DISSERTATION

submitted to the

Combined Faculty of Natural Sciences and Mathematics

of the

Ruprecht–Karls University
Heidelberg

for the degree of

Doctor of Natural Sciences

put forward by

M.Sc. Benjamin Klenk

born in
Wertheim am Main, Baden-Württemberg

Mannheim, 2017

Communication Architectures for Scalable GPU-centric Computing Systems

Advisor: Professor Dr. Holger Fröning

Date of oral exam: ....

I would like to dedicate this dissertation to my loving parents

Elfriede and Dieter Klenk

and my loved and missed grandfather

Otto August Kimmel
(† July 1st, 2014)

Abstract

In recent years, power consumption has become the main concern in High Performance Computing (HPC). This has lead to heterogeneous computing systems in which Central Processing Units (CPUs) are supported by accelerators, such as Graphics Processing Units (GPUs). While GPUs used to be seen as slave devices to which the main processor offloads computation, today's systems tend to deploy more GPUs than CPUs. Eventually, the GPU will become a first-class processor, bearing increasing responsibilities.

Promoting the GPU to a first-class processor comes with many challenges, such as progress guarantees, dynamic memory management, and scheduling. However, one of the main challenges is the GPU's inability to orchestrate communication, which is currently entirely handled by the CPU. This work addresses that issue and presents solutions to allow GPUs to source and sink network traffic independently. Many important aspects are addressed, ranging from the application level to how networking hardware is accessed.

First, important and large scale exascale applications are studied to further understand their communication behavior and applications' requirements. Several metrics are presented, including time spent for communication, message sizes, and the length of queues that are required to match messages with receive requests. One aspect the analysis revealed is that messages are becoming smaller at scale, which renders the matching of messages and receive requests an important problem to address.

The next part analyzes how the GPU can directly access the network with various communication models being presented and benchmarked. It is shown that a flat address space of distributed GPU memories shows superior bandwidth than put/get communication or CPU-controlled message passing, but less communication can be overlapped with computation. Overall, GPU-controlled communication is always superior, both in terms of time-to-solution and energy

spending.

The final part addresses communication management on GPUs, which is required to provide high-level communication abstractions. Besides other fundamental building blocks, an algorithm for the message matching is presented that yields similar performance as CPUs. However, it is also shown that the messaging protocol can be relaxed to improve performance significantly, leveraging the massive amount of parallelism provided by the GPU's architecture.

Zusammenfassung

Die Rechenleistung heutiger Rechensysteme wird hauptsächlich durch deren Leistungsaufnahme beschränkt. Dies führt dazu, dass vermehrt effizientere Beschleuniger wie zum Beispiel Grafikprozessoren (GPU) eingesetzt werden, um den Hauptprozessor (CPU) zu unterstützen. Während anfänglich GPUs verwendet wurden um rechenintensive Abschnitte eines Programmes zu beschleunigen, tendieren aktuelle Systeme dazu mehr Grafik- als Hauptprozessoren einzusetzen. Dadurch entwickeln sich GPUs zu erstklassigen Prozessoren und werden mehr und mehr Aufgaben innerhalb eines Programmes übernehmen.

Eine der größten Herausforderungen ist es Grafikprozessoren zu ermöglichen direkt miteinander zu kommunizieren. Dies wird zurzeit vollständig von der CPU übernommen. Diese Arbeit befasst sich mit dieser Problematik und präsentiert Konzepte und Techniken die es der GPU erlauben Daten über das Netzwerk zu verschicken und zu empfangen.

Im ersten Schritt werden Anwendungen für Systeme die eine Milliarden Rechenoperationen in einer Nanosekunde ausführen können, so genannte *Exascale* Systeme, analysiert. Es werden unterschiedliche Charakteristiken dieser Applikationen präsentiert, unter anderem die Kommunikationszeit, Nachrichtengrößen und die Größe von Datenstrukturen die für die Zuweisung von Nachrichten (*Message Matching)* und Empfangsanfragen (*Receive Requests)* benötigt werden. Eine Erkenntnis ist, zum Beispiel dass Nachrichten kleiner werden sobald man die Anzahl an Rechenknoten erhöht. Dies erhöht die Bedeutung des *Message Matching* umso mehr.

