host memory . However, we implemented both versions to validate the influence of the location of the buffers to the performance.

5.5.4. Micro-benchmark results for Infiniband

In this section, some micro-benchmarks are used to analyze the basic performance of GPU-vontrolled communication using Infiniband. We compare four different communication mechanisms:

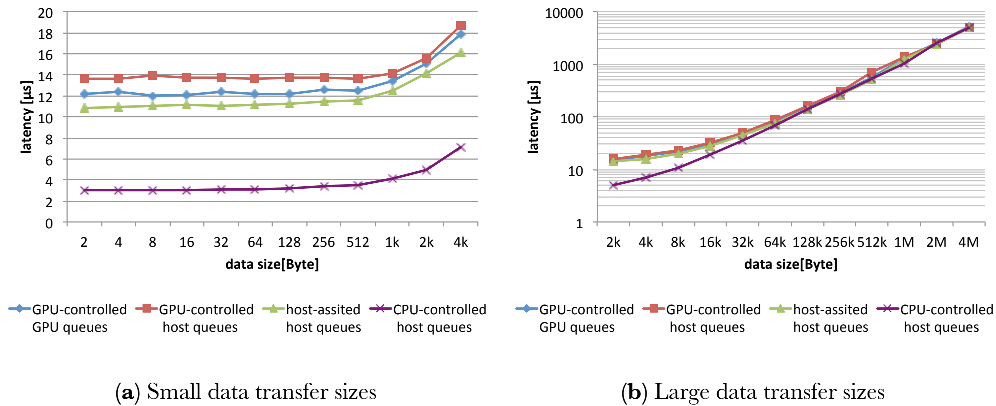
GPU-controlled, queues on GPU The communication is controlled by the GPU, the queue buffers are allocated in GPU memory.

GPU-controlled, queues on host The communication is controlled by the GPU, the queue buffers are allocated in host memory.

Host-controlled, queues on host The CPU controls the communication, no CUDA kernel is started, the queue buffers are allocated in host memory.

GPU-host-assisted, queues on host Like host-controlled, but a GPU kernel is started and the GPU triggers the communication.

The host-controlled version uses the GASPI interface for communication since it adds minimal overhead to the application. The host-assisted version is similar to the approaches described in [122] for MPI or [133] for TCP/IP connections. Here, the GPU writes to a flag in host memory to initiate a data transfer while an underlying framework on the host actually controls the NIC. By this approach, kernel synchronization can be



(**a**) Small data transfer sizes (**b**) Large data transfer sizes

Figure 5.11.: Latency of a GPU device to device transfer, initiated on host or GPU, using Infiniband

avoided. However, to get rid of the MPI or TCP/IP overhead, GASPI is used as a communication layer on the host. This approach is described in more detail in [119].

Allowing the GPU to control communication requires a lot of changes in the device drivers and Infiniband user space libraries, so we were not able to run the Infiniband benchmarks on a system with more than two workstations, equipped with two Xeon X5660 six core CPUs, one K20c GPUm and connected with Mellanox Connect X3 Infiniband. However, to analyze the capabilities of GPUs to source and synchronize network traffic, a test system with two GPUs is sufficient. As micro-benchmarks, we measure the latency, the bandwidth, the performance of atomic operations, and the message rate.

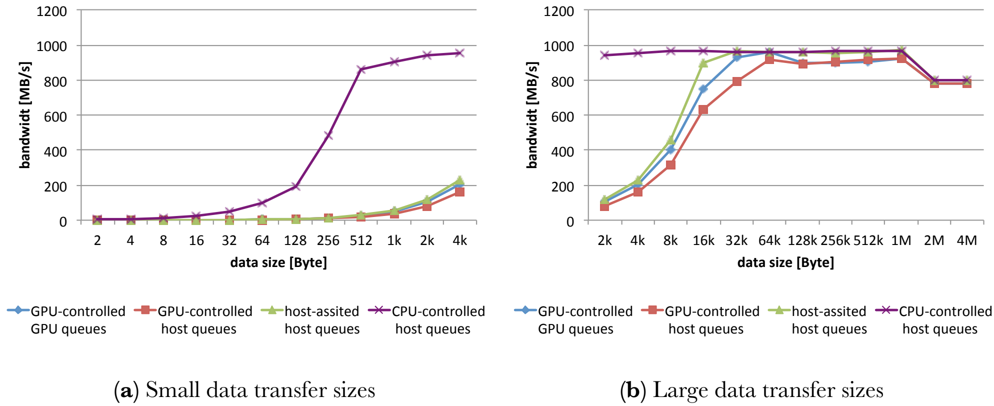
Latency test

Our first test is a simple *pingpong benchmark* to measure the latency of a GPU-to-GPU data transfer. For the GPU-controlled and host-assisted benchmarks, for every transfer size one GPU kernel with a single block of 64 threads is started on the GPU. This kernel runs the pingpong test for a given number of iterations.

We use a remote write operation to directly transfer data to the memory of a remote GPU. On the remote GPU, a thread polls on the last transferred byte for a change. This is possible since, for reliable connections, the data are transferred in the correct order. If the data are completely transferred, this last byte is reset and the data transfer in turn is started.

We measure the complete run time of this kernel and divide this by the number of iterations. If the number of iterations is large enough, the kernel launching and synchronization overhead can be neglected. The benchmark runs with 1,000 iterations.

The host-controlled version uses GASPI notifications for remote synchronization to avoid polling on GPU memory from host. Here, no GPU kernel is started. For the sake of clarity, we only use `gaspi_write_notify` for this benchmark. The performance



(**a**) Small data transfer sizes (**b**) Large data transfer sizes

Figure 5.12.: Bandwidth of a GPU device to device transfer, initiated on host or GPU, using Infiniband

difference between `gaspi_write_notify` and `gaspi_gpu_write_notify`, which means the difference between direct and staged copies, was discussed in detail in section 4.3.8.

For the host-assisted version, the helper thread on the host uses `gaspi_write`. As for the GPU-controlled communication, on the GPU a threads polls for the last transferred byte before a new communication is triggered.

The results for the half round latency can be seen in Figure 5.11. They show that GPU-controlled communication is performing poorly compared to host-controlled communication, especially for small data transfer sizes.

The difference between GPU queue buffers and host queue buffers lies around 1 μs for smaller data transfer sizes. The minimal latency, however, lies around 12.7 μs or 13.6 μs, which is much higher than the latency for a host-controlled communication, which is approximately 3 μs. Even the host-assisted version has a latency of around 12.2 μs, which is still faster than controlling the communication directly on the GPU.

The absolut difference between the four methods stays constant. For larger messages, the data transfer latency is the dominating factor, therefore the difference between the latencies become smaller.

Bandwidth

For the bandwidth benchmark, a number of remote write requests are subsequently submitted to one QP. If the send queue of this QP is full, the GPU or the CPU waits for the completion of these requests by polling on the completion queue. For GPU-controlled communication, a kernel is started to submit the communication request; for the host controlled version, no GPU kernel is started and the complete communication is handled by the CPU. The host-controlled and the host-assisted version both use `gaspi_write`, transferring data directly between GPU memory segments. The results are shown in Figures 5.12a and 5.12b.

For small messages, the bandwidth of a host-controlled data transfer is much higher than the bandwidth of a GPU-controlled data transfer. This corresponds to the results

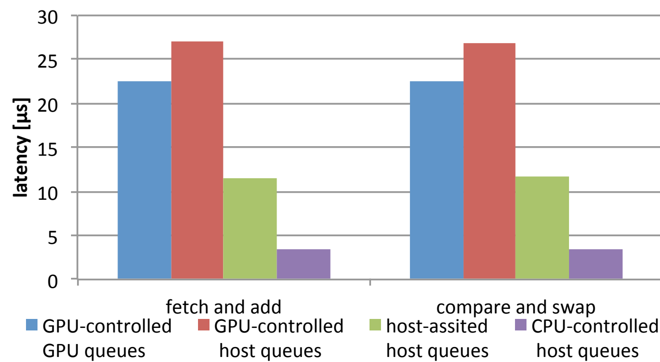


Figure 5.13.: Latency of atomic operations on GPU memory, initiated on host or GPU, using Infiniband

from the latency benchmark. The host-controlled version reaches its peak bandwidth of 980 MByte/s for a message size of 4 kb, while for the GPU-controlled versions, the maximal bandwidth is not reached for data transfer sizes smaller than 32 kb. Allocating the queues in GPU memory leads to a slightly better bandwidth for GPU-controlled communication for medium sized-messages.

The peak bandwidth for the GPU-controlled version is also slightly worse then the peak bandwidth of the host- controlled communication. Again, the PCIe problems for a direct device-to-device transfer can be observed here, especially the decline of the bandwidth for data sizes larger than 1 MB. By using staged host copies for larger transfer sizes, the difference between a GPU-controlled and a host-controlled data transfers would be even larger.

Atomic operations

Since Infiniband supports atomic operations on registered GPU memory, we also measured the latency of Infiniband atomic operations in GPU memory. The results are shown in Figure 5.13.

Again, GPU-controlled communication is performing poorly compared to host- controlled communication. Even the host-assisted version, which requires a context switch between GPU and CPU, is almost three times faster than GPU-controlled communication. However, this result follows the results of the previous benchmarks.

Message rate

As a last micro-benchmark we run a message rate test. We run the test with one QP connection and with 16 QP connections between the two GPUs.

For the tests with 16 QP connections, on the GPU, a kernel with sixteen thread blocks is started. Every block can use its own QP connection, so the communication requests can be submitted in parallel. On the host, for every connection, a CPU thread is started.

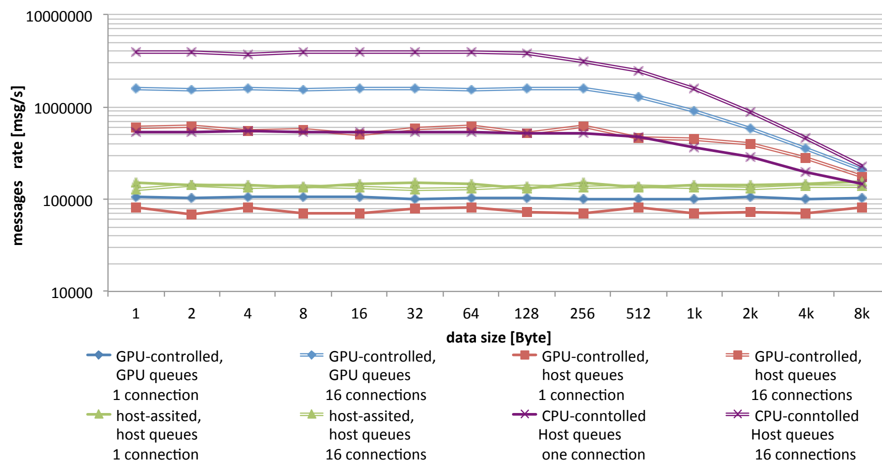


Figure 5.14.: Message rate of a GPU device to device transfer, initiated on host or GPU, using Infiniband

Every thread uses its own GASPI queue, so on the CPU, the communication requests are also submitted in parallel.

The host-assisted version starts a kernel with one block for every connection. The underlying GASPI-communication interface uses the same number of queues. However, in contrast to the host-controlled communication, a single thread serves all GPU-communication requests. Figure 5.14 shows the results of the message rate benchmark.

If the communication is controlled by the GPU, for a single connection, the message rate is very low and nearly constant for all message sizes . The same applies to the host-assisted version, although the message rate is slightly higher then the message rate of the GPU-controlled communication. Using more connections, the message rate increases for all communication methods apart from the host-assisted communication.

Especially the message rate for GPU-controlled communication with GPU queues increases almost linearly with the number of connections. For messages up to 256 bytes, the message rate that is achieved with sixteen connections (ca. 1.5 million messages per second) is fifteen times higher than the message rate that can be achieved with one connection (ca.100,000 messages per second).

However, if the queues are located in host memory, the message rate does not increase that much at all. With sixteen connections on the GPU and queues in host memory, the GPU reaches the same message rate as the CPU with one connection. For the host-assisted version, the message rate does not increase at all for more connections.

We repeated the message rate benchmark with a constant data size of 32 bytes and varied the number of connections between 1 and 32 queue pairs. The results are shown in Figure 5.15.

The message rates for GPU-controlled communication with GPU queues and host queues diverge more for a larger number of connections. While for a single connection,

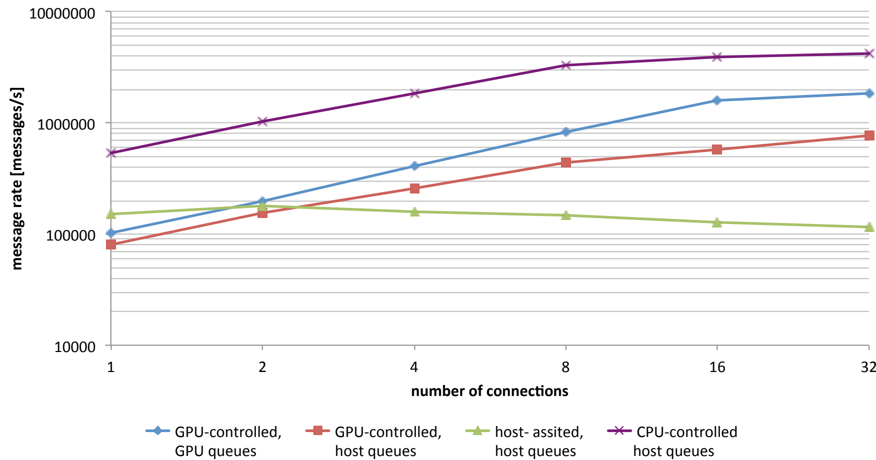


Figure 5.15.: Message rate of a GPU device-to-device transfers for a data transfer size of 32 bytes and varying numbers of connections, initiated on host or GPU, using Infiniband

the transfer with most queues reaches around 80% of the message rate of the transfer with GPU queues, for 32 connections it only reaches 40% of the message rate.

Here, the PCIe-bus is the bottleneck. If more connections are used, more thread-blocks try to access the queues in host memory over the PCIe-bus in parallel and this slows down the message rate.

For the host-assisted version, the message rate goes down for more connections. The reason for this is that all blocks on the GPU are served by the same host thread. If the host thread serves a request from one thread block, this host thread is blocked for all other aspirants. This results in a blocking state for the GPU and slows down the message rate.

5.5.5. Analysis and optimization

The results of the micro-benchmarks are quite disappointing, especially for small message sizes. Therefore, a more detailed analysis of the communication process is required.

The time fraction spent for data transfer on the network should be identical for all methods, regardless of if the communication is initiated on the GPU or on the host. Therefore, the additional latency must be caused by the communication control, the *communication overhead*. Essentially, communication control is done by the functions `ibv_post_send`, which initiates the communication and `ibv_poll_cq`, which waits for the completion of the communication. Therefore, we measure the run time of these functions on both GPU and host. The results are shown in Table 5.2. Note that the benchmark for `ibv_poll_cq` guarantees that the polling for the completion is successful, so we only measure the runtime of a successful handling of the completion element.

It takes more than 100 times longer to create a work request and add it to the queue on the GPU than it does on the host. While the network request generation can be

Table 5.2.: Latency (in μs) associated with communication control functions in Infiniband, excluding data transfer latencies

	CPU-controlled host queues	GPU-controlled GPU queues	GPU-controlled host queues
`ibv_post_send`	~ 0.01	10.66	12.46
`ibv_poll_cq`	~ 0.01	15.69	8.27

Table 5.3.: Number of memory accesses, cache misses and, instructions for Infiniband functions on the GPU

	queues on host		queues on GPU	
	ibv_post_send	ibv_poll_cq	ibv_post_send	ibv_poll_cq
host read	147	586	0	112
host write	1,404	101	197	1
device read	1,550	39,332	1,550	30,332
device write	1,600	6,700	1,600	6,700
instructions	46,918	17,158	46,918	17,158

neglected on the CPU, it exceeds the data transfer latency on the GPU. Also, the polling and handling of the completion queue elements is very time consuming, especially if the completion queue is located in host memory. There are two explanations for this:

First of all, only a single thread can create work requests and the single thread performance of GPUs is very low. Second, the work request generation requires several accesses to the global device memory, respectively the host memory. A single thread performs these accesses, so these fine-grain accesses cannot be coalesced into a larger one.

Analysis with the GPU performance counter

To verify the assumptions about the bottlenecks of GPU-controlled communication and to get a better understanding of the behavior, we use the Infiniband performance counter to measure the number of instructions and accesses to device and system memory that are required to source and synchronize communication requests on the GPU.

We measure the instructions and memory accesses for one hundred executions of `ibv_poll_cq` and `ibv_post_send` on the GPU. We use a hundred iterations since kernel launching and preparing of the communication functions require some additional accesses whose impact on the results should be minimized. The benchmark guarantees that the polling for a completion element is successful at the first try, so there are no additional instruction or accesses to memory caused by long-term polling. The results are shown in Table 5.3.

Almost 47,000 instructions are required to submit 100 communication requests to the Infiniband HCA from the GPU. This means, that roughly 470 instructions are needed for a single communication request. Additional, 170 more instructions are required

to synchronize a request locally. Most of these instructions are performed by a single thread, because there is hardly any data parallelization.

The number of accesses to host memory explains best the performance differences between GPU-queues and host-queues. Over one thousand accesses to host memory are required, if the queues are located there, to submit the communication requests, while for GPU-queues, only around 200 accesses are required. These accesses cannot be coalesced since they are performed by a single thread. The high number of host memory read accesses (ca. 39,000) explains the long duration of `ibv_post_cq` with host-queues. Read accesses from the GPU to host memory have a very long latency.

Optimization and simplification

Based on this analysis, an optimization strategy should reduce the number of instructions and accesses to system memory. Necessary accesses to systems memory should be coalesced if possible. Therefore, we use the following steps to optimize sourcing and synchronizing of communication requests:

Shared memory The work request first created in shared memory. Then multiple threads in parallel copy the request to the send queue, thereby leveraging the coalescing capabilities of the GPU's DMA engine and avoiding multiple small accesses to system memory. For the completion element, the same is done. Multiple threads copy in parallel the completion to shared memory and only then it is evaluated.

Static calculation The Infiniband HCA requires a big-endian-to-little- endian conversion for the work request elements. For the memory keys and static variables, this can be done during initialization and does not have to be repeated for every new communication request. However, for the source and destination address as well as the data transfer size, the values cannot be calculated statically since they may differ for every communication request. By this, the number of instructions can be reduced.

In addition to the optimizations, in the next step, the Infiniband functions are simplified by removing some of thefunctionalities. The goal of this is to estimate the overhead that these functionalists cause.

No stamping Consumed send queue entries are stamped during work request generation to avoid prefetching from the network device. If the queues are located on host memory, this requires additional accesses through the PCIe- bus. In order to assess the overhead that is caused by this stamping, we omit this. This reduces both the number of instructions and memory accesses. However, since the queue buffer is a ring buffer, this could mean that the Infiniband device wrongly reads an old work request in this buffer by trying to prefetch communication requests. This could result in an error. However, like mentioned above, this is omitted primarily to estimate the overhead that is caused by the stamping on the GPU.

No queue and parameter handling This is the most drastic step. The *ibv_post_send* send function normally does some parameter checks at the beginning and also checks if an overflow in the queue occurs. For optimization and to assess the overhead, we omit this.

The *ibv_poll_cq* function normally not only polls for a new completion element but also interprets the completion element. One step during this interpretation is finding the associated QP in a list of all QPs connected to the user space context. If the QP is found, the index of the send queue buffer of the QP is updated. This means that the buffer entry is freed and can be used again for new work requests. For analysis purposes, this is also omitted. This has little impact on the experiment since the overflow check is omitted anyway. Thereby, we can reduce the functionality of *ibv_poll_cq* to a minimum. The reduced polling function only checks for a new completion and for errors. Copying of the request to shared memory can be omitted. However, if more information about the completion is required, another function can be used to interpret the completion element. This step again reduces the number of instructions and accesses to host memory.

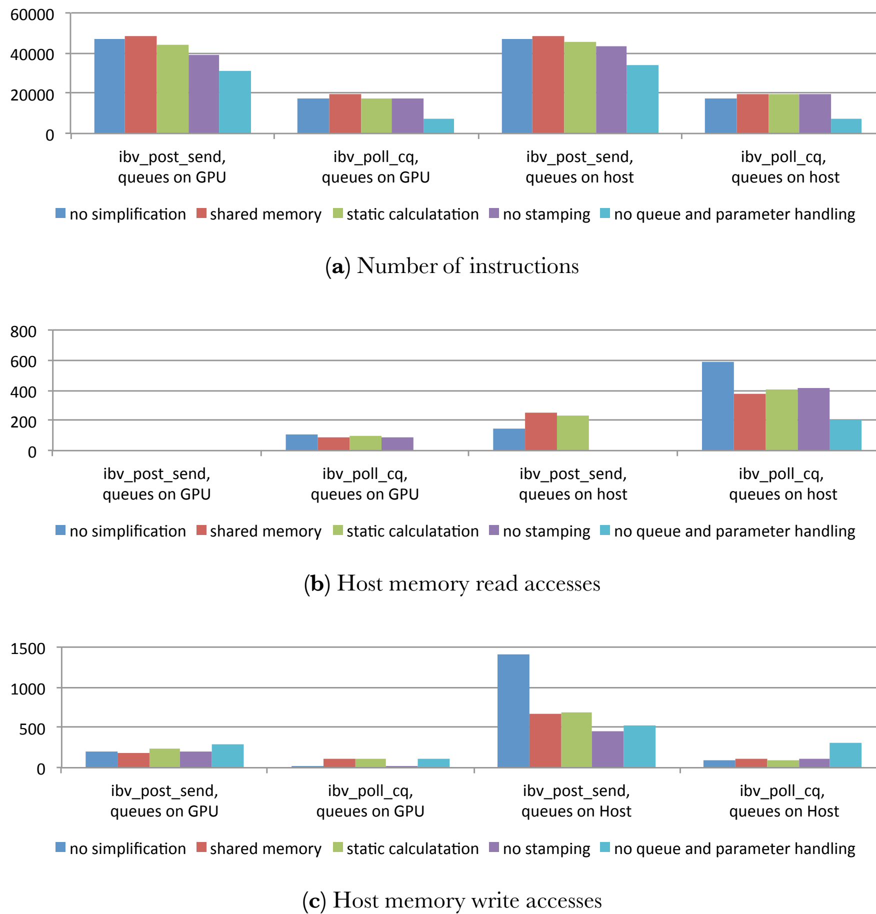
In order do determine the influence of the individual simplification steps, again the runtimes of *ibv_post_send* and *ibv_poll_cq* are measured. The results are presented in Table 5.4. The use of shared memory is valuable if the queues are located in host memory, especially for the completion handling. If the buffers are located in GPU memory, no improvement can be achieved by using shared memory.

The results show that most of the speedup is gained by the last two steps. The stamping of old work requests requires additional accesses to the host or system memory, depending on the location of queues, while the parameter checking and queue handling requires a lot of additional instructions, which can only be performed by a single thread.

Figure 5.16 shows how the number of instructions and accesses to host memory change for the different simplification steps. Using shared memory at first increases the number of instructions. Therefore, there is no benefit if the queues are located in GPU memory. However, if the buffers are located in host memory, the accesses to host memory can be reduced, especially the host memory read accesses for *ibv_poll_cq*. This explains the

Table 5.4.: Latency (in μs) of simplified and optimized versions for sourcing and synchronizing communication requests on the GPU

	queues on host		queues on GPU	
	ibv_post_send	ibv_poll_cq	ibv_post_send	ibv_poll_cq
No optimization	12.46	15.69	10.66	8.27
1. Shared memory	11.86	10.70	10.86	8.27
2. Static calc	11.83	10.70	10.56	8.20
3. No stamping	9.50	10.70	8.80	8.20
4. No parameter/ queue handling	7.52	5.45	7.00	3.12



(**a**) Number of instructions

(**b**) Host memory read accesses

(**c**) Host memory write accesses

Figure 5.16.: Results of the performance counter for the different simplification steps in comparison

reduction from 15.7μs to 10.7μs for the runtime of this function using shared memory. For the *ibv_post_send* function, the number of write accesses to the system memory is halved if the queues are located in host memory.

Using static calculated keys only reduces the number of instructions minimally, so there is hardly any performance gain. Omitting the stamping of old work requests has no influence on the *ibv_poll_cq* function. However, for *ibv_post_send*, the number of instructions and the accesses to system memory are clearly reduced. Read accesses to host memory can completely be avoided with this step, even if the queues are located in host memory.

The last step reduces the number of instructions for both functions the most. This, most likely, explains the shorter runtime. For *ibv_poll_cq*, additional read accesses to host memory are reduced massively, which saves further time.

Summary for Infiniband verbs

The results of the micro-benchmarks show, that the overhead of creating and synchronizing work requests on the GPU is much higher than on the host. The evaluation of the performance counter and different simplification steps show that the reason for this is the comparatively complicated Infiniband verbs protocol, which requires a lot of sequential work that cannot be parallelized. This work includes the stamping of old work request, error checking, and completion handling.

A suitable communication interface for GPUs should allow a simpler creating of communication requests and also a simpler local synchronization of theses communication requests.

5.6. Creating an RMA environment on the GPU

This section describes which steps are required to create an environment to allow the GPU to source and synchronize communication requests to the RMA unit of the Extoll device from the GPU.

In principle, the same steps are required as for Infiniband. A detailed description of this can be found in [80]. The aim of this section is mainly to analyze the differences and commonalities to the Infiniband implementation and to get a better understanding of what kind of communication interface is best suitable for GPUs.

5.6.1. Setting up an RMA connection on host

As for Infiniband, first, the host CPU has to establish connections between the endpoints, then, the required resources are ported to the GPU. Listing 5.3 shows how a connection is established using the RMA unit of the Extoll device. The procedure is much simpler than for Infiniband. First, a so-called port (or endpoint) is opened. Every endpoint identifies itself with a *node_id*, which is unique for the Extoll device, and a *vpid*, which is unique for an endpoint on the respective node.