Im zweiten Abschnitt wird die Interaktion zwischen GPUs und Netzwerkprozessoren betrachtet und Konzepte und Performanz von Kommunikationsmodellen präsentiert. Es wird gezeigt, dass ein flacher Adressraum über verteilte GPUs, im Vergleich zu einem *put/get* oder CPU-kontrollierten Nachrichtenmodell, hohe Bandbreiten und niedrige Latenzen ermöglicht. Allerdings erschwert es

auch Kommunikation mit Berechnungen zu überlagern, da viele GPU Ressourcen für den Datentransfer benötigt werden. Zusammenfassend gilt, dass die Anwendungszeit mit GPU-kontrollierter Kommunikation immer geringer ist und ebenso weniger Energie benötigt wird.

Im letzten Teil dieser Arbeit wird die Verwaltung von Kommunikation auf Grafikprozessoren untersucht. Dies ist besonders wichtig um komplexe Kommunikationsabstraktionen zu unterstützen. Es wird unter anderem ein Algorithmus präsentiert, der es erlaubt *Message Matching* auf GPUs durchzuführen. Diese Analyse erlaubt die Entwicklung eines weniger strikten Nachrichtenprotokolls mit sehr hohen Nachrichtenraten. Ebenso werden viele weitere Aspekte behandelt, wie Speicherverwaltung, Datentransfer, sowie Ereignis- und Benachrichtigungsmechanismen.

Table of contents

Chapter 1

Introduction

Computing has been the major driver of technology and science, allowing for many breakthrough inventions and discoveries. Although nearly every device in our everyday lives implements at least one microchip computer today, it's the supercomputers in large computing facilitates that drive rapid progress in science. The two traditional pillars of science, *theory* and *experiment*, are now complemented by *computational science*, for example simulation [1]. These systems are used by astrophysicists to simulate whole galaxies and predict their movements, by chemists to discover new materials for the next generations of computers, or by biologists to understand how cells are interacting in the human body. Supercomputing is omnipresent and helps society to be safer, find cures for the most deadly diseases, and to understand how the earth became our home. However, supercomputing itself is also an active research area in which scientists explore new architectures and algorithms to improve performance of compute installations to enable the most challenging problems to be solved.

Research in supercomputing not only encompasses several layers, such as hardware, Operating System (OS), middleware, and applications, it also aims to optimize the ecosystem to achieve the highest performance. The following elaborates on today's computing and how it is performed, but also on the system architecture. Then, the motivation for this work is presented, before an outlook for the remainder of this work is provided.

Parallel Computing

With the end of frequency scaling in the early 2000's, parallel computing has become the standard paradigm and architecture after which today's applications are designed. Parallelism can be expressed on various levels, from bit-level and instruction parallelism to task and node-level parallelism. All levels are important to achieve the highest performance in many scientific applications.

CPUs extensively exploit parallelism on instruction level due to pipelining and superscalar architectures, however, these techniques are not sufficient to meet performance goals. Furthermore, vector processing units support Single Instruction Multiple Data (SIMD) instructions to further exploit data parallelism, in which a single instruction is executed on multiple data elements. For example, current Intel CPUs have 256-bit wide vector registers. This allows to apply the same instructions to eight single-precision floating point values in just one clock cycle.

GPUs introduced the Single Instruction Multiple Threads (SIMT) model in which threads execute the same instruction on different data elements. Here, a thread is seen as a *sequence of SIMD lane operations* [2, Figure 4.12].

Parallelism of processors is exploited by the user and the compiler, which tries to vectorize instructions as much as possible. Since single processors do not offer enough performance, parallelism across processors and nodes is also much needed. This requires additional communication and synchronization between distributed processes. Common communication and programming models are further discussed in Chapter 3, while this chapter continues with the architecture of highly parallel supercomputers.

High-Performance System Architecture

Today's HPC systems have four main components. The CPU is the main processor, running the OS and application. The most common CPUs are Intel's Xeon processors with a share of 90% of the top 500 systems in 2016 [3], followed by IBM's PowerPC and AMD's Opteron. The second component is memory with the top system (Sunway TaihuLight) providing about 1GB per core, which also equals the third system's (Titan) memory provision. The next component is the interconnect, which is dominated by Infiniband (37%) and Ethernet 10G (36%).

However, half of the 500 system's performance is delivered by systems with custom interconnects, such as Cray's Aries, Fujitsu's Tofu, or Sunways' Sunway interconnect. The last important component are accelerators, whose share has been steadily increasing the past years. Here, NVIDIA's GPUs dominate the system's share, followed by Intel's Xeon Phis.