If the port is opened, a *requester page* and a *notification queue* are mapped to the user space with MMIO. The *requester page* is a window in one of the BARs of the Extoll device. The *notification queue* is a buffer for notifications, llocated in kernel space. To allow communication, one or more virtual connections to another endpoint have to be established with the `rma2_connect` function. For this, the node_id and the vpid of the remote node are required, so they have to be exchanged with the remote processes. For this, a TCP/IP connection, like for Infiniband, can be used.

However, the connection to a remote process is a *real virtual connection*. This means that it is only a software abstraction to make the addressing and mapping of operations simpler. The connections of one port use the same requester page and notification queue. So, to allow communication with a remote endpoint, the *vpid* and the *node_id* of a remote port have to be copied to the GPU.

To allow one-sided communication, the communication buffers are allocated on the GPU and registered. We use the NLA to represent a registered GPU memory region. To

Listing 5.3: Creating an RMA connection

```
  RMA2_Port port;
  RMA2_ERROR rc;                                                    2
  RMA2_Handle handle;
  uint16_t local_node,  destination_node;                           4
  uint16_t local_vpid,destination_vpid;
   ...                                                              6
   rc=rma2_open(&port);
   if (rc!=RMA2_SUCCESS)                                            8
    {
      /*Error handling */                                           10
    }
/* exchange node_id and vpid*/                                      12
   [...]
/* establish a virtual connection */                                14
 rc=rma2_connect(port, destination_node,destination_vpid,
    RMA2_CONN_DEFAULT, &handle);
   if (rc!=RMA2_SUCCESS)                                            16
   {
     /*Error handling */                                            18
   }
```

allow one-sided communication, the NLAs have to be exchanged between the processes. This can be done by using the notification mechanisms of the RMA unit, or by using the TCP/IP connection.

5.6.2. Porting of an RMA environment to the GPU

Once an RMA-connection is created on the host, it can be ported to the GPU. Therefore, the requester page and the notification queue have to be mapped to the GPU address space.

On the GPU, a very simple structure is sufficient to allow the GPU to control the communication. In Listing 5.4, the according structure for an RMA port on the GPU is shown. The *queue* is the pointer to the mapped notification queue and *req_page* is the mapped requester page. The parameters *rp*, *rw_wb*, and *rp_bak* describe counter variables which are required to find the right entry in the notification queue. The *size* parameter describes the size of the notification queue. Besides this, also the *node_id* and the *vpid* of all remote endpoints are copied to the GPU.

With the structure in Listing 5.4 it is very simple to source a communication request to the RMA unit from the GPU. A GPU just has to write the descriptor in the right format to the requester page, as shown in Listing 5.5

In contrast to the Infiniband work processing, the single-threaded workload is lower. However, next to the function in Listing 5.5, also the handling of notifications has to be ported to the GPU. These functions are sufficient to allow the GPU to source and synchronize communication requests on the GPU.

Listing 5.4: RMA port for a GPU

```
typedef struct {
        volatile RMA2_Notification *queue;
        volatile uint32_t *rp;
        volatile uint64_t *rp_wb;
        uint32_t size;
        volatile uint32_t *rp_bak;
        volatile uint64_t *req_page;
} gpu_rma_port_t;
```

Listing 5.5: Creating a descriptor on the GPU and copying it to the requester page

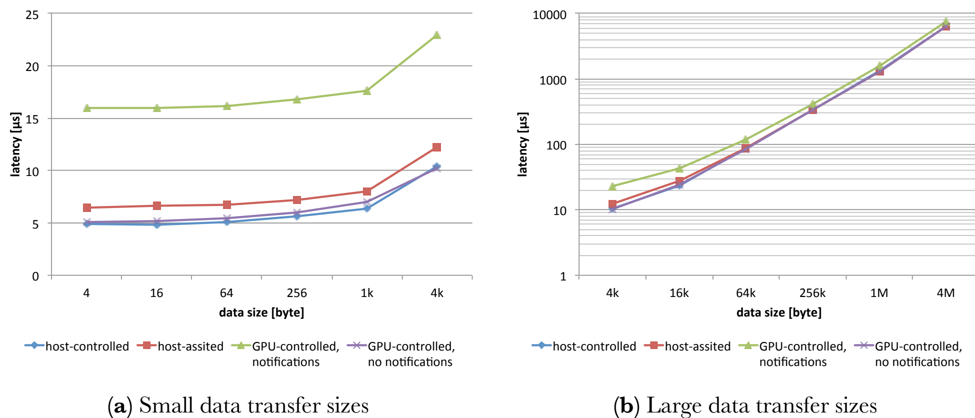
```
__device int gpu_rma2_put(···) {
 [···]
  unsigned int modifier = RMA2_CMD_DEFAULT;
  if( 0 == threadid ){
      port.req_page[0] = (((size-1) & 0x7FFFFFl) <<40l)  | \
                       ((modifier & 0xFl)      <<35l)  | \
                       ((noti     & 0x7l)      <<32l)  | \
                       ((remote_node & 0xFFFFl)<<16l)  | \
                       ((remote_vpid & 0xFFFl) <<8l )  | \
                       ((command_put & 0xFl)   <<2l )  | \
                       3l;
     port.req_page[1] = src_address + src_offset;  // src nla
     port.req_page[2] = dest_address + dest_offset; // dest nla
    }
 [···]
}
```

Location of the queue buffers

As for Infiniband, the queues for the notifications theoretically can be allocated in GPU memory instead of host memory. However, the problem is that for the RMA unit of the Extoll device these buffers are allocated in kernel space during device initialization. This is different for Infiniband where the queues are allocated dynamically during QP creation in user space. There is no way to allocate GPU memory in kernel space. Allowing a subsequent allocation of notification queues in GPU memory would require massive changes in the Extoll device drivers and therefore will be addressed in future work.

5.6.3. Micro-benchmark results for the RMA unit

We runlatency and a bandwidth benchmarks for the Extoll RMA unit. Most of the results of this chapter are already published in [80] and [120], but for the sake of completeness will listed here again. The benchmarks run on two machines with Extoll Galibier Cards, implemented on an FPGA with a core frequency of 157 MHz and a 64 bit wide data-path.



(**a**) Small data transfer sizes (**b**) Large data transfer sizes

Figure 5.17.: Latency of a GPU device-to-device transfer, inititated on host or GPU, using Extoll RMA

Latency Benchmark

To measure the latency, again a pingpong benchmark is used. For the RMA unit, we compare four different data transfer and synchronization methods:

GPU-controlled, notifications The communication is controlled by the GPU. The *completer* notification is used for remote synchronization. The GPU on the remote side polls for the *completer* notification before the data transfer is started. On the sourcing side, the *requester* notification is used for synchronization.

GPU-controlled, no notification The communication is controlled by the GPU. For remote synchronization, a GPU thread polls for the last transferred element in GPU memory before the data transfer is started. On the sourcing side, no notification is used. The ping and the pong side mutually synchronize each other. However, this method cannot be used for real-world applications, since at least local synchronization is required.

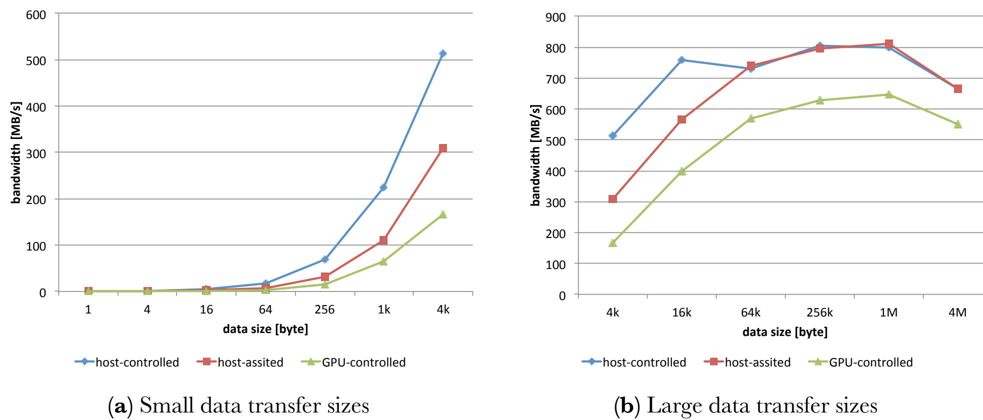
Host-assisted The GPU triggers the CPU to perform the communication.

Host-controlled The CPU controls the communication, no CUDA kernel is started.

For the host-controlled communication, we use RMA completer notifications for remote synchronization and the requester notification for local synchronization.

Again, for the GPU-controlled benchmarks, for every message size, a GPU kernel with a single block of 32 threads is started. This kernel runs the pingpong benchmark for a given number of iterations. The results are presented in Figure 5.17 and show half-round trip latency.

The results differ from the Infiniband results. If the GPU uses notifications for synchronization, the latency (15.9 μs) is three times higher than the latency of the host controlled communication (4.9 μs). Still, when a GPU thread polls on the last transferred



(**a**) Small data transfer sizes (**b**) Large data transfer sizes

Figure 5.18.: Bandwidth of a GPU device to device, using GASPI running on host and GPU

element and the communication is handled without notifications, the latency is almost as small as the latency of a CPU- initiated communication (5.1 μs). It is also better than the host-assisted version (6.4 μs).

Polling for notifications causes many additional accesses through the PCIe-bus, since the notifications are located in host memory. This may slow down the performance. If notifications are used, it is also necessary to free them after consumption to avoid an overflow of the notification queue. Freeing the notification adds some more single-threaded work to the GPU, which is not required if the thread polls on the received data. This will be analyzed later, in more detail, with the help of the performance counters.

Bandwidth

For the bandwidth measurements with the RMA unit, no remote synchronization is required. Therefore, now three different methods are used:

GPU-controlled The communication is controlled by the GPU, using the *put* operation with the *requester* notification for local synchronization.

Host-assisted The GPU triggers the CPU to transfer the data.

Host-controlled The communication is controlled by the host, using the *put* operation, with the *requester* notification for local synchronization

The results are pictured in Figure 5.18 and show a large gap between GPU-controlled communication host-controlled communication, even for larger data transfer sizes. This differs from the results of the latency benchmark where GPU controlled communication almost reaches the same latency as-host controlled communication, if no notifications are used. However, although no remote notifications are used, this benchmark requires *requester notifications* for local synchronization. This gap also stays for larger message sizes.

That is in contrast to the results for Infiniband where the gap between the peak bandwidths ofGPU-controlled and host-controlled communication is much smaller. The reason for this may be the polling on the host-notification buffer which results in a lot of additional accesses through the PCIe-bus and, therefore, slows down the bandwidth.

However, this is also true for Infiniband if the completion queue is located in host memory, because then the GPU also has to poll on host memory for local completion. But for Infiniband even then the gap between the maximal bandwidth of host-controlled communication and GPU-controlled communication is much smaller.

Using Infiniband, multiple communication requests can be added to the send queue before the GPU has to poll on the completion queue for local synchronization. Using the RMA unit, the requests are directly submitted to the hardware, so this is not possible. The bandwidth benchmark submits a put request to the hardware and then directly starts polling on the requester notification before the next communication request is submitted. On the host, a queue is used to handle failed communication requests . However, on the GPU, this would add a lot of overhead to the communication and therefore is not implemented [80].

5.6.4. Performance counter analysis for the RMA

This section is dedicated to the analysis of the previously discussesed results, which was already published in [120]. For this analysis, again the GPU performance counter are used. Therefore, the pingpong benchmark with 100 iterations and a payload size of 1 kb was started and accesses to host and device memory and the number of instructions were counted.

Table 5.5 shows the performance counters for the pingpong benchmark with 100 iterations. Compared to the Infiniband results, the number of instructions is much lower, especially if no notifications are used. Only around 46 instructions are required per iteration if notifications are used and only around 22 instructions if no notifications are used. On Infiniband, however, 460 instructions were required to create a communication request. Sourcing a communication request to the RMA unit is much simpler than using Infiniband verbs. However, using notifications results in double the number of instructions. This is caused by the handling of the notifications, since notifications have to be consumed and later freed.

The other reason for the increased latency is the number of accesses to the host memory, especially read accesses. If the notifications are used, over 4,000 read accesses to host memory are required, which means that around 40 accesses are required per iteration. However, this benchmark also includes the repeated polling on the notifications in host memory. Further accesses are required to handle and evaluate the notifications after successful polling.

If notifications are used, further write accesses to host memory are required. The read counter for the notifications, which is located in host memory, has to be incremented and the notification has to be freed, what means that it is set to zero. This has to be done for the local *requester* notification and for the *completer* notification, which results in many additional accesses to system memory through the PCIe-bus.

Summarized, the same conclusions can be drawn as for Infiniband: There are two main factors which slow down the performance of GPU -controlled put/get communication: The number of single-threaded instructions and and the number of accesses to system memory which are required to source or synchronize a communication request.

5.7. One-sided Communication Interface on the GPU

So far, only micro-benchmarks were used on the GPU. These benchmarks start with a minimum number of threads and do not perform any computations. The message rate benchmark is the only benchmark that is started with more than one thread block but still so few that all thread blocks can be scheduled in parallel.

However, although GPU-controlled communication did not provide any performance benefits so far, this may be different for an application level benchmark. For the hybrid methods, so far, no GPU kernel was started, which means also no context switches between host and GPU were required.

The micro-benchmarks also require no synchronization between the threads and blocks on the GPU. Besides, the micro-benchmarks are only started with so few threads that all threads are scheduled simultaneously on the GPU. Looking at the application level, this is seldom the case. Allowing put/get communication directly on the GPU causes several problems related to the GPU programming model. Therefore, before looking at an application level benchmark, first some observations on the communication library running on the GPU are required.

5.7.1. Communication endpoints

Communication is performed between *endpoints*, regardless of looking at one-sided or at two-sided communication. In communication libraries like MPI, GPI, or openShmem, these endpoints are identified by processes. Every process has a unique ID. However, the concept of a process does not exist on a GPU.

For DCGN [122] the concept of *slots* was introduced. Slots allow a variable number of connections between GPUs for communication. Every GPU has at least one slot, the maximal number of slots is equal to the number of threads that can run concurrently on the GPU. The results show that using too many slots creates a lot of overhead.

Table 5.5.: Comparison of different synchronization mechanism for the pingpong benchmark, using GPU Extoll RMA API

	notifications	**poll device memory**
host memory read	4,368	0
host memory write	2,908	303
device memory accesses (r/w)	6,788	1714
instruction	46,413	22,491

Using a specific communication slot for every thread or block restricts the usability of the GPU, since it allows only starting a kernel with as many threads as can be executed concurrently on the GPU. Thread-blocks are non-preemptive, so once a thread-block is scheduled, it occupies a GPU until it is completed. It is not predictable when a specific thread-block is scheduled. Using this in conjunction with message passing could easily result in a deadlock if one block or thread waits for a specific thread that is not scheduled.

This describes the main problem with message passing and synchronization on GPUs. We will return to this problem later, when we describe the implementation of our benchmarks in more detail. However, to avoid race conditions and to allow the scheduling of as many threads as supported by the GPU, the completion of a communication request or the synchronization with a remote GPU should not depend on the scheduling of a specific block or thread.

Furthermore, all threads and blocks on a GPU share the same virtual GPU address space. Changes in the device memory are visible to all threads on the GPU and communication resources are visible to all threads.

Because of this we use a GPU-centric communication library. Every GPU in the system represents an endpoint. Communication is handled between the GPUs and every GPU gets a unique ID.

5.7.2. Block-save communication

To avoid race conditions, the communication libraries must provide a kind of thread safety. However, instead of creating a thread-save communication library on the GPU, we decide to create a *block-save* communication library. A block corresponds to a virtualized *shared multiprocessor* on the GPU. Blocks are independent of each other and therefore can execute different tasks while between the threads of one block, only data parallelism is supported.

The threads of one block should call a communication request in parallel. The communication library handles the synchronization between these threads. However, if shared memory is used to optimize the communication, this is required anyway. Multiple parallel threads are required to copy the request from shared memory to the queue. To synchronize the blocks and to provide a block-save communication library, we used a mutex-variable and GPU-atomic operations, as described, for example, in [136].

Newer GPUs also allow concurrent kernel executions on the same GPU, if the kernels are submitted to different streams. If these kernels are launched from the same host process, they share one virtual GPU address space. The resources, which are required for the communication, are also located in this virtual GPU address space. The same applies to block-synchronization variables like mutexes. Therefore, concurrent kernels also share the same communication infrastructure and, therefore, represent the same communication endpoint.

5.7.3. Asynchronous one-sided communication

One-sided communication functions on the GPU should be asynchronous and non-blocking to allow overlapping of communication and computation. A thread block initiates the communication and directly returns.

Therefore, a *wait* function is required to synchronize the communication locally. It is not required that the same block that initiates a data transfer also synchronizes it, which allows a more flexible usage.

For Infiniband, this functions polls on the completion. For the RMA, notifications are used. For put operations, the *requester notification* is used for local synchronization, for get operations, it is the *completer notification*.

5.7.4. Queues

The idea of communication queues, similar to the queues in GAPSI, is very suitable for GPU-controlled communication. Queues allow multiple blocks to source communication requests independently and simultaneously, without binding a block to a specific communication slot. The previous sections have shown that using multiple connections can help to achieve higher message rates on the GPU.

Hence, at first sight, it seems to be reasonable to create one queue for every thread-block on the GPU, so every thread-block can use its own queue for communication. For Infiniband, this implies that for every thread-block, one QP connection to every remote GPU is required. For the RMA-unit, for every queue, another RMA port can be used. However, on a modern GPU, a kernel with more than one million blocks can be launched. Although not all applications start kernels with many blocks in parallel, often kernels with more than thousand blocks are started. To create one queue for every thread-block would exhaust the communication resources. Especially for the Infiniband queue buffers, the memory footprint is very large and easily exceeds the amount of available memory.

Therefore, between 2 and 32 communication queues should be used. This saves the communication resources and provides flexibility between the thread-blocks.

5.7.5. Remote Synchronization

To allow synchronization between the active and the target side of one-sided communication, we also implemented a weak synchronization (remote notification) mechanism on the GPU. For Infiniband, theoretically two possible methods for weak synchronization exist. Either a flag mechanism as in GPI-2 or remote write requests with immediate data can be used. In the first case, an additional area of flags is allocated in GPU device memory and registered for the network device. Then, the flags are updated with remote write operations to this area. However, in this case, `gaspi_write_notify`internally requires a double creation of work requests, one for the notifications and one for the actual data transfer.

By using the immediate value instead, this additional work request creation can be avoided. Instead the flag is directly transferred along with the remote write operation.

However, this method has also some disadvantages. On the remote side, a *receive* request is required to accept the immediate data. If no receive request is posted before a remote write request with immediate data arrives, an error occurs. This would also require an evaluation of the completion element on the target side, which adds a lot of additional overhead to the communication as shown in the previous section. Another problem is that the target side cannot simply ignore the notification. If the notification is not consumed, an overflow may occur on the completion queue. Also, it is not possible to just send a notification without data transfer since a zero-data transfer is not supported in Infiniband. Therefore, we decided to use a flag mechanism on the GPU.

For the RMA unit, also the flag mechanism can be used. Another possibility is the use of notifications. For put operations, the completer notification can be used. The RMA unit allows the creation of a remote notification without transferring data. In contrast to Infiniband, it is not required to post a receive request on the target side to accept a notification. However, there are still two disadvantages to this method. First, notifications have to be consumed to avoid an overflow of the notification queue. The second disadvantage is that the queues are located in host memory, so polling on the completion of a remote notification results in polling on host memory. Beside this, notifications for remote and local synchronization are submitted to the same notification queue if the same port is used. Therefore, additional overhead is added to the GPU to handle these notifications. In the previous section we showed that this overhead slows down the performance, so here also a flag mechanism is preferable.

5.8. Inter-block synchronization

The above described communication interface can now be used for applications, where the GPU controls the communication. However, the main problem of GPU-controlled one-sided communication is the intra-GPU synchronization between the thread-blocks. This problem is best illustrated by describing the implementation of a simple benchmark. As in the previous chapter, we use the *Himeno* benchmark.

The himeno benchmark, a stencil code, provides a very simple communication pattern. Every endpoint only communicates with its direct neighbors.

In contrast to the previous chapter, now changes in the GPU kernel code are required, as the communication is controlled by the GPU. Therefore, in the following, first the implementation of the GPU compute kernel is addressed in more detail.

Every iteration all points of the grid are updated using the neighboring points. On a single GPU, the 3-D domain is processed by two-dimensional thread-blocks, as shown in Figure 5.20. These thread-blocka processe the domain from the bottom to the top of the 3-D grid, as shown in Figure 5.19.

Using multiple GPUs, in the simplest case, the complete domain is sliced along the z-direction and then distributed between the GPUs. In this case, the boundary values, which have to be exchanged, are on the top and bottom of the grid, as shown in Figure 5.19 in green. That implies that all thread-blocks update a part of these boundaries – and that all thread- blocks require data from a remote GPU. Furthermore, all thread-

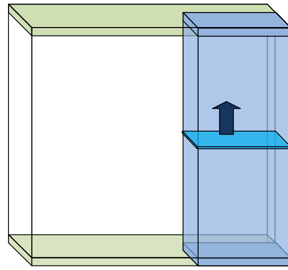


Figure 5.19.: 3-D stencil, thread-block process columns of the domain, boundaries are on top and bottom

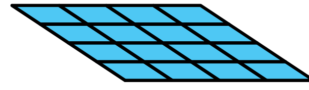


Figure 5.20.: Structure of two dimensional thread-block

blocks also require points that are processed by the surrounding blocks, since not only values on the top and bottom are required, but also from right, left, front, and back. Therefore, inter-block synchronization is required. This is the crux of the matter of GPU-controlled communication.

Normally, this inter-block synchronization is provided by starting a new kernel for every iteration and adding this kernels to the same stream. A thread-block is nonpreemptive, so once a block is scheduled, it uses GPU resources until it is completed. Staring a kernel with more blocks than can run concurrently on the GPU can result in a deadlock if another method of inter-block synchronization is used. In the following, three possible solutions for this problem are presented.

5.8.1. In-kernel synchronization and communication

In the first approach, a communication request is directly sourced in the compute kernel. Also, the synchronization with a remote GPU is handled in the compute kernel. The control flow of such an application is shown in Figure 5.21. Using this approach, inter-block synchronization within the compute kernel is required. That, however, involves the risk of deadlocks.

One possibility to avoid dead-locks is to start only as many threads on the GPU as can be scheduled concurrently. This technique is also called *persistent threads*. For intra-GPU synchronization, then a barrier as described in [137] can be used. All blocks on the GPU first synchronize using an inter-block barrier. After this, one block can start the data transfer of the complete boundary before the next iteration is started. However, using persistent threads either restricts the problem size to a minimum or requires several non-trivial changes in the code, since normally the number of used threads is determined by the problem size (one thread for every point along the x/y domain). This is not only true for the Himeno benchmark but for most GPU-related applications. Furthermore, it was shown in [138] that the use of persistent threads on GPUs often results in performance losses. Therefore, this is not a suitable solution.

Another possibility is to organize the communication in a way that every block is responsible for its own part of the boundary. For every iteration, a new kernel is started.

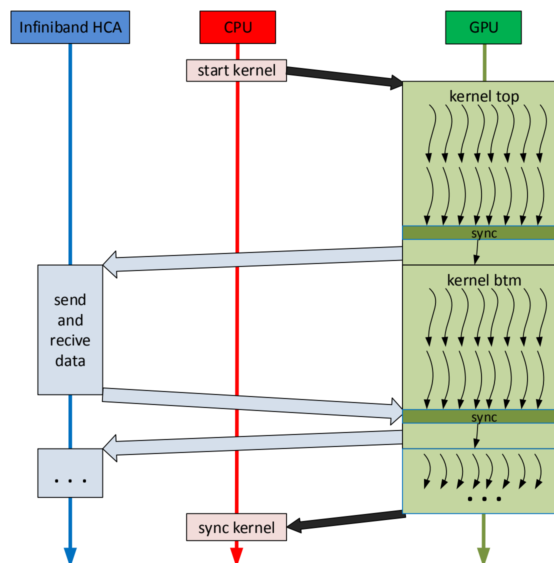


Figure 5.21.: Control flow of an application with communication control on the GPU, inter-block Synchronization within a CUDA kernel

At the beginning of a new iteration, every block checks, if its part of the remote boundary is updated by checking for a notification flag. At the end, every block starts the transfer of his part of the boundary to the remote GPU. Since for every iteration a new kernel is started, a live lock can be avoided. If two kernels are used, one for the top part of the grid and one for the bottom part, as shown in Figure 5.21, data transfer and computation can be overlapped.

The CPU is required to submit the kernels to the stream. For every iteration, at least one new kernel is required. After this, however, the CPU is not needed anymore and can be set to sleep or used for other work while the GPU subsequently executes the kernels since kernel execution is asynchronous with respect to the host CPU.

However, this solution results in many small data transfers. Due to the two- dimensional structure of the blocks, the data of one block are also non-continuous in memory. That results in more data transfers or in additional work due to data packing and unpacking. Therefore, a solution is required that allows the starting of as many threads as required but still provides intra-GPU synchronizations.

Our solution uses an atomic counter for this. Again, for every iteration, a new kernel is started to guarantee inter-block synchronization. To overlap communication and computation, it is also possible to use more than one kernel, for example, one for the top and one for the bottom part of the grid, as shown in Figure 5.21. The kernels are started with as many thread-blocks as are required to process the grid.

Once a new kernel is started, the GPU schedules the thread-blocks of this kernel until all blocks are completed. Only then a new kernel in the same stream is started. When the last block has finished the computation, all points in the grid are updated and the

Listing 5.6: Stencil code with communication in compute kernel

```
__global__ void kernel_iter_btm( float* in,  ···, int iter){
  /*wait for remote boundaries*/                                2
  if(iter>0) {
    wait_for_notification();                                    4
  }
  for( i= 0; i<Z_MID, i++){                                     6
    [···] /*calc points along column*/
  }                                                             8
   /* find last block*/
  unsigned int old;                                             10
  if (threadIdx.x==0&& threadIdx.y==0)
  old=atomicAdd(&send_counter, 1);                              12
  if(old == blocks -1) {/*last block starts data transfer*/
    if(threadIdx.x==0 && threadIdx.y==0)                        14
      atomicExch(&send_counter, 0); /*reset counter*/
       /*write borders to remote GPU*/                          16
     gpu_write(source_address, remoteGPU,
               destination_address,size,queue)                  18
    /*send notification and reset own*/
    notify_gpu(remoteGPU);                                      20
    notify_reset();
   }                                                            22
}
```

boundary can be transferred. Therefore, this last block should start the data transfer. Since it is not predictable which block is scheduled when, we use an atomic counter to determine the last thread-block.

At the end of each iteration, every block increases this counter by one. The last arriving block resets the counter and starts the data transfer, as shown in lines 10 to 22 of Listing 5.6.

To synchronize the GPUs, a flag-based notification mechanism is used. Immediately after a remote write operation, which transfers the boundaries, a notification is written to the remote GPU to announce the new data. At the beginning of every iteration, all blocks have to wait for this remote notification before the computation is continued. All threads can poll in parallel the notification flag, as shown in lines 3 to 5 of Listing 5.6. The last block resets the notification for the next iteration before starting a new data transfer. We use two notifications, one for the odd and one for the even iterations.

5.8.2. Stream Synchronization

The second approach usesstream synchronization to synchronize communication and computation on the GPU. Figure 5.22 shows the control flow for a stencil benchmark using this approach.

The communication functions *send_top/btm* are added to a stream as kernels, so now these functions are implemented as *cuda kernels* and not as GPU device functions, see Listing 5.7 for the send function. The same applies to the synchronization kernel *wait_top/btm*.

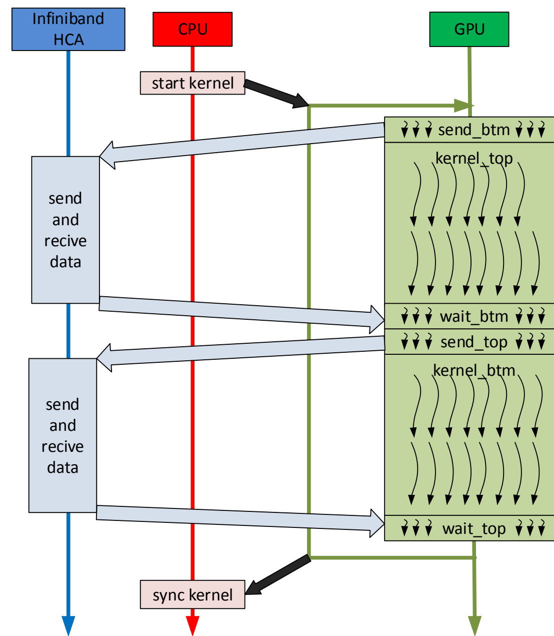


Figure 5.22.: Control flow of an application with communication control on the GPU, using stream synchronization

Listing 5.7: Communication kernel

```
__global__ void send_top(int remote, ...){
 gpu_write(source_address, remoteGPU,
               destination_address, size, queue)
 notify_gpu(remoteGPU);
}
```

To overlap communication and computation, two pure computation kernels *kernel_top* and *kernel_btm* are used. For every iteration, first a communication kernel (*send_btm*) is added to the stream. This kernel starts the data transfer of the bottom boundary which is then handled by the NIC. Next, the compute kernel for the top part of the grid is added to the stream (*kernel_top*). The execution of this kernel is overlapped with the data transfer from the bottom boundary. Next, a kernel that waits until the bottom boundaries are updated by a remote GPU is added to the stream. This kernel polls for an update of a notification flag. The same procedure is now repeated with the data transfer from the top boundary (*send_top* and *wait_top*) and the computation of the bottom part of the grid (*kernel_btm*). Since the order of the kernels in one stream is guaranteed, the data transfer is not started until the local boundary is updated and a new kernel is not started until the remote boundary is updated.

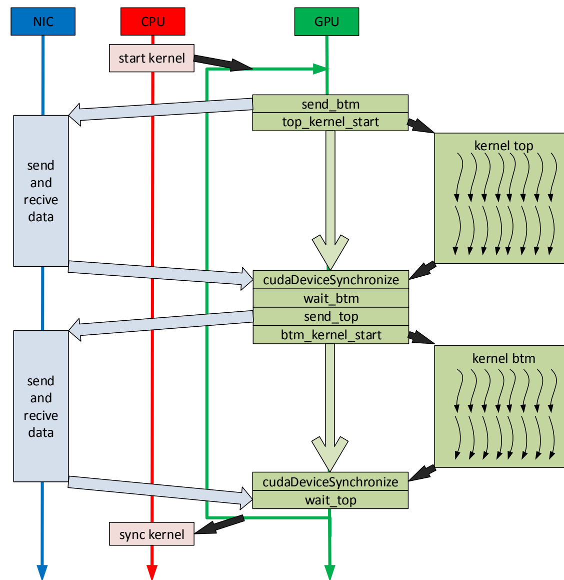


Figure 5.23.: Control flow of an application with communication control on the GPU, using cudaDynamic parallelism and device synchronization

5.8.3. Synchronization with dynamic parallelism

The last method described uses *dynamic parallelism*, which is usable for newer GPUs with Cuda compute capability 5 and higher. *Dynamic parallelism* allows starting and synchronization of compute kernels directly on the GPU. Therefore, dynamic parallelism seems to be a good method to support communication libraries which are directly running on the GPU.

In principle, the approach for hybrid applications, in which the host controls the communication can be transferred to the GPU. The host CPU only starts a single kernel with one thread-block, the *control kernel*. This control kernel starts the communication and the compute kernels on the GPU. However, the hybrid approach described in the previous chapter cannot directly be transferred to the GPU since most kernel and stream synchronization functions are not supported on the GPU. Neither *cudaEventSynchronoize* nor *cudaStreamSynchronized* are supported using dynamic parallelism, thus other synchronization methods must be used.

Our approach uses *cudaDeviceSynchronize* to synchronize computation kernels and communication functions. The workflow of this approach is shown in Figure 5.23. Again, two pure compute kernels are used. In contrast to the previous approach, the communication functions are not implemented as GPU kernels but as device functions which are called by the control kernel.

The CPU starts a kernel with a single block. For every iteration, this block first starts the data transfer of the top boundary by submitting a work request to the network hardware (*send_btm*). Then, the compute kernel for the top part of the grid is started (*kernel_Top*)

and overlapped with the data transfer, which is handled by the NIC. The control kernel blocks until the compute kernel is completed by using *cudaDeviceSynchronize*. If the compute kernel is completed, the control blocks waits for a notification of a remote GPU to ensure, that the bottom boundary is updated (*wait_btm*). Once the boundary is updated, the GPU starts the data transfer of the top boundary (*send_top*) and the compute kernel for the bottom part of the grid (*kernel_btm*). Then, the data transfer of the top boundary is overlapped with the computation of the bottom part of the grid. Again, the control block synchronizes the compute kernel by using *cudaDeviceSynchronize* and then waits for a notification of the remote GPU to ensure that the top boundaries are updated (*wait_top*) before a new iteration is started.

Dynamic parallelism and stream/in-kernel synchronization

It is also possible to combine the stream synchronization and the in-kernel synchronization with dynamic parallelism. Then, the CPU also starts a single kernel with a single thread-block. This kernel starts the computation and, if necessary, the communication kernels for every iteration.

This may be useful, as one of the main goals of this work is to relieve the CPU of additional work. Therefore, we run a small benchmark on a single GPU using the Himeno benchmark. In the single GPU version, no communication is required. However, we still use two compute kernels, one for the top part of the grid and one for the bottom part.

The first version of this benchmark does not use dynamic parallelism. A host thread adds all compute kernels for every iteration to the same stream. Then, this stream is synchronized with *cudaStreamSycnhronize*.

The version using dynamic parallelism starts a single kernel on the GPU, the *master kernel*. This kernel starts the actual compute kernels for every iteration. On the host, the *master kernel* is synchronized with *cudaStreamSynchronize*.

We measure the complete runtime of all kernels on the host and thereby the time from issuing the kernel(s) from host until *cudaStreamSynchronize* completes. In addition, we measure the time the host thread spends on adding the kernel(s) to the stream, reported as *CPU overhead*. In this time, the CPU cannot be used for other work, while for

$$t_{idle} = t_{execution} - t_{overhead}$$

the CPU can enter a sleep state or can be used for other tasks.

We run the himeno benchmark with a size of $512 \times 512 \times 256$ on a single GPU and vary the number of iterations. The results are shown in Figure 5.24. The execution times differ so little that the graphs are hardly distinguishable. For a smaller number of iterations, the CPU overhead is also negligible for both cases. However, for more than 500 iterations, the overhead rises linearly without dynamic parallelism. If dynamic parallelism is used, the overhead is still barley noticeable. The reason for this is probably the queue size of a CUDA stream. The CPU overhead rises suddenly for more than 500 iterations, while two kernels are added to the stream for every iteration. Therefore, it is obvious that a stream queue has space enough for approximately 1,000 entries.

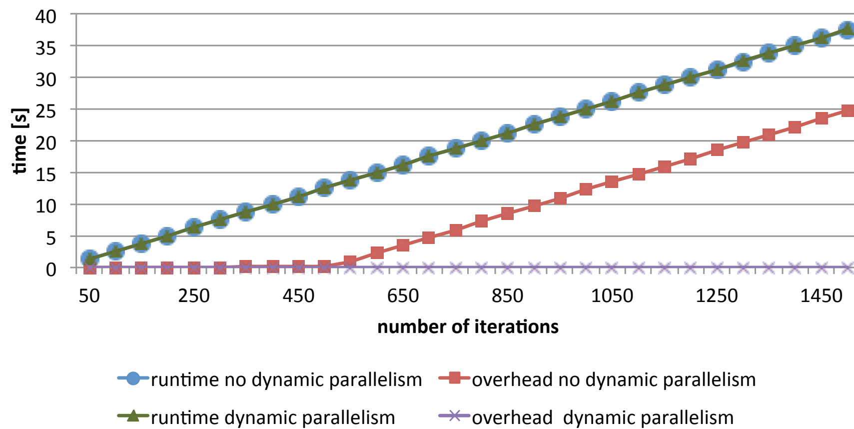


Figure 5.24.: CPU overhead with and without parallelism dynamic parallelism

Using stream synchronization for communication, for every iteration, four kernels are required, using in-kernel synchronization instead, two kernels are required. Since we normally run the Himeno benchmark with 1,000 iterations, it is useful to use dynamic parallelism to keep the overhead on the CPU as small as possible – especially since there is hardly any impact on the performance.

Using dynamic parallelism does not affect the performance but the energy efficiency. For that, the power consumption of CPU, DRAM, and GPU is measured while the benchmark is running with different number of iterations. With this power consumption, the energy consumption of the complete benchmark and the performance per watt, expressed in MFlop/second/watt (or Mflop/joul) is determined. For the method of power measurement, refer to Appendix A on page 175. The results are shown in Figure 5.25.

To synchronize the kernels, the synchronization shown in Listing 5.8 is used. This synchronization method allows a decreased polling rate so that the CPU can enter sleep states.

Since the runtime of the iterated kernels is large enough, there are no performance penalties. The results in Figure 5.25 show that with dynamic parallelism the energy efficiency of this benchmark is always better.

The reason for the increased energy efficiency becomes clear by looking at the power consumption over time for the different approaches, shown in Figure 5.26 for the benchmark with 500 iterations.

The power consumption of the GPU does not differ for both cases. This is different for the power consumption of CPU and DRAM. Although the CPU is set to a sleep state

Listing 5.8: Set CPU to sleep while waiting for completion

```
while( cudaStreamQuery(stream) == cudaErrorNotReady)
       usleep(5000);                                    2
```

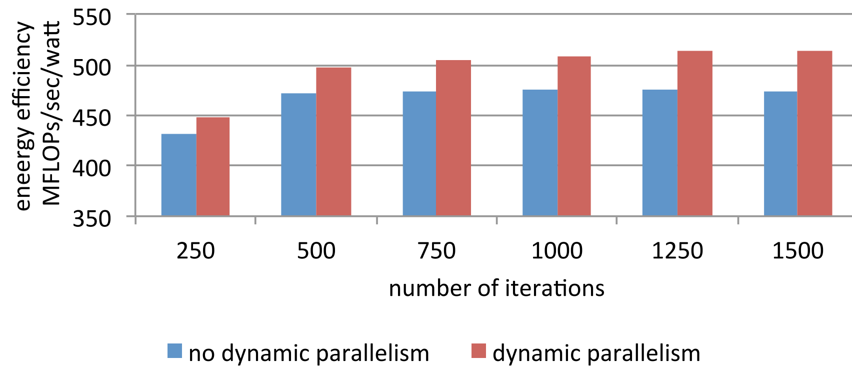
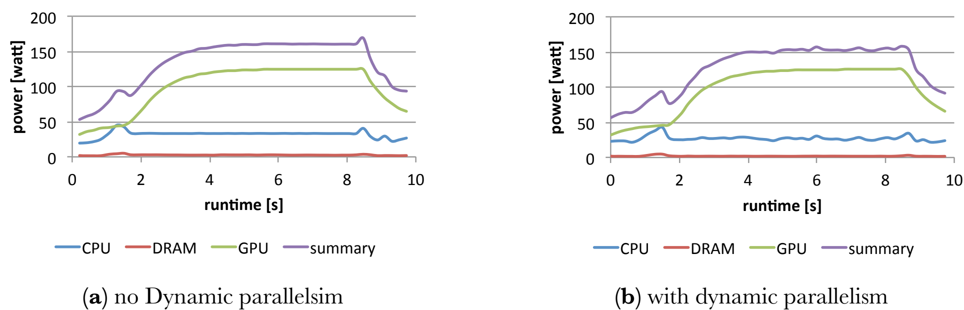


Figure 5.25.: Energy efficiency for a single GPU for different numbers of iterations, using host kernels and dynamic parallelism



(**a**) no Dynamic parallelsim (**b**) with dynamic parallelism

Figure 5.26.: Power over time for the Himeno Benchmark of a size $512 \times 512 \times 156$ with 250 iterations, with and without dynamic parallelism

in both cases after issuing the kernels, the power consumption of the CPU is significantly lower (between 22-24 watt) if dynamic parallelism is used than if not (ca. 32 Watt).

The power consumption of the DRAM memory also differa, although it can hardly be recognized in Figure 5.26. If dynamic parallelism is used, the DRAM memory consumes approximatly 2.2 watt with dynamic parallelism and 3.3 watt without dynamic parallelism. Therefore, the summarized power consumption with dynamic parallelism is significantly smaller. Summarized, using dynamic parallelism seems to be a good way to relive the CPU for GPU-centric applications and thereby save energy.

5.8.4. Himeno performance results

In this section, the performance results of the previously described approaches for intra-GPU synchronization for GPU-controlled communication are discussed and analyzed. The results are compared with the hybrid implementation of the benchmark described in section 4.5 where the CPU controls the communication, using GASPI as a communication layer.

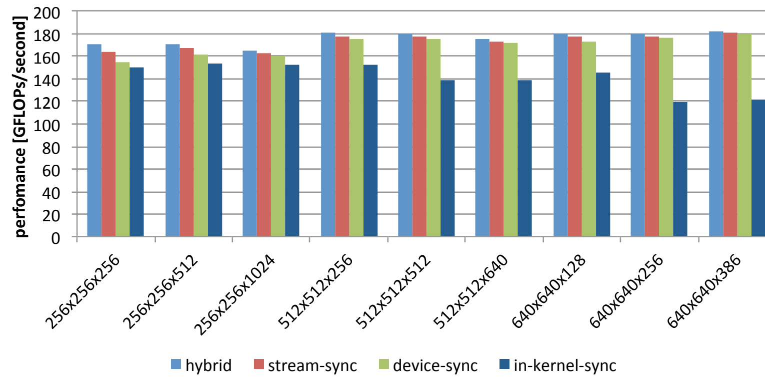


Figure 5.27.: Performance of the Himeno benchmark for different problem sizes and synchronization/ communication methods

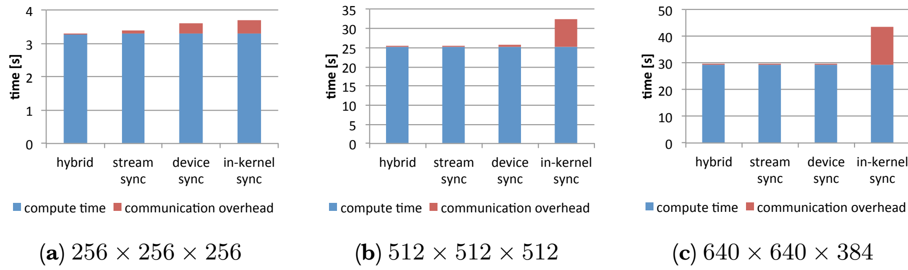
The test system consists of two nodes with two Intel 4-core Xeon E5-2609 CPUs, connected with a Mellanox ConnectX-3 FDR Infiniband. Each node is equipped with a single Nvidia K20c GPU.

We run the Himeno benchmark for different input sizes. Table 5.6 summarizes the properties for the different sizes. Since we are only able to run the benchmark on two GPUs, these sizes are selected in such a way that each of the three problem sizes requires the same amount of data transfer while the number of Floating Point Operations (FLOP) differ. By this, the ratio between communication and computation is varied.

Figure 5.27 shows the performance results for the different benchmark sizes of the Himeno benchmark. The hybrid version, using the CPU to control the communication, is always performing best, especially for small problem sizes. However, this corresponds to the results in the previous sections. As shown in the previous chapters, this kind of workload allows to exploit overlapping the communication overhead and data transfer,

Table 5.6.: Properties of the Himeno benchmark for different problem sizes and execution on two GPUs

Problem size	required memory per GPU (Byte)	FLOPs per GPU per iteration	number of blocks	boundary size (Byte)
$256 \times 256 \times 256$	$482\ M$	$280\ M$	4×64	$256\ k$
$256 \times 256 \times 256$	$954\ M$	$561\ M$	4×64	$256\ k$
$256 \times 256 \times 1024$	$1.90\ G$	$1.78\ G$	4×64	$256\ k$
$512 \times 512 \times 256$	$1.93\ G$	$1.13\ G$	8×128	$1\ M$
$512 \times 512 \times 512$	$3.81\ G$	$2.26\ G$	8×128	$1\ M$
$512 \times 512 \times 640$	$4.75\ G$	$2.83\ G$	8×128	$1\ M$
$640 \times 640 \times 128$	$1.54\ G$	$0.89\ G$	10×160	$1.5\ M$
$640 \times 640 \times 256$	$3.01\ G$	$1.77\ G$	10×160	$1.5\ M$
$640 \times 640 \times 384$	$4.48\ G$	$2.56\ G$	10×160	$1.5\ M$



(**a**) $256 \times 256 \times 256$ (**b**) $512 \times 512 \times 512$ (**c**) $640 \times 640 \times 384$

Figure 5.28.: Runtime and communication overhead for the different problem sizes of the Himeno benchmark

which is clearly done for host-controlled communication. For GPU-controlled communication, such an exploitation is not possible. This results in an increased execution time. Here, namely the data transfer can be overlapped but not the work request generation and the completion handling. Furthermore, the communication overhead on the CPU is negligible small compared to the overhead on the GPU, as we have shown in the previous section.

However, for larger problem sizes, the performance of the GPU-controlled communication gets all the more closer to the performance of CPU-controlled communication if stream or device synchronization is used. Only the version using in-kernel synchronization is performing significantly worse for all problem sizes. The stream-synchronization is performing best for all problem sizes using GPU-controlled communication. For a problem size of $256 \times 256 \times 256$ grid points, the GPU-controlled version reaches almost 96% of the performance of hybrid version while, for a problem size of $640 \times 640 \times 386$, over 99% of the performance of the hybrid version are reached. The version using in-kernel synchronization is performing worse for larger problem sizes. While for a benchmark size of $256 \times 256 \times 256$, approximately 88% of the performance of the hybrid version are reached, for a benchmark size of $640 \times 640 \times 386$, it only reaches around 67% of the performance.

The reason for this becomes clear by looking at the communication overhead of different configurations and synchronization methods. Figure 5.28 shows the execution time of the benchmark for three problem sizes. The complete execution time is divided into computation time and communication overhead. For this, the application is executed once with communication and once without. The communication overhead can then be determined with

$$t_{overhead} = t_{with_communication} - t_{without_communication}$$

For the hybrid approach, the overhead is very small for all problem sizes and is actually in the area of measurement uncertainty. The data transfer can completely be overlapped with the computation resulting in the best performance.

For GPU-controlled communication using stream-synchronization or device synchronization, the communication overhead for the small problem size is large in relation to the computation time. Therefore, the communication overhead has a large influence on

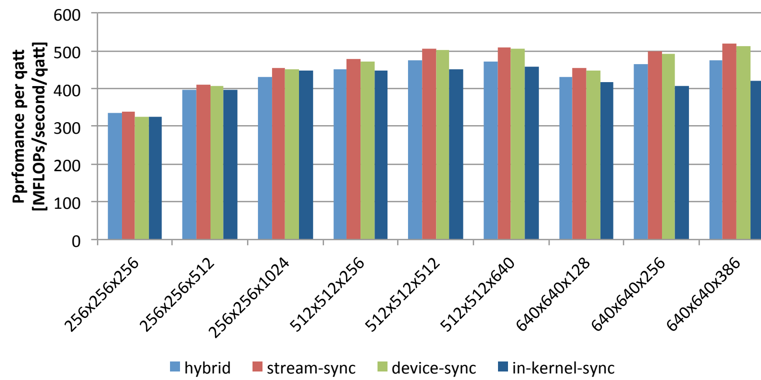


Figure 5.29.: Energy efficiency of the Himeno benchmark for different problem sizes and communication methods

the performance. The overhead caused by the stream synchronization is the smallest. However, for larger problem sizes the communication overhead stays constant, while the computational work rises, making the overhead less relevant for the performance of the complete application.

This is different for the in-kernel synchronization. Here, the communication overhead increases for larger problem sizes. We assume two reasons for this: For the larger problem size, more thread-blocks are started, resulting in an increasing overhead for inter-block synchronization. Furthermore, if more thread-blocks are started, a thread block advancing slower (e.g., the thread block that starts the communication) can slow down all the other thread-blocks, so the effect of in-kernel synchronization is strengthened.

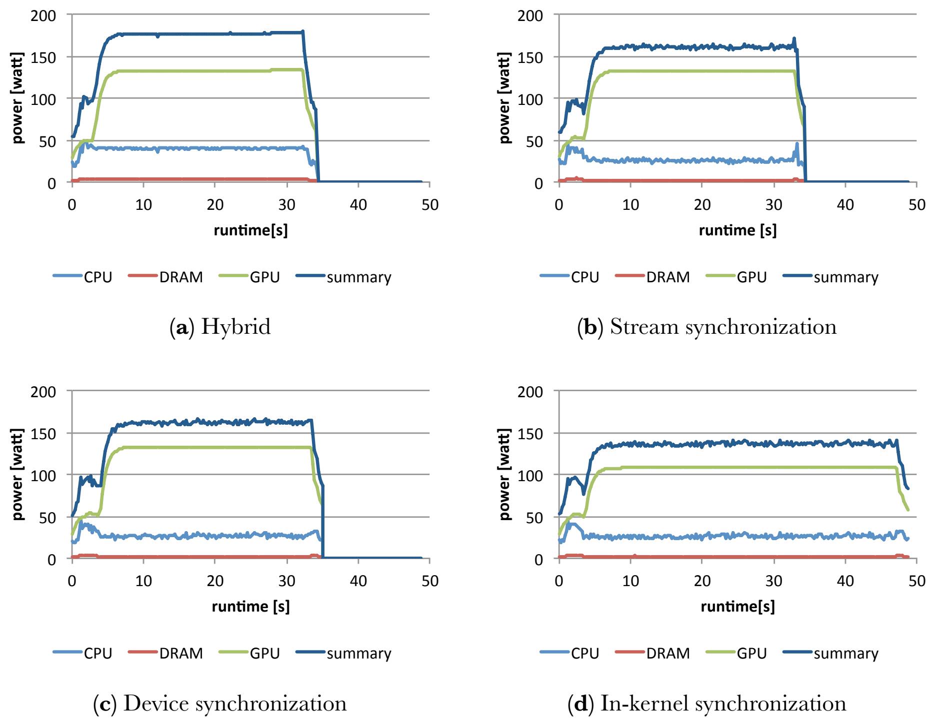
5.8.5. Energy efficiency

The results in the previous section show that there are no performance benefits to using GPU controlled communication compared to the hybrid approach. This section provides a closer look at the energy efficiency of the different communication and synchronization methods.

To allow the GPU to enter a sleep state, an increased polling period is used, as illustrated in Listing 5.8 (p. 130. For the hybrid application, an increased polling is used for the kernel synchronization and for the notifications flags for weak synchronization, as shown in Listing 5.9. If `GASPI_TEST` is used as timeout parameter, the function just checks once for a new notification and then directly returns. However, to avoid performance losses, for the hybrid version, the sleep period can set to maximal 1,000 μs.