One of the main reasons for an increasing demand for accelerators is the end of *Dennard scaling*. In the past, chip manufacturers were able to keep reducing the size of transistors to put more and more of them onto a single die, while the power dissipation per die remained constant. This has been possible, because the power dissipation per transistor decreases linearly with the size, commonly referred to as Dennard scaling [4]. As the size of transistors keeps shrinking, current leakages become dominating, thus rendering it impossible to further sustain Dennard's rule [5].

With power becoming a severe challenge, application specific hardware is necessary to satisfy performance demands. This has lead to the general purpose application of GPUs, whose massively parallel architecture delivers remarkably high performance and energy efficiency. While it was difficult to program GPUs in the beginning, NVIDIA's introduction of the Compute Unified Device Architecture (CUDA) meant a breakthrough for parallel computing as GPUs became programmable through a C-language extension. Today, other programming environments such as *OpenCL* [6], *OpenACC* [7], and *OpenMP 4.0* [8] also allow to write code for GPUs.

Table 1.1 compares the CPU architecture with two accelerators, namely Intel's Xeon Phi and NVIDIA's Tesla GPU, both gearing to HPC. The Xeon Phi co-processor can be seen as a many-core x86 CPU, which is also able to boot an OS. Its performance is significantly higher than the Xeon CPU, but only excels if applications offer enough parallelism according to Amdahl's law [9]. This also applies to the GPU, which provides a completely different architecture with thousands of light-weight cores, operated at comparably low clock rates. As shown in the table, the power efficiency is more than 3 times higher than the CPU, and 1.5 times higher than the Xeon Phi. Although the memory is significantly smaller, the GPU's memory bandwidth exceeds the CPU's by about one order of magnitude. More details on GPUs will be provided in Chapter 2.

Table 1.1 Comparison of high-end processors, namely Intel's Xeon E7, Xeon Phi 7240, and NVIDIA P100 GPU.

	E7-8894 v4 [1]	**Xeon Phi 7240P**[1]	**P100** [10]
Cores/Processors	24	68	60 (SMs)
SP[2] ops/cycle[3]	768 (AVX2)	4352 (AVX-512)	7168 (CUDA)
Base Clock	2400MHz	1300 MHz	1328 MHz
Performance (SP)	1.8 TFLOPS	6 TFLOPS	10 TFLOPS
TDP[4]	160 W	275 W	250 W
Power efficiency	11 GFLOPS/W	22 GFLOPS/W	37 GFLOPS/W
Memory capacity	512 GB	16 GB[5]	16 GB
Mem. technology	DDR4	MCDRAM	HBM2
Mem. bandwidth	80 GB/s	500 GB/s	732 GB/s
Lithography	14 nm	14 nm	16 nm (FinFET)
Price	$ 8,000	$ 3,300	$ 8,000

[1] https://ark.intel.com/, last visited on June 8, 2017
[2] Single Precision (SP)
[3] Based on Fused Multiply Add (FMA) instructions
[4] Thermal Design Power (TDP)
[5] Supports also up to 384GB DDR4 memory

Communication in HPC Systems

Unless the problem is embarrassingly parallel, compute entities are required to communicate data and synchronize along the application's run time. While compute is relatively fast, communication has been identified as one of the major bottleneck in parallel computing [11, Section 8.2.2.1], especially in regard to power consumption.

Communication in traditional HPC systems mostly relies on the Message Passing Interface (MPI) in which processes exchange messages to synchronize and move data. The application is mainly run on the CPU and accelerators are only used for certain compute-intensive parts, thus communication is entirely orchestrated by the CPU. This so-called offloading model, which is similar to the master/slave principle, requires data to be copied to the GPU, for example, where it is processed and from where results are copied back.

However, there are highly parallel applications that mostly run on the GPU, for example the training of deep neural nets. These require more than a single GPU's compute power and in order to communicate with others, the GPU relies on the CPU to orchestrate communication. This does not only require data to be copied back and forth between the two processors, which negatively affects time, power, and energy, it also renders programming more difficult as the programmer has to explicitly deal with both domains. Furthermore, the entire GPU kernel needs to be left, thus implicitly synchronizing all threads, even if only parts of the kernel want to communicate data.

Solving this issue is the main objective of this work, which envisions direct GPU communication without any CPU interference. Related work has focused on the CPU assisting the GPU with communication by dedicating threads that receive requests from the GPU and execute communication on the CPU [12], [13] [14]. Others have studied the interaction between the GPU and the network controller, but left out how communication can be managed on GPUs [15]. Details on related work are presented throughout this work.