Listing 5.9: Increased polling for a notification on the host using the GASPI interface

```
while(gaspi_notify_waitsome(···,GASPI_TEST)==GASPI_TIMEOUT)
                usleep(sleep_period);                                2
```



(**a**) Hybrid (**b**) Stream synchronization

(**c**) Device synchronization (**d**) In-kernel synchronization

Figure 5.30.: Power over time for the Himeno benchmark using a problem size of $640 \times 640 \times 384$ and different communication methods

Using GPU-controlled communication, the CPU is set to sleep right after starting the master kernel. Since the runtime of this master kernel is much longer, here an increased polling with a sleep period of 5000 μs can be used without any performance losses.

The benchmarks run with 1,000 iterations and the power consumption of GPU, CPU and DRAM is measured. Figure 5.29 shows the energy efficiency of the different communication methods, expressed in MFLOP/s/watt (or $MFLOP/Joule$). GPU-controlled communication with stream synchronization is the most energy-efficient method for all problem sizes, although the performance for small problem sizes is recognizably worse than the performance of the hybrid version.

For larger problem sizes, the energy efficiency for GPU-controlled communication is significantly better than the hybrid approach if stream or device synchronization is used. For instance, for a problem size of $640 \times 640 \times 386$, the energy efficiency of the *stream synchronization* is about 10% better than that of the hybrid version. The main reason for this is the increasing number of floating point operations which are required for a single iteration, i.e., the relation between communication requests and computation.

Figure 5.30 shows the power consumption of the different components over time for a benchmark size of $640 \times 640 \times 386$. The figures illustrate the use of power on a single machine.

While the power consumption of the GPU is identical for the hybrid, stream- and device-synchronization methods, it is considerably lower for the in-kernel synchronization. The reason for this is probably that the in-kernel synchronization slows down the complete GPU. Therefore, not all multiprocessors can be completely utilized. This leads to a considerably smaller energy consumption of the GPU. However, the increased execution time outweighs this benefit for almost all problem sizes.

For all methods using GPU-controlled communication, the power consumption of the CPU is considerably lower than for the hybrid method. Although the communication is handled by the network device, the CPU cannot enter a sleep state for a longer period and therefore it consumes about 50 watts while the benchmark is running on the GPU. If the GPU controls the communication, the CPU can be set to sleep for the complete execution time of the kernel. Therefore, it only consumes between 24 and 29 watt.

The DRAM power consumption can hardly be recognized in Figure 5.30, however it also differs for the hybrid and the GPU-controlled versions. During execution time, in the hybrid version, the memory consumes about 3.9 watt, while in GPU-controlled version it requires only about 2.2 watt.

These results show that the lower power consumption of CPU and DRAM leads to a considerably lower power consumptions of the approaches using GPU-controlled communication and, therefore, to a better energy efficiency than the hybrid version.

5.9. Summary

This chapter shows how a GPU can be enabled to control an RDMA capable network device. This allows the creation of a put/get interface on the GPU.

To create a communication environment on the GPU, a host thread is required, as the GPU is not able to access the operating system. However, once the connection to remote GPUs are set up, the GPU is able to issue communication requests to the NIC by completely bypassing the host CPU.

Still, so far there are no performance benefits gained from this method compared to a hybrid model, in which the CPU controls the communication. The reason for this is that the overhead of creating work requests and handling completion notifications on the GPU is much larger than on the CPU.

Two things slow down the performance on the GPU: Creating work requests and handling completion notifications are sequential workloads, which hardly can be parallelized. GPUs, in contrast to CPUs, are not optimized for this kind of workload.

The second reason is that some communication resources are allocated on host memory, since GPU memory is a scarce resource and due to software architecture, not all communication resources can be allocated on GPU memory. Accesses to host memory are routed through the PCIe-bus and so slow down the performance.

Another problem is the inter-block synchronization on the GPU. GPUs blocks are non-preemptive and an inter-block barrier on the GPU can easily result in a deadlock. The best way to synchronize computation and communication on the GPU is using *communication kernels*, which are added to the same stream as the compute kernels.

Using this method, the performance of applications using GPU-controlled communication is still worse than the performance of hybrid applications, but, depending on the ratio between communication and computation, the performance difference is very low. However, in the hybrid approach, the CPU requires additional power and therefore the energy efficiency of GPU-controlled communication is better than the energy efficiency of the hybrid version. Due to technical, economical, and ecological reasons, energy efficiency become more and more important for future high performance systems. Therefore, this factor should not be disregarded. The energy is saved by relieving the CPU. Alternatively, one can argue that the CPU is available for other tasks and then a higher performance of a heterogeneous application could be reached.

However, the energyefficiency shows that allowing the GPU to control the communication is a good approach for future GPU-accelerated high performance systems. Still, for better efficiency, a communication interface is required that fits better to the GPU programming model since even higher energy savings would be possible if the GPUs could handle communication more efficiently, especially for small messages. The next section will introduce a global GPU address space as one possible solution.

6. Global Address Space for GPUs

In the previous chapter, we analyzed the performance of GPU-controlled one-sided put/get communication for distributed GPUs. The results show that the overhead of creating work requests and handling notifications on the GPU surpasses the data transfer latency of small data transfer sizes. This sequential work does not fit into the massive parallel execution model of GPUs. On an application level benchmark, further problems are added, because inter-block synchronization is required, which is not naturally supported by GPUs and adds further overhead to the communication.

Therefore, GPU-controlled one-sided communication does not bring any performance benefits compared to a hybrid solution so far. However, the power and energy analysis shows that the CPU thread that is required to control the communication consumes so much power that the total power consumption of the hybrid application is higher than the power consumption of an application in which the GPU controls the communication. This results in a better energy efficiency for GPU-controlled communication, although the performance is worse. Therefore, controlling the communication from the GPU appears to be useful. Still, especially for small messages, a communication model is required which fits better to the GPU execution model.

The experiences from the pervious chapters imply the following requirements for GPU-controlled communication:

- The communication should *maintain* the massive parallel GPU execution model instead of allowing communication despite the GPU execution model.
- Sourcing and synchronizing of communication requests should be as simple as possible, reducing the communication overhead on the GPU to a minimum.
- The memory footprint for communication resources on the GPU should be as small as possible since GPU memory is a scarce resource.
- The CPU can be used for setup; however, once the communication framework is set up, the GPU should be able to communicate by completely bypassing the host CPU.
- Accesses and copies to host memory should be avoided if possible to provide small data transfer latency and avoid the PCIe bottleneck.

This chapter introduces one possible solution that fulfills these requirements: a hardware-supported global address space, shared between all GPUs in a cluster. In Figure 6.1 on the next page, the concept of such a distributed global address space is presented.

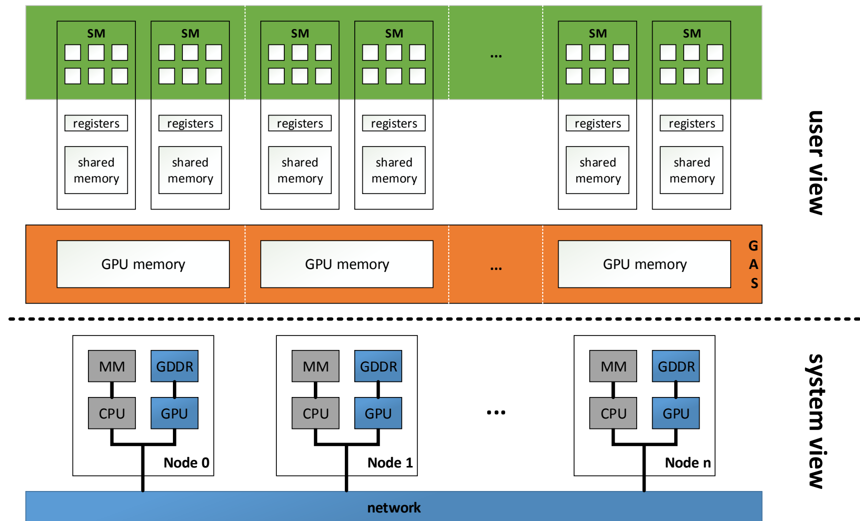


Figure 6.1.: Simplified system and user view of a cluster using a global address space

The lower part of Figure 6.1 shows the simplified structure of a GPU cluster. The cluster consists of multiple nodes, equipped with a single GPU and connected via a high performance interconnect. Each GPU in the system is a discrete system which only has direct access to its own, local device memory.

RDMA-technologies allow the creation of shared memory segments on GPU memory and a direct data transfer between these segments, but it is not possible to directly access the memory of remote a GPU with a simple load/store instruction. All communication and data transfers between the individual nodes are translated to a special communication functions like put/get or send/receive, as discussed in the previous chapters.

In contrast to this, the top part of Figure 6.1 depicts how a global address space transforms this system view into a simplified user view. The distributed device memories of all GPUs in the system are aggregated into one global GPU address space. Each thread, independently of the GPU it is running on, can access every part of this global address space directly.

All other resource, like the SMs, the caches, or the shared memory, are still local to one GPU. From the point of view of a single GPU, the memory of remote GPUs is mapped into the own, local address space. Although such a global address space allows access to the distributed memories of all GPUs in the system, locality is still an important factor for scalability since the latency of a remote memory access is higher than the latency of a local access.

Accesses to remote memory regions should only be used for synchronization and communication tasks, while only local memory should be used for computations which require frequent memory accesses. In this way, a Global GPU Address Space (GGAS) corresponds to the PGAS model, as described in section 2.1.5.

However, in most known PGAS languages, the accesses to remote memory are internally translated (by the compiler) to one- or two-sided communication requests. On a GPU, this is not an appropriate solution to meet the previously announced claims. Either these communication requests have to be performed by the GPU or by the CPU. Then, we have the same starting point as in the previous chapters and nothing would be gained by using a global, shared address space.

Instead, an efficient hardware support for a global address space is required to avoid the translation of remote load/stores to one- or two-sided communication requests. The hardware should directly forward load and store instructions to a remote node. In this case, the previously announced requirement for a communication interface are fulfilled:

- Communication using global, shared address space is realized with simple load and store instructions. These instructions are simple and can be performed by many threads in parallel, maintaining the massive parallel and bulk synchronous GPU programming model.
- A hardware-supported global address space requires structures for organization of the global memory but no additional memory for queues or notifications. Therefore, the memory footprint is very small.
- A hardware-supported global address space allows bypassing the CPU,if the GPU can directly access the network hardware.

In this chapter, such a hardware-supported global address space for GPUs is discussed. The Shared Memory Function Unit (SMFU) of the Extoll device provides the necessary hardware functionality to support such a global address space. The support for GPUs requires some adaptions in the device drivers, which will be explained in more detail in this chapter.

We analyze the performance, advantages, and disadvantages of this communication model for distributed GPUs and compare it with the previously described approaches.

6.1. Related work

As multi-GPU programming becomes more and more important, different models for inter-GPU data transfer were discussed, as were shared memory approaches. In [139], an environment for distributed texture memory for distributed GPUs is implemented and discussed. The framework allows programs to run across multiple GPUs, which can be distributed over different nodes, with a common consistent but distributed address space.

The core of the system is a directory-based shared memory system, which handles the memory management transparent for the user. However, the GPU memory is only handled as cache while the host memory builds the main memory of this system.

An underlying CPU-controlled framework controls all communication and the GPU data are always buffered in main memory. However, the benefit of this model is that it provides better programmability for multi-GPU programs.

A similar approach is GPUZippy [140], which follows the concept of global arrays (GA) [141]. A global array is a multidimensional array that is distributed among multiple GPUs, providing a virtual, shared address space among these GPUs. However, internally, MPI is used for communication and the CPU handles the communication. The goal of this approach is to provide a better programmability for multi GPU programs instead of providing high performance.

Between multiple GPUs on the same node, the unified virtual address space (UVA), introduced with CUDA 5, provides a virtual address space for all GPUs on the same node. The GPUDirect peer-to-peer technology allows load/store access to the memory of another GPU on the same node, without any copies in host memory or CPU involvement. The GPU memory controller forwards an access to another GPU directly over the PCIe-bus. The idea of the global GPU address space discussed in this chapter is to extend this concept to GPUs, that are distributed among multiple nodes.

6.2. GPU memory coherence and consistency

Working with global and distributed memory systems, the coherence and consistency of GPUs are important. Therefore, the next paragraphs give a short overview of the coherence and consistency models of GPUs.

6.2.1. Coherence

Caches were introduced to GPUs to reduce accesses to global device memory rather than to hide latency. A modern Fermi GPU provides four different kinds of caches which are now explained in more detail.

The *constant cache* is a *read-only* cache which allows fast access to constant variables. Every SM has one constant cache. The *texture cache* is also a read-only cache, which is used for memory regions which are defined as textures.

However, since neither constant variables nor textures should be changed during kernel runtime, these read-only caches are not coherent. Constant variables and textures should not be touched from outside (from host) or by another kernel. Before a new kernel is launched, these caches are invalidated.

Every SM has its own *L1 cache*, which uses the same hardware unit like the shared memory. L1 caching in Kepler GPUs is reserved only for local memory accesses, such as register spills and stack data, but not for accesses to the global device memory [142].

These accesses are cached in the *L*2 cache which is shared between all SMs on the GPU. Therefore, no cache coherency protocol is required.

6.2.2. Consistency

The consistency model of GPUs is *relaxed*. This means that it is not defined when an access to the global device memory is visible for other threads and reordering between the accesses is allowed. Sequential consistency is only ensured within one thread. In other words:

- The order in which a CUDA thread writes data to shared memory, global memory, page-locked host memory, or to the memory of a peer device is not necessarily the order in which the data is observed being written by another GPU or host thread.
- The order in which a CUDA thread reads data from shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which the read instructions appear in the program for instructions that are independent of each other. [143]

CUDA provides several functions to guarantee memory consistency manually:

__threadfence_block all read and write operations to shared and global device memory previous to the threadfence_block call are observed to occur before all read and write operations after the threadfence_block call, for all threads of the block of the calling thread.

__threadfence as threadfence_block, but additionally all read and write accesses to global device memory before the threadfence call are observed to occur before all read and write accesses after the threadfence call, for all thread-blocks running on the GPU.

__threadfence_system as threadfence_block, additionally all read and write accesses to global memory, page-locked host memory, and the memory of a peer device before a threadfence_system call are observed to occur before all read and write accesses of the calling thread after calling threadfenc_system, for all threads on the GPU, the host, and a peer GPU.

Using a global, shared address space for distributed GPUs, the last memory fence function, *threadfence_system* should be used to guarantee the order of requests to the shared memory area.

6.3. Hardware support for distributed global address spaces

At the beginning of this chapter, we stated that for a hardware-supported global address space, a hardware unit is required that allows the forwarding of load and store requests to a remote node. The shared memory function unit (SMFU) of the Extoll device provides the required support for such a global address space [17]. The SMFU aggregates memory on different nodes to a single, non-coherent, distributed shared memory area. This is different to approaches like ScaleMP or NumaScale which aggregate the complete resources of a distributed system to a single coherent system image. Cache coherence is one of the main bottlenecks of these systems. The idea of a PGAS system is to provide local coherency domains while the global memory is non-coherent. This is also in line with the GPU coherence and consistency model.

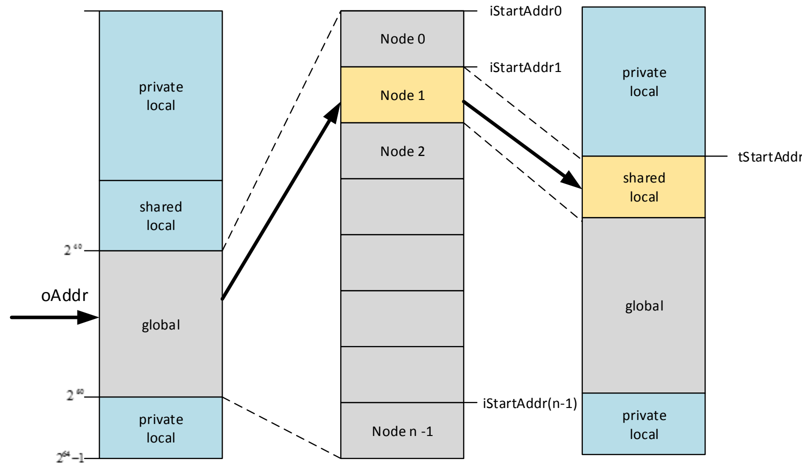


Figure 6.2.: SMFU Address Space Layout[17]

In this section, the functionality of the SMFU is illustrated, first without observing GPUs. Figure 6.2 shows the address space layout on a machine using the SMFU [17] to aggregate the memory. On the left, the address space of a target node is shown. The address space consists of the local memory and the global memory area. In this example, all addresses between 2^{40} and 2^{60} belong to the global memory. The global memory area is split into intervals, where every interval belongs to a remote node. Every node ports a part of its own, local memory to the global memory area. The SMFU is responsible for forwarding read/write access from an address in the global address area to the destination node.

For the SMFU, the global addresses correspond to one of the base address registers (BAR) of the Extoll device. A write or read to an address within this BAR is forwarded to a remote node. For user space processes, the BAR is mapped to the user space.

The simplest way to use the SMFU is the *interval* configuration. Start (`iStartAddr`) and end (`iEndAddr`) addresses of the intervals and the respective nodes are configured with the Extoll device registers.

For every node, also the target destination address (`tStartAddr`) is defined. This address marks the start of the local shared memory area. The SMFU forwards an incoming request to this address. To minimize the communication overhead, the shared memory must be physically continuous to avoid logical to physical address translation.

The SMFU is divided into two parts, the *egress unit* and the *ingress unit*. A load/store request to an address within the SMFU BAR is forwarded to the egress unit. The egress unit receives this load (*non-posted)* or store (*posted*) request to the address *oAddr*. If this address lies in the interval *n*, the egress unit determines the *global address (gAddr)* with:

$$gAddr = oAddr - iStartAddr_n$$

which *iStardAddr_n* beiing the start address of the hit interval n. With this address, a network package is created and sent to the node the hit interval is assigned to. On this

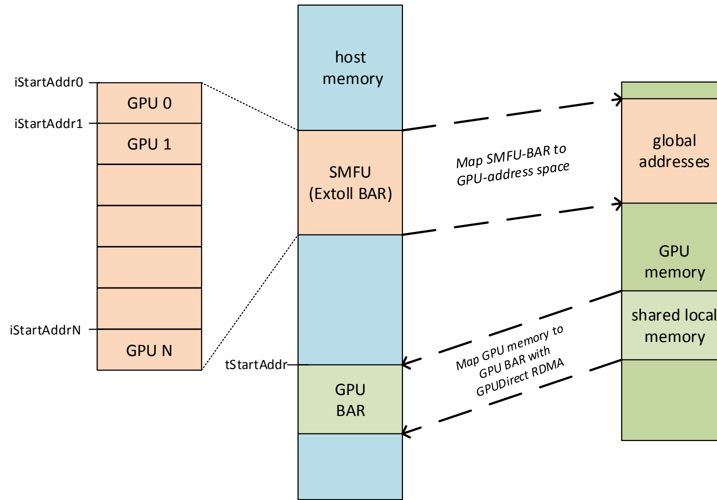


Figure 6.3.: GGAS address space layout

node, the ingress unit receives the package and calculates the *destination address (tAddr)* with:

$$tAddr = gAddr + tStartAddr$$

tStartAddr is the start address of the shared memory region on the destination node. For a store request, the ingress unit forwards the transaction to the memory address. For a load request, the data is read and then sent back to the destination node.

Since the number of intervals is limited, it is also possible to use an address mask instead of the interval configuration. In this case, the destination node is coded into the global address *gAddr*. In simplified terms, the node ID can be determined with the address mask and a shift operation :

$$iNodeID = (gAddr\&mask) >> shif_count$$

On the target side, the destination address can be determined with the inverted mask:

$$tAddr = (gAddr\& \sim mask) + tStartAddr$$

The operation is very similar to the one described in [16] for the T3E multi-processor. Due to the simplicity, only minimal latency is added to the communication process.

6.4. Extending the global address space for GPUs

To create a Global GPU Address Space (GGAS), the concept of the shared memory described above is extended to GPU memory. For this, two steps are required. In the first step, the ingress unit of the SMFU has to be configured to forward an incoming request to GPU device memory instead of host memory. To allow this, the GPU memory

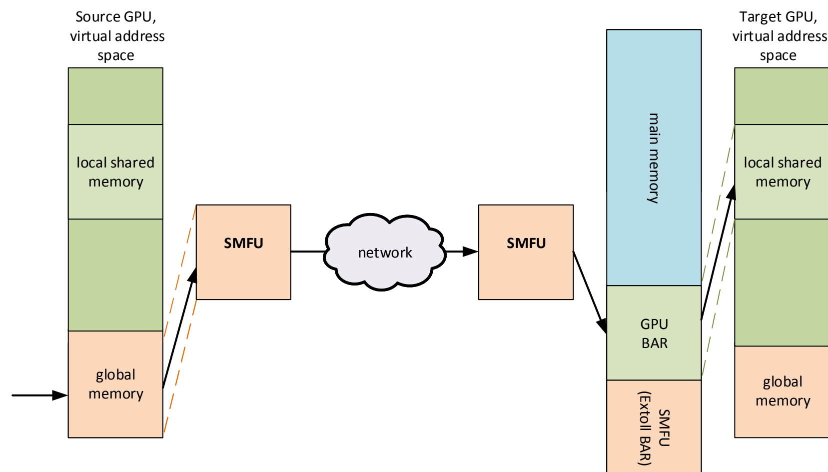


Figure 6.4.: Simplified way of a store request using GGAS

requires an address in the physical address space of the host system, which is allowed by GPUDirect RDMA. Therefore, a part of the GPU device memory is locked and mapped to the physical continuous, addresses of the BAR of the GPU, using GPUDirect RDMA. This memory is the local shared memory of the GPU.

The SMFU is now configured to forward an incoming read or write request to an address within the GPU BAR by setting the *t*StartAddr to the address of the GPU BAR, as shown in Figure 6.3.

Currently, with a single Extoll device, it is only possible to share the memory of a single GPU since only one target address can be configured. Two GPUs in the same node would map their memory to different BARs, which cannot be managed by the SMFU.

The second step to create a GGAS is to enable the GPU to access the global address space. Therefore, the shared memory region must be accessible to the GPU. As mentioned in the previous section, the global address space is accessible over one of the BARs of the Extoll device. This BAR has to be mapped to the virtual GPU address space, as illustrated in Figure 6.3.

In this work, we use the interval configuration of the SMFU. The intervals of the SMFU are evenly distributed over the involved GPUs. Since only *nodes* can be targeted, in fact, the intervals are configured to forward an outgoing request to a remote node. This node, however, is equipped with a single GPU and the SMFU is configured to forward an incoming request to the memory of this GPU.

Figure 6.4 shows how a store request to a remote GPU is forwarded to this remote GPU, using the GPU configuration for the SMFU.

A GPU thread writes to an address that points to the mapped SMFU BAR and, therefore, the GPU memory controller forwards this request to the SMFU. Depending on the hit interval, the SMFU forwards the request over the network to a remote node. On this node, the SMFU accepts the package and forwards the request to the target address in

GPU BAR. Finally, the GPU memory controller accepts the data and forwards the package to GPU memory.

By this, the GPU can directly read and write the memory of a remote GPU, by completely bypassing the host CPU. The host CPU is only required for setup, and thus the locking of GPU memory and the configuration of the SMFU.

6.4.1. Restrictions for the global memory size

Currently, the size of the global address space is limited to 256 MB, because the BAR size is limited to 256 MB. These 256 MB have to be split over all GPUs. So for eight GPUs, every GPU only can port 32 MB of its device memory to the global address space and for sixteen GPUs, only 16 MB of the memory of every GPU can be mapped to the global address space. This currently limits the capabilities of the GGAS model.

However, theoretically, the SMFU supports larger BAR sizes, if the host system supports this. In this case, the BIOS often assigns the BARs to addresses above 2^{40}. The GPU memory controller, however, only supports addresses with 40 bits or less, so addresses above 2^{40} cannot be managed by the GPU memory controller. This prevents mapping the SMFU BAR to the GPU address space. Possible solutions for this problem are enlarging the address size for GPUs or enabling the BIOS to assign BAR addresses to a lower range. However, both solutions are currently not enabled. Therefore, to date the shared address space is limited to 256 MB. Parts of this work were already published in [144] and [145].

6.5. The GGAS Software

Figure 6.5 shows the software stack for a GPU application using GGAS for communication. The GGAS software stack has parts in the user space,the kernel space, and on the GPU.

The user space part is required for setup, starting and stopping GPU kernels, and for communication with the device drivers, since these tasks cannot be handled by the GPU. However, once the setup phase is completed, the GPU can directly access the SMFU hardware by completely bypassing the host CPU.

The GGAS GPU-library provides basic functions to allow communication with GGAS on the GPU. In the following, the individual parts of this software stack are described in more detail.

In the setup phase, the global address space is created and initialized. This is realized by the GGAS host library and the device drivers. Figure 6.6 on page 149 illustrates the procedure of initializing the global GPU address space. First, the local part of the shared memory (*gpu_local_pointer*) is allocated with `cudaMalloc`. This requires an access to the GPU device driver. The next step is to lock the GPU device memory, map it to the GPU BAR, and configure the SMFU to use this GPU BAR as target address for incoming requests. For this, a small device driver *(ggas_driver)* is used. The local GPU memory pointer (gpu_local_pointer) is forwarded to the kernel space using a pinning function (*ggas_pin*) of the GGAS device driver. In the old version of GPUDirect RDMA, also the

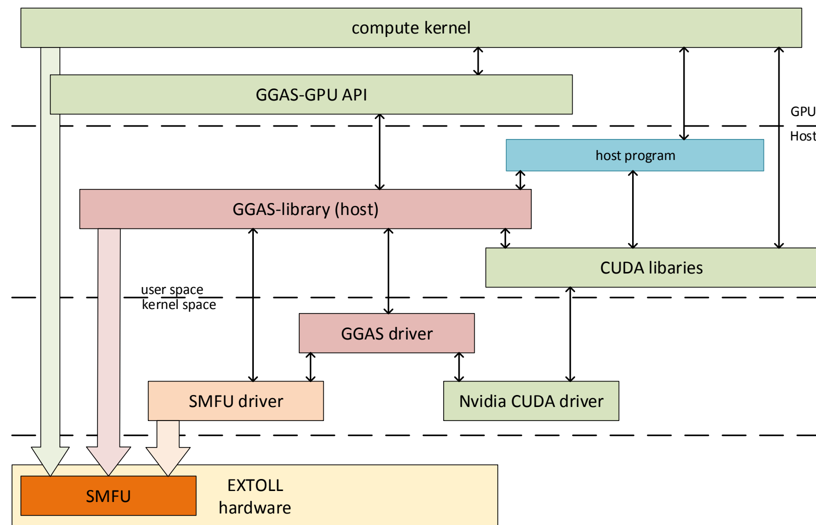


Figure 6.5.: GGAS software

two tokens *p2pToken* and *vaSpaceToken* were required and also forwarded to kernel space (see section 3.3.1). However, the GGAS driver uses GPUDirect RDMA to lock and map the GPU memory to the GPU BAR by calling the nvidia kernel function *nv_p2p_get_pages*. The returned BAR address is used to configure the target start address *tstartAddr* for the SMFU (*smfu_set_intervalls*). This function also splits the global address space among the GPUs by defining the intervals.

6.5.1. GGAS setup

In the next step, the SMFU BAR is mapped to the GPU address space. To do so, the SMFU BAR is first mapped to the user space with memory mapped I/O, using the mmap function of the SMFU device driver. This allows the host direct access to the memory of remote GPUs with simple read and write instructions, using the virtual *host_global_address*. This is especially useful for testing and debugging – but it also allows the host to transfer small data sizes to a remote GPU in a fast way. The virtual host address (host_global_address) cannot be used on the GPU, a read or write access on the GPU would result in a segmentation fault.

To allow the GPU to access the global memory, the SMFU BAR has to be mapped to the virtual address space of the GPU, as shown in Figure 6.3. For that, the MMIO host address (*host_global_address*) is handed over to the GPU device driver, using the `cudaHostRegister` to register this address for the GPU.

Again, the low level driver patch, described in section 3.3.3, is required to allow the registration of MMIO addresses for the GPU. If the MMIO address is registered for the GPU, the function `cudaGetDevicePointer` returns a pointer in the virtual address space of the GPU (*gpu_global_pointer*) which can be used in a CUDA kernel to access the global memory space.

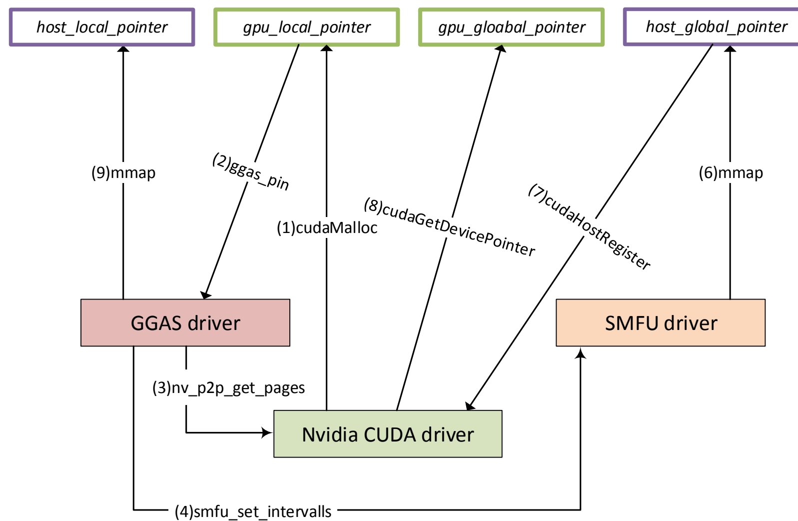


Figure 6.6.: Initialization of the global GPU address space

The GGAS device driver also allows a direct mapping of the GPU BAR to the user space. Therefore, there are two ways to access the local part of the shared GPU memory from host, firstly over the BAR of the GPU (*host_local_pointer*) and secondly over the mapped BAR of the SMFU (*host_global_pointer*), taking a detour over the SMFU. The GGAS-host-library provides functions for both, but for latency reasons, the direct way is preferable.

6.5.2. GGAS GPU API

To allow a GPU kernel to communicate using GGAS, a small API for the GPU was developed to provide access to the shared address space and to manage the communication resources. Every GPU using GGAS has a unique ID for identification. The local GPU ID and the number of GGAS-nodes are stored in the constant GPU memory. The same applies to the size and the shared memory address.

Remote read and write

The simplest way to communicate in GGAS is with remote read and write instructions. To get a pointer to the shared memory area of a remote GPU, the function `_ggas_get_pointer_of_node(int remote_node)` is used. Listing 6.1 shows how GGAS can be used to write to the memory of a remote GPU. This remote write function can be called by multiple threads in parallel. All threads copy data to the memory of the remote GPU in a parallel manner.

Listing 6.1: Simple remote write example with GGAS

```
__device__ void remote_write(float* input,int remote)
{                                                          2
  float *remote= _ggas_get_pointer_of_node(remote);
  remote[threadId]= input[threadId]                        4
}
```

Local shared memory

As on the host, the local part of the global shared memory can be accessed in two ways. Once over the address in the global address space (*gpu_global_poiner*), one is over the original address in the GPU address space (*gpu_local_poiner*) which is assigned to the memory when it is allocated with `cudaMalloc`. For local accesses, the local address should be used. Otherwise, the local access is forwarded over the PCIe-bus to the SMFU, that forwards it to back to the GPU. This detour would result in a much longer latency for a local memory access.

The function `ggas_get_poiter_of_node` with the local GPU ID returns the local device pointer. However, the function `ggas_get_global_pointer`returns a pointer to a global address and can be used for debugging.

Collective operations

The GGAS-API also supports two collective operations: a barrier (`ggas_barrier`) and an allreduce operation (`ggas_allreduce`). A part of the shared address space is used for these collective operations and therefore not available for remote read or write accesses. Since the implementation of these collective operations gives a deeper insight to the GGAS programming model, the implementation is explained in more detail in sections 6.7 and 6.8.

6.6. Micro-benchmark performance

In this section the basic performance of a global GPU address space for communication is evaluated. The goal of this section is to get an overview of the performance capabilities of this communication method and thus the strength and weakness of remote load and store operations for communication.

The test system consists of eight dual-socket nodes with two Intel E5-2630 Ivy Bridge processors and one NVIDIA K20 GPU. The Extoll network cards based on an FPGA run with 175 MHz core frequency and 64 bits wide data paths. The NIC and the GPU use PCIe 2.0 and share one root complex. For some of the benchmarks, twelve nodes are used. Two of the additional nodes are also Ivy Bridge dual socket machines with two Intel E5-2630 processors. The other two nodes are Intel Sandy Bridge machines with two Intel E5-2609 processors. All four additional nodes are also equipped with one K20 GPU.

Listing 6.2: Code example using GGAS, pingpong Benchmark

```
__device__ ping ( int remote_id )
{
 volatile int* local = __ggas_get_ptr_of_node ( ggas_id );
 long* remote = __ggas_get_ptr_of_node (remote_id);
 int ix = threadIdx.x + blockIdx.x * blockDim.x;
 long tmp;
 // start collective ping by all threads
 remote [ ix ] = 1;
 // poll collectively for pong
 do {
 tmp = local [ ix ];
 } while ( !tmp );
 local [ ix ] = 0; // reset for next polling
}
```

6.6.1. Latency

As first tests, we run read and write latency benchmarks. For the write latency, a pingpong benchmark is used. As GGAS benefits from a collaborative use by multiple threads, an extended parallel version of the pingpong test was developed.

This parallel version starts a bundle of threads in parallel on the GPUs. On the ping side, all threads write in parallel to the shared memory of the remote GPU. On the pong side, all GPUs poll on the local part of the shared memory until the memory is updated. Then, the local data is reset and the threads on the pong side write back to the global shared memory of the ping GPU. Listing 6.2 shows a code snippet for the ping side. The treads write to different but consecutive addresses in remote memory.

Since the execution of a CUDA thread-block is non-preemptive, the possibility of deadlocks is present if more threads are scheduled than cores are available. Our experiments validate this, and for an Nvidia Kepler-class K20 GPU. up to 8,192 threads for the pingpong benchmark can be started without running into unsafe situations. This number only applies to this pingpong test. The exact number of threads for a given workload depends on the overall resource usage, including shared memory and registers. The remote pointer in Listing 6.2 has to be defined as `volatile`. Otherwise, the threads would read the remote value (line 11) from the registers or the L2 cache and not repeatedly from the remote memory.

Listing 6.3: Code example for remote read using GGAS

```
__device__ remoteRead(long* dest)
{
  long* remote = __ggas_get_ptr_of_node (remote_id);
  dest[idx] =remote[idx]
  __threadfence_system();
}
```

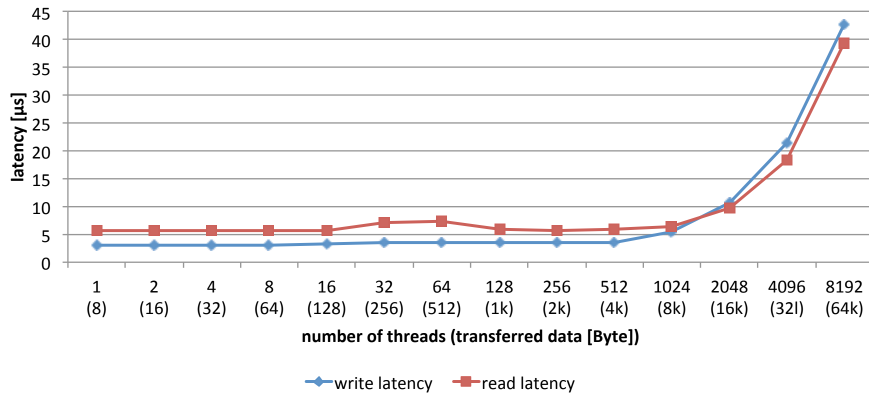


Figure 6.7.: Latency for communication using GGAS

For the *read* latency, another benchmark is required. In this benchmark, the threads simply load a value from the memory of a remote GPU and write it to their own local device memory as shown in Listing 6.3. The fence operation is required to ensure that the read operation is executed before the next iteration is started. The read benchmark can be started with more than 8192 threads, since the risk of a deadlock is not present.

Latency results

Figure 6.7 shows the results of the latency benchmarks. Every thread reads or writes a *long* value in remote memory, so 8 bytes are transferred with a single load/store instruction. For the write latency, the half round trip latency of the pingpong benchmark is presented.

Up to a data transfer size of 4 kb or for up to 512 parallel threads, the write latency is approximately 3 μs and the read latency is aboutt 5.6 μs. This results surpasses clearly the results for put/get communication presented in the previous chapter.

Up to a transfer size of 1,024 bytes, the write latency is better than the read latency. However, for larger transfer sizes – or a larger number of threads –, the read latency gets a little better than the write latency. However, the reason for this may also be the simpler structure of the read latency benchmark.

6.6.2. Bandwidth

To measure the sustained read and write bandwidth using GGAS, we implement two different bandwidth benchmark types:

Remote load and stores This benchmark uses remote load and store instructions to copy data between a local and a remote memory region. The benchmark is started with 8,192 threads, grouped into blocks of 64 threads each. These threads read and write data from and to remote memory in parallel, using the coalescing capabilities of the GPU.

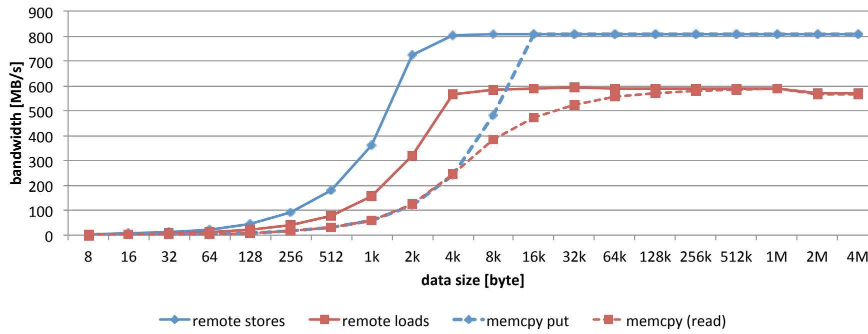


Figure 6.8.: Bandwidth for communication using GGAS

cudaMemcpy Dynamic parallelism allows the GPU to call asynchronous memory copy operations in a CUDA kernel. This can be used to transfer data between a local and a remote memory buffer. Therefore, this benchmark starts a CUDA kernel with only one thread. This thread initiates multiple CUDA copy operations on the GPU and synchronize these copy operations with `cudaDeviceSynchronize`. These one-sided communication operations correspond to one-sided put and get operations.

Bandwidth results

Figure 6.8 shows the results of the bandwidth benchmarks. The maximal bandwidth for remote store/put operations is about 800 MB/s and for remote load/get operations about 600 MB/s. We see two reasons for the worse read bandwidth:

First, the GPU can only handle a limited number of outstanding remote read requests. If the number of remote read requests is too large, it takes longer to complete a request. The evaluation of the Extoll register file, using the *sys*-filesystem, shows that the maximal number of outstanding read requests on the SMFU is 25 while the read-bandwidth test is running. This suggests, that the GPU cannot handle more then 25 outstanding read requests.

Second, a remote read request requires two non-posted device-to-device accesses over the PCIe-bus. On the sourcing side, the GPU loads the values from the SMFU, on the target side, the SMFU loads the values from the GPU. So here, the bandwidth is limited due to the PCIe bottleneck described in chapter 3. Using remote write operations, this PCIe bottleneck can be avoided: On the sourcing side, the GPU pushes the data to the SMFU and on the target side, the SMFU pushes the data to the GPU. Therefore, for remote write operations, the decrease of the bandwidth for transfer sizes larger than 1 Gbytes cannot be observed.

For smaller data transfer sizes, remote loads and stores provide a better bandwidth than their counterparts using *cudaMemcpy*. This is caused by the overhead of calling *cudaMemcpy* on the GPU, the dynamic parallelism overhead. However, for transfer sizes larger than 16 kbytes for remote writes and 256 kbytes for remote reads, this overhead becomes negligible compared to the data transfer latency.

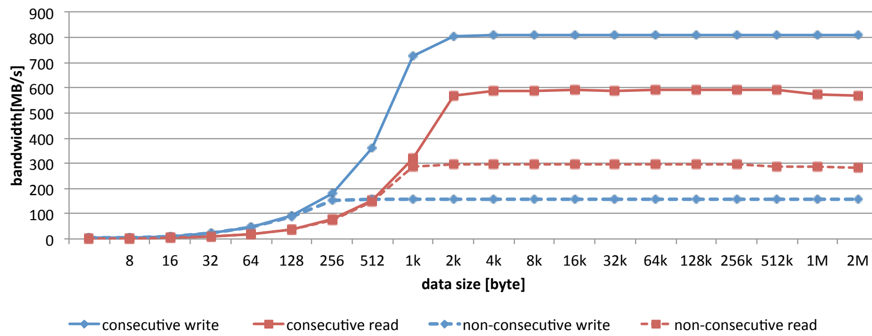


Figure 6.9.: Coalescing effects for remote memory accesses

6.6.3. Coalescing effects

Data with remote load and store instructions theoretically causes a network transfer for every remote memory accesses. But, in fact, the GPU does not forward every single request to the NIC but coalesces accesses to consecutive addresses, as shown in section 2.2.2 for the local device memory. In the following, the impact of this coalescing capabiliy for accesses to remote GPU memory is examined.

Now, the bandwidth of thread-collective load and store operations to remote memory regions, using consecutive and non-consecutive addresses, is measured. For the nonconsecutive case, only every second value is written or read.

Figure 6.9 shows the results of these benchmarks. The bandwidth of consecutive writes is about seven times higher than the bandwidth of writes to nonconsecutive addresses. The bandwidth of consecutive reads is twice the bandwidth of reads to non-consecutive addresses. This is due to the coalescing abilities of the GPU memory controller.

The Extoll device allows counting of incoming requests to the SMFU. This feature allows counting the number of incoming requests for consecutive and non-consecutive reads and writes. Table 6.1 shows the number of written and read values and the number of incoming requests to the hardware for both cases.

The memory controller of the GPU coalesces read and write accesses to consecutive addresses to packages of 16 values, which corresponds to 64 bytes for integer values with a size of 16 bytes. For non-consecutive writes, the number of incoming requests matches the number of the written values. Thus these accesses are not coalesced and the band-

Table 6.1.: Incoming requests to the SMFU for consecutive and non-consecutive remote writes

Values	1	16	64	256	1024	4096
consecutive writes/reads	1	1	4	16	64	256
non-consecutive writes	1	16	64	256	1024	4096
non-consecutive reads	1	2	8	32	128	512

width is massively slowed down. Remote read accesses to non-consecutive addresses are also coalesced. Since only every second value is read, twice as many requests are required and the bandwidth is halved.

6.7. GGAS barrier

A global address space model like GGAS requires explicit synchronization to ensure consistency. In parallel computing, a barrier is a synchronization primitive that guarantees that each thread or process reaches a specific point in its control flow before proceeding. Using GGAS, a barrier must also guarantee that all remote reads and writes to the GGAS memory previous to the barrier are completed before a thread leaves the barrier.

Therefore, to synchronize distributed GPUs, an efficient and fast barrier, developed for distributed shared memory architectures is used, similar to the barrier described in [146] for shared memory architectures or the barrier described in [137] for inter-block synchronization on the GPU. It is a lock free barrier that does not require atomic operations and avoids multiple writes to a single location. The barrier consist of two phases:

1. *Check-in phase:* For every GPU arriving at the barrier, a single thread sets the *check-in-flag*. This check-in-flag is located in the global address space. One *master GPU* waits for the flags of all other GPUs by polling on them. This can be done by multiple threads of one block in parallel, so every thread of this block is responsible for polling on the flag of one remote GPU. Once all GPUs have reached the barrier, the check-in phase in completed.

2. *Check-out phase* If all GPUs have reached the barrier, the *master GPU* sets the *check-out-flags* for all GPUs. The check-out-flags are also located in the global address space. Again, the check-out-flags can be updated by multiple threads of the master GPU simultanously. On all other GPUs, a single thread polls on the checkout-flag. If this flag changes, the GPU can leave the barrier.

To achieve good performance, the location of the flags is important. As shown in [146] and the previous section, for a strong scalability, remote loads should be avoided and polling on remote locations massively increases latency. We avoid remote loads by placing the check-in flags on the master GPU and the check-out flags on the respective GPU. Thus, remote loads are completely avoided and, instead, the barrier relies solely on a push model.

6.7.1. Intra-GPU synchronization

A GGAS barrier can be used for inter-GPU synchronization between distributed GPUS while inter-block synchronization on the GPU is still required. One possible solution is the use of *dynamic parallelism* for intra-GPU synchronization. The `ggas_barrier` can be employed for synchronization between separate compute kernels, as shown in Listing 6.4.

In this case, the compute kernels have to be synchronized with `cudaDeviceSynchronize` before the barrier is executed. The GGAS-barrier also can be used as CUDA kernel in-between two separate compute kernels, as shown in Listing 6.4. Here, the stream ensures the intra-GPU synchronization.

Another possibility for intra-GPU synchronization is to use *persistent threads*. In this case, only as many threads are started on the GPU as can run in parallel, so the risk of deadlocks can be avoided. Then, for inter-block synchronization, a barrier as described in [137] is used. In GGAS, a hierarchical approach is used: First, all thread-blocks within a GPU execute the check-in phase of the intra-GPU barrier; then the GGAS barrier for inter-GPU synchronization is executed before the intra-GPU barrier is completed with the check-out phase. However, in the last passage in section 5.8, we already argued that the use of persisted threads is not very efficient on GPUs. However, in the event that the ggas_barrier is called by multiple thread-blocks in parallel, the hierarchical approach is used.

6.7.2. Barrier performance

To evaluate the performance of the barrier, two different benchmarks are started. The first benchmark starts a single kernel from host with a sufficient number of threads. This kernel calls a ggas_barrier device function for several iterations, as shown in Listing 6.4. In the second benchmark, also a single kernel from host is started, but this one only with a single thread. This thread, however, starts the barrier as a *kernel* on the GPU, using *dynamic parallelism*, as shown in Listing 6.5. Figure 6.10 shows the results for both approaches.

The performance of the *barrier kernel* is worse than the performance of the *device function barrier*. This is due to the *dynamic parallelism overhead* [147], the overhead that is caused by starting and synchronizing kernels on the GPU. According to Figure 6.10, this overhead lies around 10 μs. The benchmark for the *device function barrier* does not count in the overhead of the synchronization compute kernels, which may be required for a real application. Both barriers show very good scaling. The latency for two GPUs is about

Listing 6.4: Barrier using dynamic parallelism

```
if(threadIdx.x==0)
  compute_kernel1<<<blocks, threads,0,0>>>(···);                          2
cudaDeviceSynchronize(); /*Make sure the kernel is completed*/
_ggas_barrier(); /*sync remote GPUs*/                                      4
if(threadIdx.x==0)/*start new kernel*/
  compute_kernel2<<<blocks, threads,0,0>>>(···);                          6
```

Listing 6.5: Barrier using a barrier kernel

```
compute_kernel1<<<blocks, threads,0,stream>>>(···);
ggas_barrier_kernel<<<1, 64, 0, stream>>>(); /*sync between two           2
    kernels*/
compute_kernel2<<<blocks, threads,0,stream>>>(···);
```

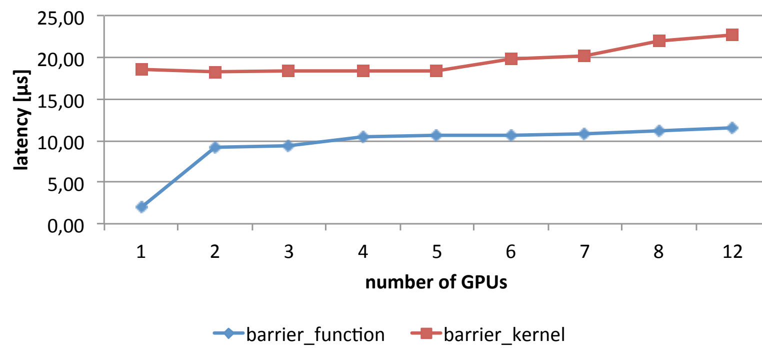


Figure 6.10.: Performance of the GGAS-barrier

9 μs and for eight GPUs it is 11 μ for the barrier function; for the barrier kernel the latency for two GPUs is about 18 μs, for eight GPUs it is 22 μs, so the latency of the barrier only increases marginally for more GPUs.

6.8. Allreduce and reduce using GGAS

Reduce and allreduce operations are among the most commonly used collective operations in high performance computing. Most parallel programming languages and libraries provide implementations of these operations. For host-controlled communication APIs or languages, every process submits an input vector to the collective operation. The function performs a reduction operation across the input vectors of all processes and returns a vector with the combined values. For the reduce operation, the output vector is returned by the root process, while for an allreduce operation, the result is returned by all processes. For a multi-threaded program, only one thread should call the reduce operation.

For a GPU, the concept of processes does not exist. Since GGAS is a communication model for inter-GPU communication, reduce and allreduce operations in GGAS are performed between multiple GPUs. Since GGAS maintains the parallel programming model of GPUs, the reduction operation is not only performed by a single thread but by multiple threads in parallel. Therefore, the allreduce and reduce operations should be used similarly to the ggas_barrier: either the operation is submitted to a stream as kernel function or used between two compute kernels, using dynamic parallelism. Thecollective reduce and allreduce functions also synchronize the distributed GPUs.

However, there are many possible implementations for reduce and allreduce operations using a hardware-supported global address space. Therefore, in the following, different implementation methods are introduced, discussed, and evaluated to find the best model for the different input sizes and numbers of GPUs. The idea of this section is also to get a better understanding of the behavior of the global address space for GPUs.

Listing 6.6: Reduce with remote read

```
idx = threadIdx.x+blockDim.x*blockIdx.x; /*thread ID*/
if(idx <vector_size) {                                          2
  result[idx] = 0;
  __ggas_barrier(); /*sync GPUs*/                               4
  for(i=0; i<ggas_nodes; i++)  /*read data from remote GPUs*/
   result[idx]+=pointer_to_gpu_i[i][idx];                       6
}
```

6.8.1. Reduction with remote read operations

A global address space allows direct reading of remote GPU memory, so in the naïve implementation of a distributed reduce operation, the input data are directly read from the global shared memory, as shown in Listing 6.6 for a global summation. Before the GPUs read the input data, first the GPUs have to be synchronized with a barrier to guarantee that the data in the global shared memory are valid. For larger input vectors, the reduction operation is performed by multiple threads in parallel, using the massive parallelism of the GPU.

For a reduce operation, only the root GPU executes the reduction operation while for an allreduce operation, the reduction operation is executed by all GPUs in parallel. This adds some network traffic to the operation, since every GPU has to read the input data from the memory of all other GPUs. On the other hand, data copies and additional synchronization between the GPUs can be avoided.

The main disadvantage of this implementation is that it requires remote reads. In section 6.6, it was shown that the bandwidth of remote reads is much lower than the bandwidth of remote writes. Hence, this implementation may be only suitable for small input vectors. For larger input vectors, another implementation which avoids remote read operations may be more performant.

6.8.2. Reduction with remote write operations

To avoid remote reads, the input data has to be copied between the individual GPUs using remote writes. Then the GPUs can read the input data from local device memory to perform the reduction. A reduce operation based on remote writes can be divided into two steps, an allreduce operation into three steps:

1. In the *collection* phase, input data are collected on one or more GPUs.
2. In the *reduction* phase, the reduction operation is performed on the collected input data.
3. For an *allreduce* operation, in the *distribution* phase, the result is distributed back to all GPUs.

In the naïve implementation, all input data are collected on one GPU, the *root GPU*. which then performs the reduction. This means that all other GPUs copy their input

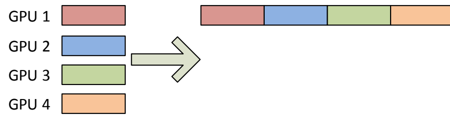


Figure 6.11.: Gathering

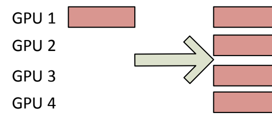


Figure 6.12.: Broadcast

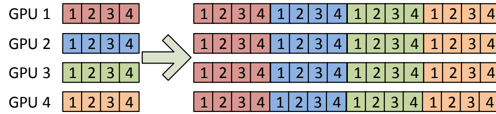


Figure 6.13.: All-gather

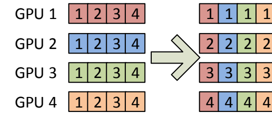


Figure 6.14.: All-to-all

data to the root GPU. This collection of data is similar to the MPI-*gather*-operation, illustrated in Figure 6.11. For an allreduce operation, the root-GPU distributes the reduction result back to all other GPUs, which corresponds to a *broadcast* operation, illustrated in Figure 6.12. Therefore, these methods are referred to as *gather-reduce* and the *bcast-allreduce* method in the following.

Using this approach, the allreduce operation requires a synchronization between the GPUs at least twice. The first one is required after the *gather* operation to guarantee that all GPUs have transferred their input data to the reduction buffer on the root GPU. The second one is required after the result is copied back with the *bcast* operation to inform the GPU that the output vector is available.

Although GGAS allows fast synchronization between the GPUs with a barrier, as shown in the previous section, every synchronization requires additional data transfers and adds overhead to the application. Especially for small sizes, this synchronization overhead may surpass the data transfer latency. The *all-gather* allreduce operation avoids this double synchronization. Using this method, *all* GPUs transfer their input data to *all* other GPUs in an *all-gather* manner, illustrated in Figure6.13.

This allows all GPUs to perform the reduction operation on their local device memory; and thus the second synchronization and broadcasting of the results are avoided. Still, the *all-gather* operation increases the amount of data transfers, so this method may only scale for a small number of input elements.

6.8.3. Work sharing: data distribution over multiple GPUs

For larger message sizes and a larger number of involved GPUs, the implementations described above may not scale. Either the same work is performed several times on different GPUs or a single GPU performs all the reduction work while all other GPUs idle. Moreover, the data collection and distribution cause a lot of network traffic between the GPUs, which may also slow down the performance.

To achieve a better scaling for a larger number of input elements and a larger number of GPUs, the work is distributed among the GPUs. We implemented two work distri-

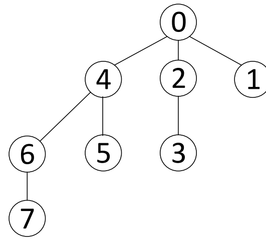


Figure 6.15.: Binomial tree for eight nodes

bution strategies which also could be combined for larger input vectors and a larger number of GPUs.

Tree-based design

For larger cluster sizes, a tree-based design is recommended to collect and distribute the data. However, some work has already been done in optimizing these trees for MPI, e.g.,[148, 149], and this is not within the scope of this work. Still, it is worth to mention that these structures can also be used for GGAS and GPU clusters. In the presented work, a binomial tree, as shown in Figure 6.15, for eight endpoints was implemented to compare the performance against other methods.

Work distribution

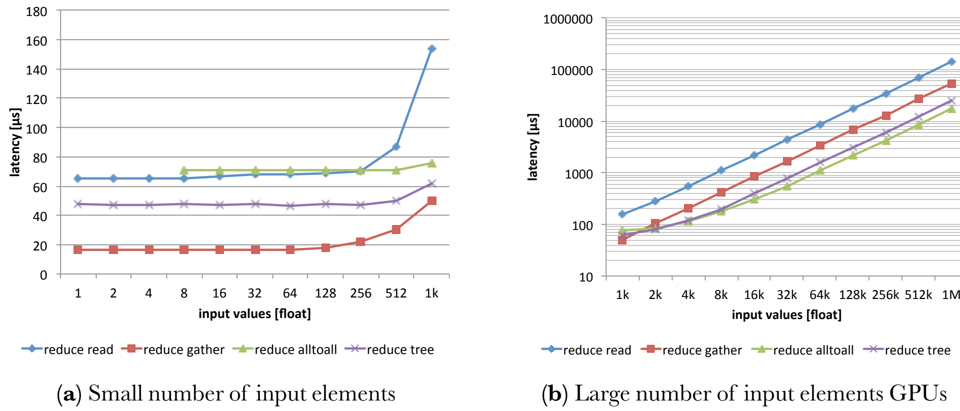
For a larger number of input-elements, the input vectors of all GPUs can be split up into smaller blocks and then evenly distributed among all GPUs. Analogous to MPI, this operation is called all-to-all (see Figure 6.14). This method is referred as the *all-to-all*-reduce/allreduce operation.

Every GPU executes the reduction operation on its part of the input data. For a reduce operation, all GPUs write their results back to the root GPU in a (*gather*) manner; for an allreduce operation, the results are distributed over all GPUs in an *all-gather* manner.

6.8.4. Performance results for the reduce and allreduce operation

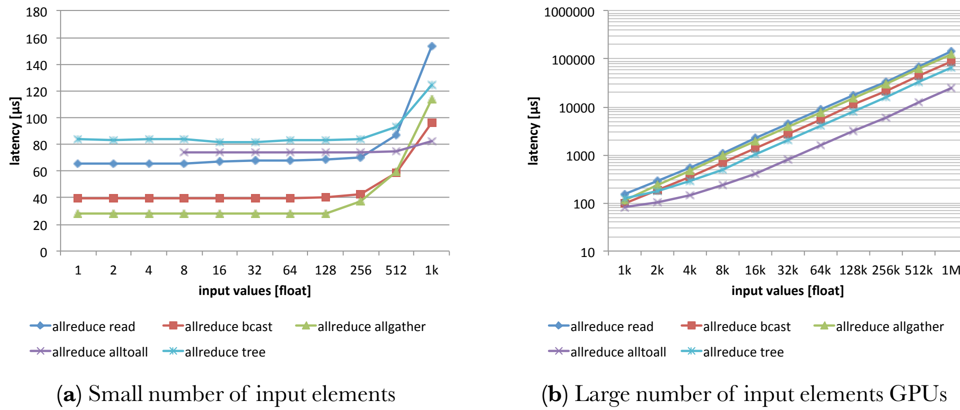
In this section, the performance results for the reduce and allreduce operations with GGAS, using the different approaches described above, are discussed. We consider the results for eight GPUs. We run the operations with `float` values as input. Figure 6.16 shows the results for the reduce operation. The input vector is only split if the number of values is equal or larger than the number of GPUs; therefore, some of the values of the all-to-all method are missing.

For small input sizes, the *gather* method, whereby all GPUs send the input data to the root GPU, is performing the best. The work distribution, using a tree-based approach or using a work distribution with an *all-to-all* data transfer, is performing worse than the gather-approach. For only a few input elements, the overhead due to work distribution and creating tree-structures outperforms the overhead of collecting the data on a single



(**a**) Small number of input elements (**b**) Large number of input elements GPUs

Figure 6.16.: Results of the reduce operation on eight GPUs

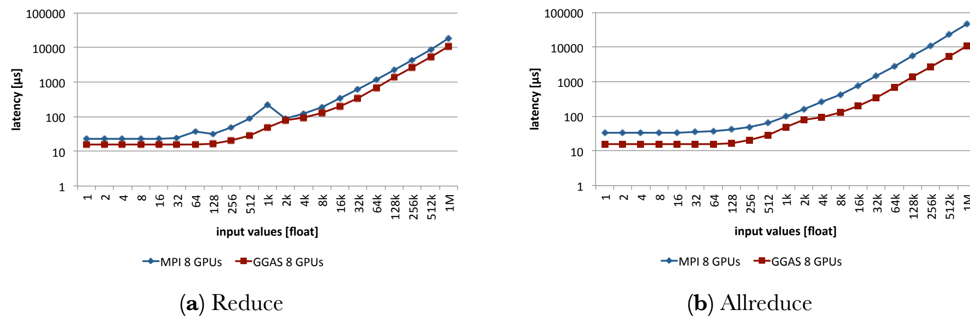


(**a**) Small number of input elements (**b**) Large number of input elements GPUs

Figure 6.17.: Results of the allreduce operation on eight GPUs

GPU. The reduction operation based on remote reads is performing worse than other approaches and the performance gets significantly worse for larger input vectors. For larger messages, the work distribution pays off and the all-to-all method, whereby the input data are distributed among the GPUs, is performing the best. The tree-based method is only slightly worse. However, the benchmark runs only on a GGAS system with eight GPUs, so this may be different for a larger cluster.

Figure 6.17 shows the results of the allreduce benchmarks. For small input vectors, the *allgather* method is performing best. Using this method, all GPUs transfer their complete input data to all other GPUs, and all GPUs perform the reduction operation on their own, local memory. The overhead that is caused by the several data transfers and the multiple execution of the same reduction operation is smaller than the overhead that is caused by the two-time synchronization if the root GPU first collects all input data and then sends the result back to all other GPUs. Similar to the reduce operation, for a small number of input elements, the work distribution does not pay off.



(**a**) Reduce (**b**) Allreduce

Figure 6.18.: Results of the reduce and allreduce benchmarks for an optimized GGAS version and MPI

For a larger number of input elements, the work distribution, using an all-to-all data transfer to distribute the input vectors among the GPUs and later collect the results with an all-gather operation, is clearly performing the best. A tree- based approach currently is not very efficient. Again, the benchmark runs only on a GGAS system with eight GPUs, so this may also be different for a larger cluster.

The read-method is not performing well for any problem size. Therefore, remote read operations, especially from multiple GPUs in parallel, should be avoided.

6.8.5. Comparison to MPI

In this section, the performance of the optimized reduce and allreduce operations is compared against the performance of MPI reduce and allreduce.

The optimized GGAS reduce operation uses the *gather* method for small messages and the *all-to-all* method for larger messages. The optimized allreduce operation uses the *all-gather* method for small messages and the *all-to-all* method for large messages.

We compare the GGAS version with openMPI, version 1.6.1, using the RMA and VELO unit of the Extoll device. This comparison seems to be fair, as both methods use the same FPGA. A comparison to Infiniband would result in a comparison between an ASIC with a peak bandwidth of 32 Gbps and an FPGA with a peak bandwidth of 9.8 Gbps and not a comparison between two different GPU communication methods.

Although the GPUDirect RDMA technology allows the network hardware to directly read and write GPU memory, this technique cannot be used for MPI for reduce and allreduce operations without further ado, as explained above for the allreduce operation in GASPI (see section 4.3.7).

Either the data has to be copied to the host to perform the reduction operation or the data stays on the GPUs and a reduction kernel is started to perform the reduction. We use the first method, so the GPU input vector is copied to host memory and the result is copied back to GPU memory.

Figure 6.18 illustrates the results for the reduce and allreduce benchmarks on eight GPUs. The results show that GGAS is outperforming MPI for all vector sizes. The performance differences are clearer for the allreduce benchmark. For eight GPUs, the

Listing 6.7: boundary transfer using GGAS

```
for ( z = 1; z < ZMAX; z++ ) {
  ··· // do calculations                                              2
  p_new [ index ] = new_value;
  if (z ==ZMAX || z == 1) /* z is part of the boundary*/              4
  remote_buf [ index ] = new_value;
}                                                                     6
```

main overhead for the MPI allreduce operation is the transfer between host and GPU memory. The allreduce operation requires two of these transfers, while the reduce operation only require one data transfer on most GPUs. This additional transfer operations explain best the performance differences. For larger input vectors, GGAS also benefits from the parallel execution of the reduction operation on the GPU.

6.9. Application-level performance

In this section, the performance of different applications, using GGAS as communication layer, are evaluated and discussed.

The results of the benchmarks are compared to a hybrid version, using openMPI version 1.6.1 with Extoll and – for some benchmarks – with GPU-controlled put/get communication, using the RMA unit of the Extoll device. A comparison to Infiniband would rather result in a comparison between an ASIC and an FPGA, which is not within the scope of this work.

In contrast to the previous chapters, a wider range of benchmarks is evaluated to get a better understanding of the advantages and disadvantages of the different communication methods. A more in-depth analysis of the implementation of some of these benchmarks can be found in [80].

6.9.1. Stencil code

The first benchmark is the already known Himeno benchmark. The GGAS solution uses dynamic parallelism for intra-GPU synchronization. In contrast to the approaches using put/get communication, no special communication functions are required, but the data are directly copied in the compute kernels, using store instructions to a remote memory region. Listing 6.7 shows, in a simplified manner, how this is realized during the computation. On the remote side, the boundary values are directly read from local memory and no further copy operations are required. To synchronize the GPUs, after every iteration a barrier is required. Figure 6.19 illustrates the control flow of the stencil code using GGAS for the boundary exchange.

Due to the limitation of the size of the global address space, the problem size for this benchmark is very limited for GGAS. The complete grid has to fit into the global address space.

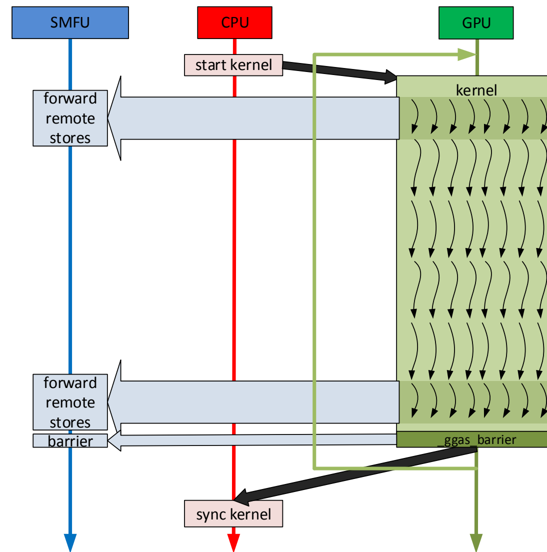


Figure 6.19.: Control flow of a stencil code, using GGAS for communication, transfer top and bottom boundaries

The first benchmark evaluates the weak scaling capabilities of GGAS and the other communication methods. The grid size per GPU stays constant while the number of GPUs increases. So the total grid size gets larger for more GPUs. Figure 6.20 shows the weak scaling results for two, four, eight, and twelve GPUs. On every GPU, the grid has as size of $128 \times 192 \times 64$ points.

The results show that GGAS is performing poorly compared to RMA put/get communication and the hybrid MPI version. The RMA version is performing best for more thqn two GPUs. Still, the MPI version does not use GPUDirect RDMA, but staged host copies on both sides. Therefore, this may be different for an MPI version using GPUDirect RDMA.

The put/get communication and MPI allow overlapping of communication and computation, which is not supported for remote store instructions. Figure 6.19 illustrates that the communication adds some overhead to all threads running on the GPU, since they have to copy the boundaries. This additional remote write operations cannot be overlapped with computations on the GPU. Although the latency of a data transfer using GGAS is smaller than the latency of the other communication methods, the missing capability of asynchronous communication slows down the performance.

In Figure 6.21, the strong scaling results for the Himeno benchmark are shown. For strong scaling, the total grid size stays constant if more GPUs are used, so the size on a single GPU decreases. The size of the grid that is distributed over the GPUs is $128 \times 128 \times 192$ points, so for twelve GPUs, on a single GPU, only $128 \times 128 \times 16$ points are calculated. For twelve GPUs, GGAS shows the best performance. However, all approaches scale not very well. For GGAS, the performance of twelve GPUs is only

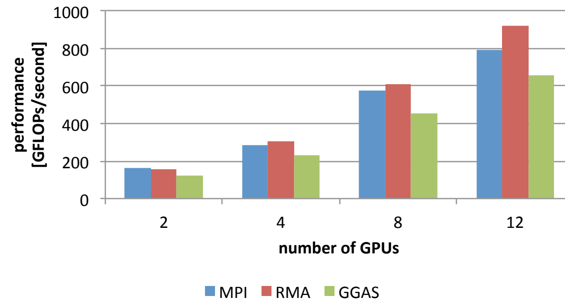


Figure 6.20.: Himeno weak scaling, ideal scaling is compared to two GPUs, problem size per GPU: $128 \times 192 \times 64$

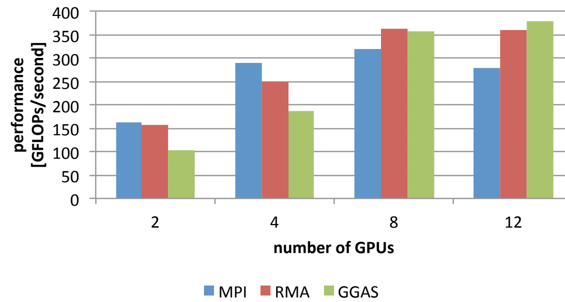


Figure 6.21.: Himeno strong scaling, problem size: $128 \times 128 \times 192$

slightly better than the performance of eight GPUs, whereas for MPI, the performance of twelve GPUs is worse than the performance for eight GPUs. The reason for the better scalability of GGAS is the lower data transfer latency. For strong scaling of this stencil benchmark, the work load per GPU becomes smaller if more GPUs are used, while the amount of transferred data stays constant. At one point, the data transfer latency is larger than the compute time, so the communication cannot completely be overlapped. Then, the communication latency is the dominating factor. However, due to the bad scalability of all approaches, the benefit of GGAS seems to be limited. This kind of problem seems to be not suitable for a communication method like GGAS.

6.9.2. Global reduction benchmark

The second benchmark is a *global reduction* benchmark. This benchmark reduces an array of input values, which is distributed over the GPUs, to a single value with a specified reduction operation. This can be used, for example, to find a global maximum or to perform a global summation of all input values. First, every GPU reduces its own field of input values, then an allreduce operation is used to determine the global reduction product. This operation is often required in numerical algorithms, for example, to determine a residual.

The MPI version starts a compute kernel to determine the local reduction result. This local result is copied to host memory and an MPI-allreduce operation is used to find the

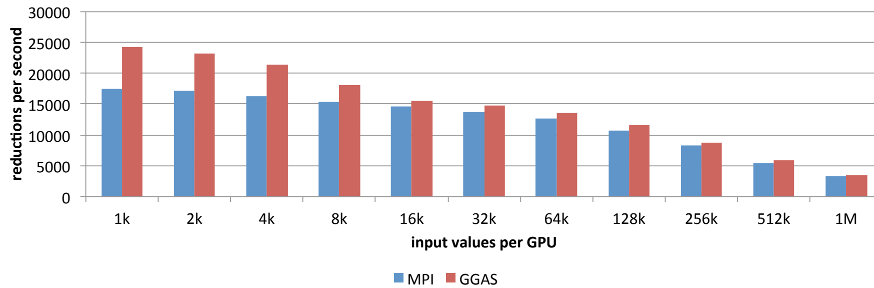


Figure 6.22.: Performance of the global reduction benchmark on eight GPUs

global reduction result. This result is then copied back to the GPUs. The GGAS version handles everything on the GPU.

In contrast to the Himeno benchmark, communication and computation cannot be overlapped. However, the amount of data that have to be exchanged is very small, because the allreduce operation is executed with only a single input value, independent of the input size.

This benchmark was not implemented for RMA put/get communication. Only one value has to be transferred for every iteration and the overhead of creating work requests would clearly outperform the latency of the data transfer in GGAS. Figure 6.22 shows the performance in *reductions per second* of the global reduction benchmark for eight GPUs.

In this benchmark, GGAS is clearly outperforming MPI, especially for smaller input vectors. For a small number of input elements, the difference between the GGAS version and the MPI version is larger, since then the data transfer is the dominating part of the distributed reduction operation. For a lager number of input values, the local reduction is dominant, so the performance difference between GGAS and MPI is less significant.

6.9.3. RandomAccess benchmark

The RandomAccess benchmark[150] was introduced with the HPC Challenge Benchmark Suite [151]. The goal of the RandomAccess benchmark is to asses the network performance. It was first implemented for MPI but without GPU support.

The benchmark operates on a large distributed (2^k) table. Each processor creates a sequence of random 64-bit integer values. The most significant values of these random numbers are used to index an entry in the table. This entry can be on a local or on a remote part of the table. The selected entry of the table is updated using a bit-wise *xor* operation between the random number and the table entry. Every process updates four times the size of the local table size. The benchmark specification allows to store at most 1,024 updates before they are applied to the table. Such an update is then called *look ahead*. The accesses are random, but every process accesses the table of every other node at least once during run time. The performance of the benchmark is measured in giga updates per second (GUPS). A detailed description of the implementation of this benchmark for GPUs, using MPI, RMA put/get and GGAS for communication, can be found in [80].

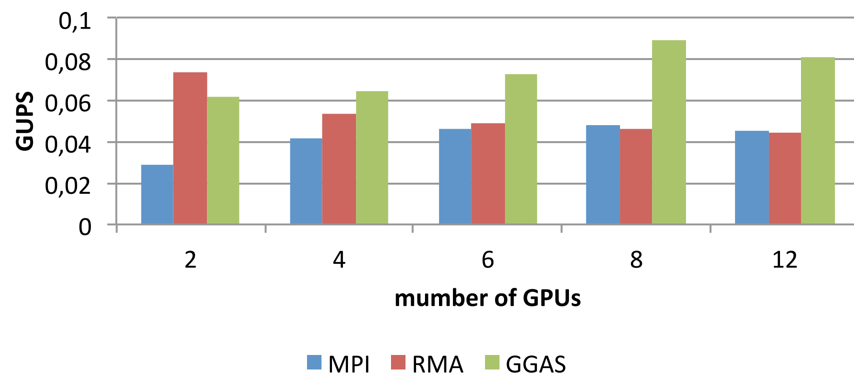


Figure 6.23.: Performance of the RandomAccess benchmark in giga updates per second (GPUS)

The MPI/GPU version creates the table in GPU memory, but the sequence of random numbers is created on the CPU. MPI is used to distribute these values along the other processes. The table updated on the GPU. The RMA version uses put-operations to distribute the random numbers, which are required to update the table. The put-operations are issued from the GPU. For every look-ahead, a block with 1,024 threads is started to update the table with parallel threads.

Due to the size of the table, for GGAS, the table cannot be located in the shared memory, as this would allow every GPU direct access to any entry in the table, regardless of where it is located. However, this approach would also require remote read accesses, which are performing worse compared to remote write accesses. Another reason why this approach should not be used is that the modification of a table entry requires an atomic read-modify-write access to a table entry. Since the shared address space does not allow atomic operations, this could result in data-hazards if two GPUs tried to access the same table entry at the same time. Therefore, a *mailbox-system* is used and every GPU writes the random number to the mailbox of the appropriate node.

Figure 6.23 shows the results of the RandomAccess benchmark for the different communication methods. The size of the input table corresponds to half of the available GPU memory.

The results show that GGAS is outperforming RMA and MPI communication for more than two GPUs. The more GPUs are used, the more significant the difference between GGAS and the other communication methods will become. The RMA version is performing worse for more than two GPUs. The reason for this is that the RandomAccess benchmark requires frequent and small communication requests, which are random and can hardly be overlapped. For this kind of workload, the overhead of creating and synchronizing work requests using the RMA unit on the Extoll device carry on more weight and the simpler GGAS interface allows more efficient communication. For eight and twelve GPUs, GGAS reaches almost twice the performance of the RMA and MPI versions of the benchmarks.

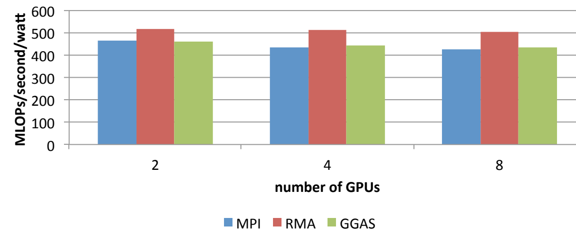


Figure 6.24.: Energy efficiency for the Himeno benchmark

6.10. Energy Efficiency

In this section, the energy efficiency of the different benchmarks for the different communication approaches is evaluated. For this, we again use the Himeno, the global reduction, and the RandomAccess benchmark.

If the communication is controlled by the GPU, the CPU can be set to sleep and an increased polling rate can be used, as described in section 5.8.5. If MPI is used for communication, a CPU thread is required to control the communication. For comparison, openMPI, version 1.6.1 , using the RMA and VELO unit of the Extoll device, is used.

6.10.1. Energy efficiency of the Himeno benchmark

Figure 6.24 shows the results for the energy efficiency of the himeno benchmark (weak scaling) for two, four and eight GPUs.

The energy efficiency is expressed in MFLOPs per second per watt. Using GPU-controlled put/get communication shows the best energy efficiency for all numbers of GPUs. However, this is obvious since the RMA version shows the best performance (see Figure 6.20) and relieves the CPU from the communication work so it can be set to sleep for a longer period of time. This leads to a 10% better energy efficiency for two GPUs and to a 15% better energy efficiency for four and eight GPUs compared to MPI and GGAS. These results are based on a very small problem size with $128 \times 192 \times 64$ points per GPU. Regarding the results in section 5.8.5, a greater increase may be reachable for larger benchmark sizes.

The MPI and the GGAS version have a very similar energy efficiency. For two GPUs, the MPI version is slightly better, for four and eight GPUs it is the GGAS version. Although the weak scaling performance of the MPI is clearly better then the GGAS version, relieving the CPU from additional work leads to a similar energy efficiency for both approaches. However put/get communication controlled from the GPU, seems to be the best solution for this kind of workload, because it allows an overlapping of communication and computation and relieves the CPU.

6.10.2. Energy efficiency of the global reduction benchmark

Figure 6.25 shows the energy efficiency, expressed in *reductions per joule*, for the global reduction benchmark on eight GPUs. The energy efficiency of the global reduction

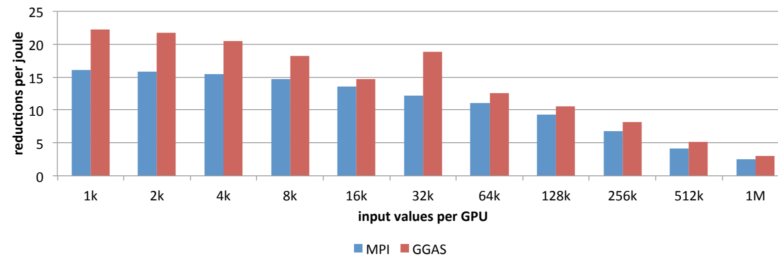


Figure 6.25.: Energy efficiency of the global reduction benchmark on eight GPUs

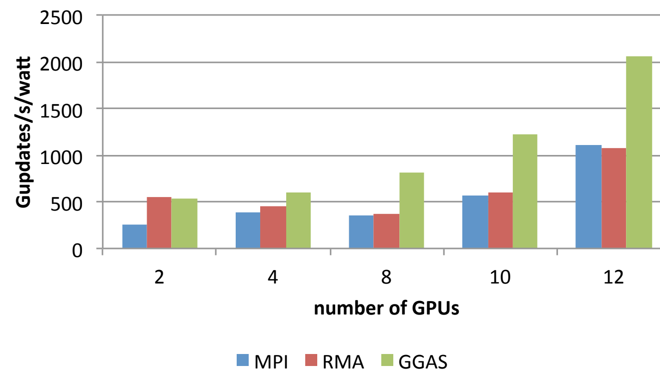


Figure 6.26.: Energy efficiency of the RandomAccess benchmark

benchmark using GGAS is significantly better than the energy efficiency of the MPI version for all input sizes. For a small number of elements, this difference is clear. On the whole, the energy efficiency results correspond to the performance results of the reduction benchmark. However, for larger input sizes, the energy efficiency of GGAS is still clearly better than the energy efficiency of the MPI version, which is not the case for the performance. The reason for this is again the relieve of the CPU.

6.10.3. Energy efficiency of the RandomAccess benchmark

Figure 6.26 shows the energy efficiency of the randomAccess benchmark, expressed in updates per second per watt (or updates per joule). The results for the energy efficiency also correspond to the performance results of the RandomAccess benchmark. For two GPUs, RMA put/get communication has the best energy efficiency, for more GPUs, the GGAS version is better. It is notable that for eight and more GPUs, the energy efficiency of the GGAS version is more than twice as good as the energy efficiency of the MPI and the RMA version. This is caused by the better performance and the lower CPU utilization using GGAS for communication.

6.11. Summary

In this section, a hardware-supported global address space for thread collective communication between distributed GPUs was introduced. The performance of the micro-benchmarks showed that by using this communication model, the latency for GPU-to-GPU data transfer can be reduced to 3 μs for small data transfer sizes. By using remote write operations, the PCIe issue for non-posted device-to-device transfer can be avoided, resulting in a peak bandwidth of 800 MB/s, using an FPGA.

We showed that this communication model is best suitable for small data transfers and blocking collective operations like a barrier or an allreduce operation. However, the main drawback of this communication method is that it currently does not support asynchronous communication. Therefore, communication and computation cannot be overlapped. Furthermore, a lot of GPU-threads are required to transfer a large amount of data which adds a lot of overhead to the GPU. For applications that allow a good overlapping of communication and computation, for example, the Himeno benchmark, other communication methods like GPU-conrtolled put/get communication are preferable.

Still, for applications which require small, non-blocking, and irregular data transfers, a global address space is a very efficient communication method. The performance results of the global reduction and the RandomAccess clearly show the benefits of the GGAS communication model.

The results of the energy efficiency for the different benchmarks and communication methods show that by relieving the CPU and by using an efficient, application-specific GPU-controlled communication method, GPU-controlled communication has a better energy-efficiency than a hybrid model using the CPU for communication.

7. Conclusion and Future Outlook

The goal of this work was to implement, analyze, and evaluate different communication methods for distributed GPUs. The main contributions of this work are the integration of GPUs into a host-controlled, one-sided communication library and the implementation of GPU-controlled put/get and load/store communication libraries for distributed GPUs. Furthermore, the strengths and weaknesses of these communication methods were analyzed within the scope of performance and energy efficiency. All implementations and modifications used in this work are done in the software level, in kernel-space, in user-space, and on the GPU. So far, no changes on the used hardware were required. However, the results of this work give hints and tips to hardware developers about the requirements for communication-centric, GPU-accelerated systems.

In the first part of this work, the contribution was the implementation of the support for GPUDirect RDMA for the RMA function unit of the Extoll device. This technique allows a direct data transfer between two GPUs without copies in host memory. The support for GPUDirect RDMA forms the basis for the communication methods described in the subsequent chapters. The performance results show that, although GPUDirect RDMA brings a lot of benefits for small- and medium-sized data transfers, for larger data transfers, the performance is limited due to PCIe issues since non-posted peer-to-peer transfers are not well supported on current chipsets.

A further contribution of this work is the integration of GPUs into the host-controlled communication specification GASPI and its implementation GPI-2. So far, GPI-2 only supports Infiniband, therefore the GPU integration is also based on Infiniband. The integration of GPUs to the GASPI specification requires only minor changes to the API. However, to reach optimal performance, some additional, GPU-related functions were added. The performance results show that GPI-2 for GPUs shows a significantly better performance than CUDA-aware MPI which is the current state-of-the-art.

The rest of this work concentrates more on GPU-controlled communication, a research area that has hardly been explored so far. As far as it is known, this is the first work in which the GPU is enabled to source and synchronize communication requests directly to the hardware. In all preliminary work, a CPU thread was always required to control the NIC.

The first individual contribution here was the implementation and evaluation of the Infiniband verbs interface for GPUs. Due to the low performance compared to host-controlled communication, the bottlenecks of this method were analyzed. The results show that a communication protocol like Infiniband verbs causes too much overhead on the GPU. The creation of work requests for Infiniband devices takes on the GPU more than 100 times longer than on the CPU. The main reason for this is the Infiniband verbs protocol, which requires complicated error checking and big-to-little endian

translation. This is a sequential workload and cannot be parallelized for GPUs. For GPUs, a simpler and more effective communication framework is required which should best allow thread-collaborative communication. Another important factor for efficient GPU-controlled communication is the amount od accesses to the host memory over the PCIe-bus. For a good performance, these accesses should be reduced to a minimum. A comparison with a GPU-interface for the RMA unit of the Extoll device showed that a simplified communication interface could lead to better performance.

The problems that are related with to the GPU-programming model and put/get-communication were analyzed and possible solutions were introduced. The main problem is that communication requires intra-GPU synchronization. This work showed that *dynamic parallelism* provides a good solution for this. The performance results show that GPU-controlled put/get communication has a lower performance than CPU-controlled communication, using GPI-2 as communication layer. Still, the power and energy analysis shows that relieving the CPU from the communication work results in better performance per watt for GPU-controlled communication.

The last but not least individual contribution of this work is the implementation of a hardware-supported Global GPU Address Space (GGAS) for distributed GPUs. This global address space uses the shared memory function unit (SMFU) of the Extoll device. It allows fast and thread-collective communication on the GPU with simple load and store instructions. The latency and the bandwidth of this communication method surpass the performance of all other communication methods. However, the main bottleneck is that the communication cannot be overlapped and for larger data transfer sizes, a huge overhead on the GPU is created. Therefore, this communication method is only suitable for small and irregular communication patterns and applications hardly having room for overlapping communication and computation. Here, a GPU-controlled put/get communication model appears to be more efficient.

In conclusion, currently no ideal communication method for all sizes of data transfers and communication patterns for distributed GPUs exist. However, the power and energy analyses show that GPU-controlled communication always shows a better energy efficiency than CPU-controlled communication and energy efficiency will become more important for future high-performance systems.

By using commodity hardware like Infiniband, the performance per watt increases by approximalty 10%; with more specialized hardware, even better results can be achieved. Future energy-efficient systems require specialized processors like GPUs to improve the performance-per-watt metric. Communication models and methods have to match these specialized architectures as, otherwise, gains in energy efficiency can be diminished or even neutralized.

Currently, using these communication methods requires a lot of efford from the programmer and a lot of work is required to optimize an application for the best communication method. In future systems, this must be more simplified for the user. In a next step, a transparent communication interface is needed that allows a programmer to source and synchronize communication requests transparently from the underlying communication hardware.

Currently, one of the main bottlenecks for GPU communication is the PCIe-bus. One reason for this is the insufficient support for non-posted PCIe peer-to-peer data transfers. But, also the usual bandwidth of the PCIe bus is far bellow the memory bandwidth of the GPU memory. However, future GPUs may no longer be only connected with the PCIe-bus. Instead, new technologies like NVLINK will overcome some of the limitations of the PCIe-bus. Still, to allow efficient communication between GPUs across the borders of a single node, the interconnection networks for inter-node communication must keep up with these developments.

A. Power Measurement

In this work, a software bases approach is used to measure the power. There are two possibilities to measure the power consumption of a running application: internal and external power measurements. [152].

External power measurements use external devices to measure the power of a running application. The easiest type of equipment is a power meters for outlets, which measures the power of the complete system. However, this method does not allow a break down to the individual components like CPU, DRAM or GPU.

A fine grain-measurement is enabled by hooking wires into power supplies for the individual components, as for example described in [153]. However, this approach usually requires extensive installation costs and is often not practical or obstructive.

Another disadvantage of this method is that the temporal resolution is often too coarse to make precise conclusions to the actual power consumption of an application.

Therefore, in this work *internal* power measurement is used. This approach uses the internal hardware support to measure energy and power consumption. This approach is less accurate then external power meters, but allows an easy break down to the individual components and a fine grain resolution in time.

Nvidia offers utilities to measure power accurately using the Nvidia Management Library (NVML). The power is is estimated with milliwatt precision within a range of ± 5.

For Intel CPUs, the Running Average Power Limit (RAPL) registers[154] are used. This registers report power consumption of CPU and DRAM.

Both techniques use power modeling to estimate the current power consumption of the respective hardware components, and are widely accepted as accurate [152].

This approach leaves other components like the network device uncovered. However, communication patterns to not differ for the different communication approaches used here, and network power consumption is typically independent of the actual traffic, as it is dominated by (de-)serializers that employ serial link coding and embedded clocking.

B. Acronyms

API Application Programming Interface

APU Accelerated Processing Unit

ATU Address Translation Unit

AVX Advanced Vector Extensions

BAR Base Address Register

COF Co Array Fortran

CPU Central Processing Unite

CQ Completion Queue

CUDA Compute Unified Device Architecture

DMA Direct Memory Access

GASNet Global-Address Space Networking

GASPI Global Address Space Programming Interface

GDDR Graphical Double Rate DRAM

GGAS Global GPU Address Space

GPGPU General Purpose Graphic Processing Unit

GPI Global Address Space Programming Interface

GPU Graphic Processing Unit

GPGPU General Purpose

HCA Host Channel adapter

HT Hyper Transport

I/O Input/Output

IB-verbs Infiniband Verbs

IN Interconnection Network

MIC Many Integrated Core

MIMD Multi Instruction Multiple Data Stream

MMIO Memory Mapped I/O

MPI Message Passing Interface

NIC Network Interface Card

NI Network Interface

NVML Nvidia Management library

OpenCL Open Computing Language

PCIe Peripheral Component Interconnect Express

PGAS Partitioned Global Address Space

QPI QuickPath Interconnect

QP Queue Pair

RAM Random Access Memory

RDMA Direct Remote Memory Access

RMA Remote Memory Access

SHMEM Symmetric Hierarchical Memory

SIMD Single Instruction Multiple Data Stream

SM Streaming Multiprocessor

SMFU Shared Memory Function Unit

SPMD Single Programm Multiple Data

SSE Streaming SIMD Extensions

UPC Universal Parallel C

VELO Virtual Engine for Low Overhead

UVA Unified Virtual Address Space

hUMA heterogeneous Unified Memory Access

List of Figures

1.1. Floating points per second, GPUs vs.Processors 1

2.1. Simplified structure of a GPU accelerated cluster 5
2.2. Parallel architectures 6
2.3. MIMD architectures 6
2.4. Communication models in distributed memory systems 8
2.5. Two-sided communication 9
2.6. One-sided communication paradigma 9
2.7. Remote load stores 9
2.8. Explicit und implicit synchronization 10
2.9. Synchronous and asynchronous communication 11
2.10. Performance parameter for communication 13
2.11. General-active-target-synchronization in MPI 15
2.12. PGAS Modell 17
2.13. OpenSmem memory model 18
2.14. Difference between a GPU and a CPU 19
2.15. Block diagram of a GK100 GPU 21
2.16. Block diagram of a streaming multi processor 21
2.17. GPU memory write bandwidth 22
2.18. Simplified block diagram of the Intel Xeon Phi 23
2.19. Simplified block diagram of a AMD Liano APU 24
2.20. A 2-dimensional GPU-thread grid with 2-dimensional blocks 27
2.21. CUDA memory Modell 28
2.22. Design space of network interfaces 30
2.23. PCIe topology 31
2.24. QPI interconnect 32
2.25. HT interconnect 32
2.26. Two register interface 33
2.27. Infiniband subnet topology 35
2.28. Work request processing in Infiniband 36
2.29. Simplified view of the Infiniband software stack 39
2.30. The principle block diagram of the Extoll NIC 40
2.31. Extoll software stack 43
2.32. Communication Control for inter-GPU communication 44
2.33. Workflow, communication is controlled by CPU 45
2.34. Workflow, communication is controlled by GPU 45

2.35. Communication methods for distributed GPUs 46

3.1. Data transfer methods between distributed GPUs 48
3.2. Sequential and pipelined data transfer 48
3.3. Data transfer without GPUDirect 1.0 49
3.4. Data transfer with GPUDirect 1.0 49
3.5. Data transfer with GPUDirect peer-to-peer 50
3.6. Unified virtual address space (UVA) 50
3.7. Data transfer with GPUDirect RDMA 51
3.8. Mellanox GPUDirect RDMA support 53
3.9. GPUDirect RDMA with MMIO 55
3.10. Registration of MMIO addresses 56
3.11. GPUDirect RDMA performance of Infiniband 57
3.12. GPUDirect RDMA performance 58
3.13. Bandwidth using different transfer directions 59
3.14. Bandwidth on Ivy Bridge 59
3.15. Bandwidth over the QPI link 60

4.1. GASPI memory model 68
4.2. Passive communication in GASPI 69
4.3. Weak synchronization in GASPI 71
4.4. GASPI support for GPUs 72
4.5. Data flow for gaspi_gpu_write 77
4.6. Staged data transfers on Sandy Bridge, K20 GPUs 78
4.7. Staged data transfers on Ivy Bridge, K40 GPUs 78
4.8. Bandwidth, GPI-2 vs. Mvapich2 80
4.9. Latency, GPI-2 vs. Mvapich2 81
4.10. Overhead, GPI-2 vs. Mvapich2 82
4.11. GASPI total runtime of the allreduce operation 83
4.12. GPU-allreduce, relative to host-allreduce operation 83
4.13. Synchronization methods for GPU/host synchronization 84
4.14. Stencil environments 85
4.15. Boarder exchange for a 2-D stencil code 86
4.16. Control flow of a hybrid stencil application 87
4.17. Strong scaling of the Himeno benchmark using GPI-2 and MPI 88

5.1. Sample work request for Infiniband 92
5.2. Sample completion element for Infiniband 93
5.3. Structure of an RMA descriptor for remote put/get commands 93
5.4. Structure of a RMA notification for put/get commands 94
5.5. Software stack to allow a GPU to communicate 95
5.6. Steps to create an communication context on the GPU 96
5.7. Location of communication resources 97
5.8. Access to remote resources 98

5.9. PCIe accesses with queue buffers on host 102
5.10. PCIe accesses with queue buffers on GPU 103
5.11. Latency of GPU-controlled Infiniband communication 105
5.12. Bandwidth of GPU-controlled Infiniband communication 106
5.13. Latency of GPU-controlled Infiniband atomic operations 107
5.14. Message rate of GPU-controlled Infiniband communication 108
5.15. Message rate of GPU-controlled Infiniband communication 2 109
5.16. Performance counter for Infiniband communication 113
5.17. Latency of GPU-controlled communication, using Extoll RMA 117
5.18. Bandwidth of GPU-controlled communication, using Extoll RMA . . . 118
5.19. 3-D stencil thread blocks 124
5.20. 2-D thread block 124
5.21. Control flow of a stencil application, in kernel synchronization 125
5.22. Control flow of a stencil application, stream synchronization 127
5.23. Control flow of a stencil application, device synchronization 128
5.24. CPU overhead dynamic parallelism 130
5.25. Energy efficiency for a single GPU 131
5.26. Power over time without communication 131
5.27. Performance of the Himeno benchmark 132
5.28. Communication overhead of the Himeno benchmark 133
5.29. Energy efficiency of the Himeno benchmark 134
5.30. Power over time for the Himeno benchmark 135

6.1. Simplified system and user view of a GGAS cluster 140
6.2. SMFU Address Space Layout 144
6.3. GGAS address space layout 145
6.4. Simplified way of a store request using GGAS 146
6.5. GGAS software 148
6.6. Initialization of the global GPU address space 149
6.7. Latency for communication using GGAS 152
6.8. Bandwidth for communication using GGAS 153
6.9. Coalescing effects for remote memory accesses 154
6.10. Performance of the GGAS-barrier 157
6.11. Gathering 159
6.12. Broadcast 159
6.13. All-gather 159
6.14. All-to-all 159
6.15. Binomial tree for eight nodes 160
6.16. Results of the reduce operation on eight GPUs 161
6.17. Results of the allreduce operation on eight GPUs 161
6.18. Results of the reduce and allreduce benchmarks 162
6.19. Control flow of a stencil application, using GGAS 164
6.20. Himeno weak scaling 165
6.21. Himeno strong scaling 165

6.22. Performance of the global reduction benchmark 166
6.23. Performance of the RandomAccess Benchmark 167
6.24. Energy efficiency for the Himeno benchmark 168
6.25. Energy efficiency of the global reduction benchmark 169
6.26. Energy efficiency of the RandomAccess benchmark 169

List of Tables

2.1. CUDA extensions for function calling 26
2.2. Possible configurations for notifications 42

3.1. Maximal bandwidth in MB/s using GPUDirect RDMA 59

4.1. Synchronization time for different host-GPU synchronization methods . 84
4.2. Himeno problem size 87

5.1. List o Infiniband resources 99
5.2. Latency of Infiniband communication control functions 110
5.3. Performance counter Infiniband 110
5.4. Latency of simplified communication requests on the GPU 112
5.5. Performance counter for Extoll RMA 120
5.6. Properties of the Himeno benchmark 132

6.1. Incoming requests on SMFU 154

Listings

2.1. Example for memory consistency 7
2.2. CUDA kernel launch 27

3.1. Stepwise creation of the page table for MMIO addresses 56

4.1. Using MPI for data transfer between GPUs, no CUDA-aware MPI . . . 64
4.2. Using CUDA-aware MPI 64
4.3. GASPI code example 73
4.4. GASPI for GPUs code example 73
4.5. Post-work-wait loop for GASPI 81
4.6. Post-work-wait loop for MPI 81

5.1. Creation of a QP for the GPU, host function 101
5.2. Creation of a QP for the GPU, GPU kernel 101
5.3. Creating an RMA connection 115
5.4. RMA port for a GPU 116
5.5. Creating a descriptor on the GPU 116
5.6. Stencil code with communication in compute kernel 126
5.7. Communication kernel 127
5.8. Set CPU to sleep while waiting for completion 130
5.9. Increased polling with GASPI 134

6.1. Simple remote write example with GGAS 150
6.2. Code example using GGAS, pingpong Benchmark 151
6.3. Code example for remote read using GGAS 151
6.4. Barrier using dynamic parallelism 156
6.5. Barrier using a barrier kernel 156
6.6. Reduce with remote read 158
6.7. boundary transfer using GGAS 163

Danksagung

An dieser Stelle möchte ich mich bei allen Menschen bedanken, die mich beim Erstellen dieser Arbeit Unterstützt haben.

An erster Stelle möchte ich mich bei Professor Dr. Ulrich Brüning bedanken, der mich bei dieser Dissertation betreut hat und mir mit fachlichen und persönlichen Rat zur Seite stand und mich sicher durch die Zeit der Promotion geleitet hat. Ein besonderer Dank gilt auch Professor Dr. Holger Fröning, der durch Ratschläge und fachliche Diskussionen viel zur Steigerung der Qualität dieser Arbeit beigetragen hat und mir gerade in der Anfangszeit bei Weiterentwicklung meiner wissenschaftlichen Fähigkeiten geholfen hat. Ein weiterer Dank geht an Benjamin Klenk, der mit seinen Arbeiten auch zum gelingen dieser Promotion beigetragen hat. Da ich an dieser Stelle nicht alle einzeln nennen kann, möchte ich dem gesamten Lehrstuhl für Rechnerarchitektur der Universität Heidelberg danken.

Weiter Dank geht an Dr. Franz-Joseph Pfreundt und Dr. Carsten Lojewski vom Fraunhofer Institut für Wirtschafts- und Technomathematik (ITWM), die mir diese Promotion ermöglicht haben und mir in meiner Zeit am ITWM immer mit Rat und Tat zu Seite standen. Mein Dank gilt der Fraunhofer Gesellschaft, die mir diese Promotion finanziell ermöglicht hat und mir die Möglichkeit gab, an internationalen Konferenzen teilzunehmen. Ich bedanke mich bei dem gesamten Team vom Competence Center High Performance Computing am ITWM für die gute Aufnahme und herzliche Unterstützung. Ganz besonderen Dank möchte ich an dieser Stelle Frauke Santa Cruz aussprechen, die mir bei vielen organisatorische Aufgaben zur Seite stand und mir auch viele persönliche Ratschläge gegeben hat.

Weiterhin möchte ich mich bei Nvidia, der Extoll GmbH und bei Xilinx Inc. für die Unterstützung durch Hardware und inhaltliche Anregungen bedanken.

Ein weiterer Dank gilt auch den Reviewern meiner Veröffentlichungen, die mit ihrer teilweise konstruktiven Kritik zur Steigerung der fachlichen Qualität dieser Arbeit beigetragen haben. Ferner bedanke ich mich bei Hemant Shulka für die Betretung in meiner Zeit in Berkeley. Sean danke ich für das gewissenhafte Korrekturlesen dieser Arbeit.

Großer Dank gilt meiner Familie, die mir durch ihre finanzielle und vor allem emotionale Unterstützung den Weg in die Wissenschaft erst ermöglicht hat. Bei meinem Freundeskreis möchte ich mich ebenfalls bedanken, da sie mir gelegentlich zum nötigen Abstand verholfen haben. Ganz besonders möchte ich mich bei Philipp bedanken, der mir nicht nur in den schweren Stunden die Stabilität gegeben hat, dieser Promotion erfolgreich zu schaffen.

Bibliography

[1] H. Sutter and J. Larus, "Software and the concurrency revolution", *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[2] H. Sutter, "The free lunch is over: a fundamental turn toward concurrency in software", *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[3] *CUDA C programming guide*, version 6.5, Nvidia, Oct. 19, 2014. [Online]. Available: `http://docs.nvidia.com/cuda/index.html`.

[4] *The OpenCL specification*, Khronos OpenCL Working Group, Nov. 2013. [Online]. Available: `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[5] *The OpenACC application programming interface*, OpenACC Working Group, CRAY Inc, The Portland Group Inc, and NVIDIA, Aug. 2013. [Online]. Available: `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`.

[6] (Aug. 30, 2014). Green500, [Online]. Available: `http://www.green500.org/`.

[7] D. Jacobsen, J. Thibault, and I. Senocak, "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters", in *Proceedings of 48th AIAA Aerospace Sciences Meeting and Exhibit*, vol. 16, Orlando, Florida, USA, 2010, pp. 6151–6166.

[8] E. Phillips and M. Fatica, "Implementing the Himeno benchmark with CUDA on GPU clusters", in *Proceedings of International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, Atlanta, Georgia, USA, 2010, pp. 1–10.

[9] W. Ma, S. Krishnamoorthy, O. Villay, and K. Kowalski, "Acceleration of streamed tensor contraction expressions on GPGPU-based clusters", in *International Conference on Cluster Computing (CLUSTER)*, IEEE, Heraklion, Crete, Greece, 2010, pp. 207–216.

[10] (Sep. 3, 2014). Top 500 website, [Online]. Available: `http://www.top500.org/`.

[11] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges", in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR)*, Berkeley, California, USA: Springer, 2011, pp. 1–25.

[12] D. Sima, T. Fountain, and P. Kacsuk, *Advanced Computer Architectures A Design Space Approach*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[13] M. Flynn, "Exploring memory consistency for massivelythreaded throughput-oriented processors", *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[14] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs", *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.

[15] B. A. Hechtman and D. J. Sorin, "Exploring memory consistency for massively-threaded throughput-oriented processors", in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ACM, Tel-Aviv, Israel, 2013, pp. 201–212.

[16] S. L. Scott, "Synchronization and communication in the T3E multiprocessor", *ACM SIGPLAN Notices*, vol. 31, no. 9, pp. 26–36, 1996.

[17] H. Fröning and H. Litz, "Efficient hardware support for the partitioned global address space", in *Proccedings ofInternational Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, IEEE, Atlanta, Georgia, USA, 2010, pp. 1–6.

[18] R. Machado, C. Lojewski, S. Abreu, and F.-J. Pfreundt, "Unbalanced tree search on a manycore system using the GPI programming model", *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 229–236, 2011.

[19] R. Machado, S. Abreu, and D. Diaz, "Parallel local search: experiments with a PGAS-based programming model", in *International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS)*, Budapest, Hungary, 2012, pp. 101–116.

[20] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken, *LogP: Towards a realistic model of parallel computation*, 7. ACM, 1993, vol. 28.

[21] D. Grünewald, "BQCD with GPI: a case study", in *International Conference on High Performance Computing and Simulation (HPCS)*, IEEE, Madrid, Spain, 2012, pp. 388–394.

[22] *MPI: A Message-Passing Interface Standard*, Knoxville, TN, USA, May 1994.

[23] W. D. Gropp, "Learning from the success of MPI", in *High Performance Computing (HiPC)*, ser. Lecture Notes in Computer Science, vol. 2228, 2001, pp. 81–92.

[24] R. Rabenseifner, "Optimization of collective reduction operations", in *In Proceedings of 4th International Conference, Computational Science (ICCS)*, Kraków, Poland: Springer, 2004, pp. 1–9.

[25] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH", *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[26] W. D. Gropp, *MPI: the complete reference. the MPI-2 extensions*, MIT Press, 2003.

[27] (Oct. 1, 2014). General active target synchronization, [Online]. Available: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node240.htm.

[28] R. Thakur, W. D. Gropp, and B. Toonen, "Minimizing synchronization overhead in the implementation of MPI one-sided communication", in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 3241, Springer, 2004, pp. 57–67.

[29] D. Ashton, W. Gropp, R. Thakur, and B. Toonen, "The CH3 design for a simple implementation of ADI-3 for MPICH-2 with a TCP-based implementation", Tech. Rep., May 2004.

[30] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "An implementation and evaluation of the MPI 3.0 one-sided communication interface", *Preprint, Argonne National Labs*, 2013.

[31] *UPC language specifications v1.2*, UPC Consortium et al., 2005. [Online]. Available: http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf.

[32] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming", in *Proccedings of ACM SIGPLAN Fortran Forum*, ACM, vol. 17, New York City, New York, USA, 1998, pp. 1–31.

[33] A. Aiken, P. Colella, D. Gay, S. Graham, P. Hilfinger, A. Krishnamurthy, B. Liblit, C. Miyamoto, G. Pike, L. Semenzato, *et al.*, "Titanium: a high-performance java dialect", *Concurrency: Practice and Experience*, vol. 10, no. 11–13, pp. 825–836, 1998.

[34] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing", *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[35] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, *et al.*, *The fortress language specification*, Sun Microsystems, 2005, p. 140.

[36] *Chapel language specification version 0.95*, Cray Inc, Apr. 2014. [Online]. Available: http://chapel.cray.com/spec/spec-0.95.pdf.

[37] M. Weiland, "Chapel, fortress and x10: novel languages for hpc", The University of Edinburgh, Tech. Rep., 2007.

[38] D. Bonachea, *GASNet specification, v1.* University of California, Berkeley, 2002. [Online]. Available: http://gasnet.lbl.gov/CSD-02-1207.pdf.

[39] (Oct. 6, 2014). Berkeley unified parallel C, Lawrence Berkeley National Labs and UC Berkeley, [Online]. Available: http://upc.lbl.gov/.

[40] J. A. Kuehn, B. Chapman, A. R. Curtis, R. Mauricio, S. Pophale, R. Nanjegowda, A. Banerjee, K. Feind, S. W. Poole, and L. Smith, *OpenSHMEM application programming interface, v1.0 final*, Oak Ridge National Laboratory (ORNL), 2012. [Online]. Available: http://info.ornl.gov/sites/publications/files/Pub38857.pdf.

[41] R. E. Kessler and J. L. Schwarzmeier, “CRAY T3D: a new dimension for cray research”, in *Proccedings ofInternational Computer Conference, Compcon Spring’93, Digest of Papers*, IEEE, San Francisco, California, USA, 1993, pp. 176–182.

[42] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, “Introducing OpenSHMEM: SHMEM for the PGAS community”, in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS)*, ACM, New York, New York, USA, 2010, 2:1–2:3.

[43] *Specification of a PGAS API for communication v. 1.0*, GASPI: Global Address Space Programming Interface, Jun. 2013. [Online]. Available: `http://www.gaspi.de/fileadmin/GASPI/pdf/GASPI%5C_Standard%5C_Draft.pdf`.

[44] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[45] *NVIDIA’s next generation CUDA compute architecture: Kepler GK110*, whitepaper, 2013. [Online]. Available: `http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf`.

[46] G. Chrysos. (Nov. 11, 2014). Intel Xeon Phi coprocessor - the architecture, Intel, [Online]. Available: `https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner`.

[47] A. Branover, D. Foley, and M. Steinman, “AMD’s llano fusion APU”, *Micro*, vol. 32, no. 2, pp. 28–37, Mar. 2012.

[48] M. Daga, A. Aji, and W.-C. Feng, “On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing”, in *Proceedings of Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, Knoxville, Tennessee, USA, Jul. 2011, pp. 141–149.

[49] Wikipedia. (Sep. 23, 2014). Playstation 4 , wikipedia, the free encyclopedia, [Online]. Available: `http://en.wikipedia.org/w/index.php?title=PlayStation_4&oldid=626004784`.

[50] ——, (Sep. 23, 2014). Xbox one, wikipedia, the free encyclopedia, [Online]. Available: `http://en.wikipedia.org/w/index.php?title=Xbox_One&oldid=626747406`.

[51] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, “Sparse matrix solvers on the GPU: conjugate gradients and multigrid”, *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 917–924, 2003.

[52] J. Krüger and R. Westermann, “Linear algebra operators for GPU implementation of numerical algorithms”, *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 908–916, 2003.

[53] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "GPGPU: general-purpose computation on graphics hardware", in *Proceedings of the Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Tampa, Florida, USA: ACM/IEEE, 2006.

[54] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture", in *Proceedings of the 6th international symposium on Memory management, ISMM*, vol. 7, Montreal, Canada, 2007, pp. 103–104.

[55] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA", *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008, ISSN: 1542-7730.

[56] M. J. Atallah, *Algorithms and theory of computation handbook*, 1st. Boca Raton, FL, USA: CRC Press, Inc., 1998, ISBN: 0849326494.

[57] (Apr. 26, 2014). Nvidia kepler tuning guide, Nvidia, [Online]. Available: `http://docs.nvidia.com/cuda/kepler-tuning-guide/#l1-cache`.

[58] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL", in *Proccedings ofInternational Conference on Parallel Processing (ICPP), 2011*, IEEE, Taipei, Taiwan, 2011, pp. 216–225.

[59] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of CUDA and OpenCL", Tech. Rep., 2010.

[60] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC—first experiences with real-world applications", in *Proccedings ofEuro-Par – Parallel Processing*, Rhodes Islands, Greece: Springer, 2012, pp. 859–870.

[61] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski, "OpenMP for accelerators", in *OpenMP in the Petascale Era*, Springer, 2011, pp. 108–121.

[62] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "CUDA vs OpenACC: performance case studies with kernel benchmarks and a memory-bound cfd application", in *Proccedings of13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2013*, IEEE, Delft, Netherlands, 2013, pp. 136–143.

[63] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, ser. The Morgan Kaufmann Series in Computer Architecture and Design Series. Morgan Kaufmann Publishers, 2004.

[64] T.-y. Feng, "A survey of interconnection networks", *Computer*, vol. 14, no. 12, pp. 12–27, Dec. 1981, ISSN: 0018-9162.

[65] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm, "Analysis of the memory registration process in the mellanox infiniband software stack", in *Proceedings ofEuro-Par – Parallel Processing*, Dresden, Germany: Springer, 2006, pp. 124–133.

[66] T. Woodall, G. Shipman, G. Bosilca, R. Graham, and A. Maccabe, "High performance RDMA protocols in HPC", in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, B. Mohr, J. Tröff, J. Worringen, and J. Dongarra, Eds., Springer Berlin Heidelberg, 2006, pp. 76–85.

[67] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over infiniband", *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[68] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "RDMA read based rendezvous protocol for MPI over infiniband: design alternatives and benefits", in *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming(PPoPP)*, ACM, New York City, New York, USA, 2006, pp. 32–39.

[69] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda, "Host-assisted zero-copy remote memory access communication on infiniband", in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, Santa Fe, New Mexico, USA, 2004, pp. 31–36.

[70] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down cache: a virtual memory management technique for zero-copy communication", in *Proceedings of the First Merged International Conference and Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, IEEE, Orlando, Florida, USA, 1998, pp. 308–314.

[71] F. Mietke, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm, "Reducing the impact of memory registration in infiniband", pp. 1–13, Nov. 2005.

[72] G. F. Pfister, "An introduction to the infiniband architecture", *High Performance Mass Storage and Parallel I/O*, vol. 42, pp. 617–632, 2001.

[73] M. Koop, S. Sur, Q. Gao, and D. Panda, "High performance MPI design using unreliable datagram for ultra-scale infiniband clusters", in *Proceedings of the 21st annual international conference on Supercomputing (ISC)*, ACM, Dresden, Germany, 2007, pp. 180–189.

[74] M. J. Koop, S. Sur, and D. K. Panda, "Zero-copy protocol for MPI using infiniband unreliable datagram", in *Proceedings of International Conference on Cluster Computing (CLUSTER)*, IEEE, Austin, Texas, USA, 2007, pp. 179–186.

[75] "Mellanox OFED for linux user's manual rev. 1.5.1", Mellanox Technologies, Tech. Rep., 2010. [Online]. Available: `http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_user_manual_1_5_1.pdf`.

[76] M. B. Nüssle, "Acceleration of the hardware-software interface of a communication device for parallel systems", PhD thesis, Universität Mannheim, 2009.

[77] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning, "An open-source hypertransport core", *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 1, no. 3, 14:1–14:21, 2008.

[78] B. Geib, "Hardware support for efficient packet processing", PhD thesis, Universität Mannheim, 2012.

[79] N. Burkhardt, *Fast hardware barrier synchronisation for a reliable interconnection network*, Diploma Thesis, 2007.

[80] B. Klenk, "Comparing different communication paradigms for data-parallel processors", Master Thesis, Institute of Computer Engineering, Department of Physics and Astronomy, University of Heidelberg, 2013.

[81] H. Litz, H. Fröning, M. Nuessle, and U. Brüning, "Velo: a novel communication engine for ultra-low latency message transfers", in *Proceedings of the 37th international conference on parallel processing (ICPP)*, Portland, Oregon, USA, 2008, pp. 238–245.

[82] M. Nuessle, M. Scherer, and U. Bruening, "A resource optimized remote-memory-access architecture for low-latency communication", in *Proceedings of the 38th Conference on Parallel Processing (ICPP)*, Vienna, Austria, 2009, pp. 220–227.

[83] C. Leber, "Efficient hardware for low latency application", PhD thesis, Universität Mannheim, 2012.

[84] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing", in *Proceedings of the Conference on High Performance Computing, Networking and Storage , SC*, Pittsburg, Pennsylvania, USA: IEEE, ACM, 2004, p. 47.

[85] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU clusters", in *Proceedings of International Symposium on Parallel & Distributed Processing Symposium (IPDPS),*, IEEE, Anchorage, Alaska, USA, 2011, pp. 1068–1079.

[86] J. C. Thibault and I. Senocak, "Cuda implementation of a navier-stokes solver on multi-gpu desktop platforms for incompressible flows", in *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, Orlando, Florida, USA, 2009, pp. 2009–758.

[87] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of mellanox/nvidia GPUDirect over infiniband—a new model for GPU to GPU communications", *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 267–273, 2011.

[88] R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, P. Paolucci, R. Petronzio, D. Rossetti, A. Salamon, G. Salina, *et al.*, "Apenet+: a 3d toroidal network enabling petaflops scale lattice qcd simulations on commodity clusters", *Computing Research Repository (CoRR)*, vol. abs/1012.0253, 2010.

[89] R. Ammendola, M. Bernaschi, A. Biagioni, M. Bisson, M. Fatica, O. Frezza, F. Lo Cicero, A. Lonardo, E. Mastrostefano, and P. S. Paolucci, "GPU peer-to-peer techniques applied to a cluster interconnect", in *Proceedings of 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, IEEE, Boston, Massachusetts, USA, 2013, pp. 806–815.

[90] R. Ammendola, A. Biagionil, O. Frezza, F. Cicero, A. Lonardo, P. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, “Design and implementation of a modular, low latency, fault-aware, fpga-based network interface”, in *Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, Dec. 2013, pp. 1–6.

[91] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, “Efficient inter-node mpi communication using GPUDirect RDMA for infiniband clusters with nvidia GPUs”, in *Proceedings of International Conference on Parallel Processing (ICPP)*, Lyon, France, 2013, pp. 80–89.

[92] S. Potluri, D. Bureddy, K. Hamidouche, A. Venkatesh, K. Kandalla, H. Subramoni, and D. K. Panda, “MVAPICH-PRISM: a proxy-based communication framework using infiniband and SCIF for intel MIC clusters”, in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, IEEE, Denver, Colorado, USA, 2013, p. 54.

[93] *Intel 5520 chipset and intel 5500 chipst,datasheet*, Intel, 2006. [Online]. Available: `http://www.intel.com/content/www/us/en/chipsets/5520-5500-chipset-ioh-datasheet.html`.

[94] D. Komatitscha, G. Erlebacherc, D. Göddeked, and D. Michéaa, “High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster”, *Journal of Computational Physics*, vol. 229, pp. 7692–7714, 2010.

[95] L. Oden, “GPI2 for GPUs: a PGAS framework for efficient communication in hybrid clusters”, in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, M. Bader and A. Bode, Eds., ser. Advances in Parallel Computing, vol. 25, IOS Press, Mar. 2014, pp. 461–470.

[96] M. Bernaschi, M. Bisson, and D. Rossetti, “Benchmarking of communication techniques for GPUs”, *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 250–255, 2013.

[97] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. Panda, “GPU-aware MPI on RDMA-enabled clusters: design, implementation and evaluation”, *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 2595–2605, 2013.

[98] (Apr. 5, 2014). CUDA-aware OpenMPI distribution, [Online]. Available: `http://www.open-mpi.org/faq/?category=building#build-cuda`.

[99] D. K. Panda. (Apr. 5, 2014). CUDA-aware MVAPICH2 distribution, [Online]. Available: `http://mvapich.cse.ohio-state.edu/overview/mvapich2/`.

[100] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda, “Design of high performance MVAPICH2: MPI2 over infiniband”, in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CC-Grid)*, IEEE, vol. 1, 2006, pp. 43–48.

[101] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, “MVAPICH2-GPU: optimized GPU to GPU communication for infiniband clusters”, *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 257–266, 2011.

[102] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, "Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication", in *Proccedings of6th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, IEEE, Shanghai, China, 2012, pp. 1848–1857.

[103] A. K. Singh, S. Potluri, H. Wang, K. Kandalla, S. Sur, and D. K. Panda, "Mpi alltoall personalized exchange on gpgpu clusters: design alternatives and benefit", in *Proccedings ofInternational Conference on Cluster Computing (CLUSTER)*, IEEE, Austin, Texas, USA, 2011, pp. 420–427.

[104] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda, "Optimized non-contiguous MPI datatype communication for GPU clusters: design, implementation and evaluation with MVAPICH2", in *Proceedings of International Conference on Cluster Computing (CLUSTER)*, IEEE, Austin, Texas, USA, 2011, pp. 308–316.

[105] A. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, K. R. Bisset, and R. Thakur, "MPI-ACC: an integrated and extensible approach to data movement in accelerator-based systems", in *Proccedings of International Conference on High Performance Computing and Communication & Embedded Software and Systems (HPCC-ICESS)*, IEEE, Liverpool, UK, 2012, pp. 647–654.

[106] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu, "The asynchronous partitioned global address space model", in *Proceedings of The First Workshop on Advances in Message Passing (AMP)*, Toronto, Canada, 2010, pp. 1–8.

[107] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou, "Unified parallel c for gpu clusters: language extensions and compiler implementation", in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, vol. 6548, Springer, 2011, pp. 151–165.

[108] S. Potluri, D. Bureddy, H. Wang, H. Subramoni, and D. K. Panda, "Extending openSHMEM for GPU computing", in *Proccedings of International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, Shanghai, China, 2013, pp. 1001–1012.

[109] D. Panda, *Enabling efficient use of upc and openshmem pgas models on gpu clusters*, San Jose, CA, USA, Mar. 2013. [Online]. Available: `http://on-demand.gputechconf.com/gtc/2014/presentations/S4528-upc-openshmem-gpas-gpu-clusters.pdf`.

[110] M. Baker, S. Pophale, J.-C. Vasnier, H. Jin, and O. Hernandez, "Hybrid programming using OpenSHMEM and OpenACC", in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, Springer, 2014, pp. 74–89.

[111] S. Poole, P. Shamis, A. Welch, S. Pophale, M. G. Venkata, O. Hernandez, G. Koenig, T. Curtis, and C.-H. Hsu, "Openshmem extensions and a vision for its future direction",

[112] D. Grünewald and C. Simmendinger, “The GASPI API specification and its implementation GPI 2.0”, in *Proccedings of7th International Conference on PGAS Programming Models*, Edinburgh, Scotland, UK, 2013, pp. 243–248.

[113] R. Machado and C. Lojewski, “The fraunhofer virtual machine: a communication library and runtime system based on the rdma model”, *Computer Science-Research and Development*, vol. 23, no. 3-4, pp. 125–132, 2009.

[114] (Oct. 12, 2014). Osu micro benchmarks, [Online]. Available: `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[115] D. Doerfler and R. Brightwell, “Measuring MPI send and receive overhead and application availability in high performance network interfaces”, in *Proceeding of PVM/MPI user Group meeting, Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Bonn, Germany: Springer, 2006, pp. 331–338.

[116] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, “Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers”, in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE,ACM, Seattle, Washington, USA, Nov. 2011, pp. 1–12.

[117] P. Micikevicius, “3d finite difference computation on gpus using cuda”, in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ACM, Washington, DC, USA, 2009, pp. 79–84.

[118] (Oct. 11, 2014). Himeno benchmark, Riken Institute, [Online]. Available: `http://accc.riken.jp/2444.htm`.

[119] L. Oden, H. Fröning, and F.-J. Pfreundt, “Infiniband-verbs on GPU: a case study of controlling an infiniband network device from the GPU”, *Proceedings of International Symposion on Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pp. 1–8, 2014.

[120] B. Klenk, L. Oden, and H. Fröning, “Analyzing put/get apis for thread-collaborative processors”, in *Proceedings of International Conference on Parallel Processing Workshops (ICPPW)*, IEEE, Minneapolis,Minnesota, USA, 2014, pp. 1–8.

[121] L. Oden, B. Klenk, and H. Fröning, “Energy-efficient stencil computations on distributed gpus using dynamic parallelism and gpu-controlled communication”, in *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, in Press, New Orleans, Louisiana, USA, 2014, pp. 1–10.

[122] J. A. Stuart and J. D. Owens, “Message passing on data-parallel architectures”, in *Proccedings ofInternational Symposium on Parallel & Distributed Processing, IPDPS*, IEEE, Roma, Italy, 2009, pp. 1–12.

[123] J. A. Stuart, P. Balaji, and J. D. Owens, “Extending MPI to accelerators”, in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ser. ASBD ’11, Galveston Island, Texas: ACM, 2011, pp. 19–23.

[124] M. Si and Y. Ishikawa, "Design of direct communication facility for many-core based accelerators", in *Proceedings of International Symposion on Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, IEEE, Shanghai, China, 2012, pp. 924–929.

[125] M. Si, Y. Ishikawa, and M. Tatagi, "Direct mpi library for intel xeon phi co-processors", in *Proceedings of International Symposion on Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, IEEE, Boston, Massachusetts, USA, 2013, pp. 816–824.

[126] S. Potluri, K. Hamidouche, D. Bureddy, and D. Panda, "MVAPICH2-MIC: a high performance mpi library for xeon phi clusters with infiniband", in *Extreme Scaling Workshop (XSCALE)*, Aug. 2013, pp. 25–32.

[127] *Fraunhofer itwm demonstrates gpi 2.0 with mellanox connect-ib and intel xeon phi*, Mellanox Technologies, Fraunhofer ITWM, and Intel, Jun. 2013. [Online]. Available: `http://www.mellanox.com/related-docs/whitepapers/WP_Fraunhofer_ITWM_Mellanox_Intel_CollaborationVersion1.pdf`.

[128] N. Bell and J. Hoberock, "Thrust: a productivity-oriented library for CUDA", *GPU Computing Gems*, vol. 7, 2011.

[129] (Jul. 14, 2014). Cublas-xt – accelerate blas calls with multiple gpus, Nvidia, [Online]. Available: `https://developer.nvidia.com/cublasxt`.

[130] (Jul. 14, 2014). Cufft, Nvidia, [Online]. Available: `https://developer.nvidia.com/cufft`.

[131] T. Scudiero and M. Murphy. (Jul. 14, 2014). Separate compilation and linking of CUDA C++ device code, Nvidia, [Online]. Available: `http://devblogs.nvidia.com/parallelforall/separate-compilation-linking-cuda-device-code/`.

[132] D. Merrill, *CUB: Kernel-level software reuse and library design*, GPU Technology Conference, 2013. [Online]. Available: `http://on-demand.gputechconf.com/gtc/2013/poster/pdf/P0267%5C_DuaneMerrill.pdf`.

[133] J. A. Stuart, M. Cox, and J. D. Owens, "Gpu-to-cpu callbacks", in *In Proceeding of Euro-Par 2010 Parallel Processing Workshops*, Springer, Bordeaux, France, 2011, pp. 365–372.

[134] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "Gpufs: integrating a file system with gpus", *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 1, p. 1, 2014.

[135] G. Kerr, "Dissecting a small infiniband application using the verbs api", *Computer Research Repository (CoRR)*, vol. abs/1105.1827, 2011.

[136] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for GPUs", *Computing Research Repository (CoRR)*, vol. abs/110.4623, pp. 1–12, 2011.

[137] S. Xiao and W. Feng, "Inter-block gpu communication via fast barrier synchronization", in *Proceedings of International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, Atlante, Georgia, USA, 2010, pp. 1–12.

[138] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads", in *Proceedings of Conference on Innovative Parallel Computing (InPar), 2012*, IEEE, San Jose, California, USA, 2012, pp. 1–14.

[139] A. Moerschell and J. D. Owens, "Distributed texture memory in a multi-gpu environment", *Computer Graphics Forum*, vol. 27, no. 1, pp. 130–151, 2008.

[140] Z. Fan, F. Qiu, and A. Kaufman, "Zippygpu: programming toolkit for general-purpose computation on gpu clusters", in *GPGPU Workshop at Supercomputing, Poster*, Tampa, Florida, USA, 2006.

[141] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: a nonuniform memory access programming model for high-performance computers", *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[142] (Jul. 30, 2014). Tuning CUDA applications for kepler, Nvidia, [Online]. Available: `http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html#ixzz38wvj9rz3`.

[143] (Jul. 30, 2014). CUDA C programming guide, memory fence functions, Nvidia, [Online]. Available: `http://docs.nvidia.com/cuda/cuda-c-programming-guide#memory-fence-functions`.

[144] L. Oden and H. Fröning, "GGAS: global GPU address spaces for efficient communication in heterogeneous clusters", in *Proccedings ofInternational Conference on Cluster Computing (CLUSTER)*, IEEE, Indianapolis, Indiana, USA, 2013, pp. 1–8.

[145] L. Oden, B. Klenk, and H. Fröning, "Energy-efficient collective reduce and allreduce operations on distributed gpus", in *Proceedings of International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE/ACM, Chicago,Illinois, US, 2014, pp. 483–492.

[146] H. Fröning, A. Giese, H. Montaner, F. Silla, and J. Duato, "Highly scalable barriers for future high-performance computing clusters", in *Proccedings of 18th International Conference on High Performance Computing (HiPC)*, IEEE, Bangalore, India, 2011, pp. 1–10.

[147] Y. Yang and H. Zhou, "CUDA-NP: realizing nested thread-level parallelism in gpgpu applications", in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP*, Orlando, Florida, USA, 2014, pp. 93–106.

[148] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for mpi", in *Proccedings ofInternational Conference on High Performance Computing, Networking, and Analysis (SC)*, IEEE,ACM, Reno, Nevada, USA, 2007, pp. 1–10.

[149] M. G. Venkata, P. Shamis, R. Sampath, R. L. Graham, and J. S. L. Ladd, "Optimizing blocking and nonblocking reduction operations for multi core clusters: hierarchical design and implementation", in *Proccedings ofInternational Conference on Cluster Computing (CLUSTER)*, IEEE, Indianapolis, Indiana, USA, 2013, pp. 1–8.

[150] V. Aggarwal, Y. Sabharwal, R. Garg, and P. Heidelberger, "HPCC randomaccess benchmark for next generation supercomputers", in *Proceedings of International Symposium on Parallel Distributed Processing (IPDPS)*, IEEE, Rome, Italy, May 2009, pp. 1–11.

[151] P. Luszcek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, *Introduction to the HPC challenge benchmark suite*, 2005.

[152] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with PAPI", in *41st International Conference on Parallel Processing Workshops (ICPPW)*, Sep. 2012, pp. 262–268.

[153] J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. Stone, and J. Phillips, "Quantifying the impact of GPUs on performance and energy efficiency in HPC clusters", in *Proceedings of Green Computing Conference*, IEEE, Chicago, Illinois, USA, 2010, pp. 317–324.

[154] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: memory power estimation and capping", in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, Austin, Texas USA: ACM/IEEE, 2010, pp. 189–194.