GPUs for Exascale Computing

The next milestone in supercomputing is to provide a system that is able to perform a quintillion (10^{18}) floating point operations per second, which is 10 times faster than the currently fastest computer. Furthermore, it is expected that an exascale system may not consume more than 20-30WM [16], which is only 1.5-2 times more than today's fastest computer and equals an efficiency of 16G Floating Point Operations Per Second (FLOPS)/W. Consequently, systems have to become much more power efficient, which renders GPUs more and more important.

While there will always be a fast single-thread optimized processor, the ratio of GPUs per CPU is increasing to further improve power efficiency. An example is NVIDIA's own supercomputer *SaturnV*, which is the 28th fasted in the world (November 2016). The system provides eight GPUs per node with two CPUs and is the most power efficient computer with 9G FLOPS/W. This is 90 times more power efficient than Sunway's TaihuLight system, the current number one system with respect to raw compute power.

The current trend toward GPU-centric systems also requires to move away from the offloading model. Instead, GPUs should be seen as peer processors, bearing the same responsibilities as a CPU. As peers, GPUs also need to be able to orchestrate communication and synchronization in order to not longer rely on the CPU.

This work studies how current GPUs can directly communicate with each other, without involving the CPU in the communication process. It is analyzed how the GPU can interact with Network Interface Controllers (NICs), which are going to be introduced in Chapter 5. Furthermore, in order to understand constraints, various exascale applications are studied in regard to their communication behavior and requirements. Based on this analysis, it is examined how communication can be managed on GPUs, for example how messages can be matched with receive requests.

Contributions

This work makes the following contributions:

I. *Understanding large scale communication* - various exascale proxy applications are analyzed in regard to communication. The analysis presents various metrics, valuable to application developers to understand bottlenecks and system architects to understand the applications' demands. For this work, it provides insights on requirements that communication architectures have to meet. This has been published at the *International Supercomputing Conference (ISC) 2017* [17] and was nominated for the conference's best paper award. Details are found in Chapter 4.

II. *Comparison of GPU-centric communication models* - three different models for direct inter-GPU communication are studied, based on different benchmarks with distinct characteristics. These models are traditional CPU-centric message passing, GPU-centric shared address space, and GPU-centric put/get. This is an important step toward specialized communication in heterogeneous systems as it reveals how well these models can be applied to GPUs in terms of time and energy. This contribution comprises various international workshop and conference publications [18] [19] [20] and is elaborated in Chapter 5.

III. *Introduction of a Software NIC (SoftNIC)*[1] *as a new concept for GPU-centric and managed communication* - managing communication is especially im-

[1]This has been part the Mantaro project, which explores communication management on GPUs. It is named after the river in Peru, which is believed to be one of the sources of the Amazon River.

portant at larger scale and well explored for CPUs. This work analyzes and shows what is necessary to manage communication on GPUs. This includes complex tasks, such as the matching of messages and receive requests, which has been published at the *International Parallel and Distributed Processing Symposium (IPDPS) 2017* [21] and received a best paper award. Furthermore, various building blocks and related work are discussed and evaluated. The SoftNIC aims to provide a flexible and highly scalable communication infrastructure for a massive amount of endpoints. The details are provided in Chapter 6.

Organization of this Work

Figure 1.1 shows various abstraction layers and the chapters in which these items are covered. Chapter 2 provides the background on the GPU's hardware, memory, and execution model. This is required for almost the entire remainder of this work.

Chapter 3 reviews various communication and programming models that are common in HPC. Besides MPI, shared memory models like Partitioned Global Address Space (PGAS) are also introduced and briefly discussed.

This is followed by the exascale application analysis in Chapter 4. The analysis focuses on message passing characteristics as this is also the most prevalent communication model in HPC. Insights are especially valuable for the communication management in Chapter 6. Here, it is evaluated how well the GPU can provide higher level communication abstractions, as opposed to relying on a flat shared address space. The chapter focuses on the matching of messages and receive request, because it is one of the main tasks required for message passing.

The insights of this work are discussed in Chapter 7. It aims to provide answers to what is still needed to render GPUs able to orchestrate communication. Features of the upcoming Volta GPU architecture are also taken into account. The discussion is followed by the conclusion in Chapter 8.

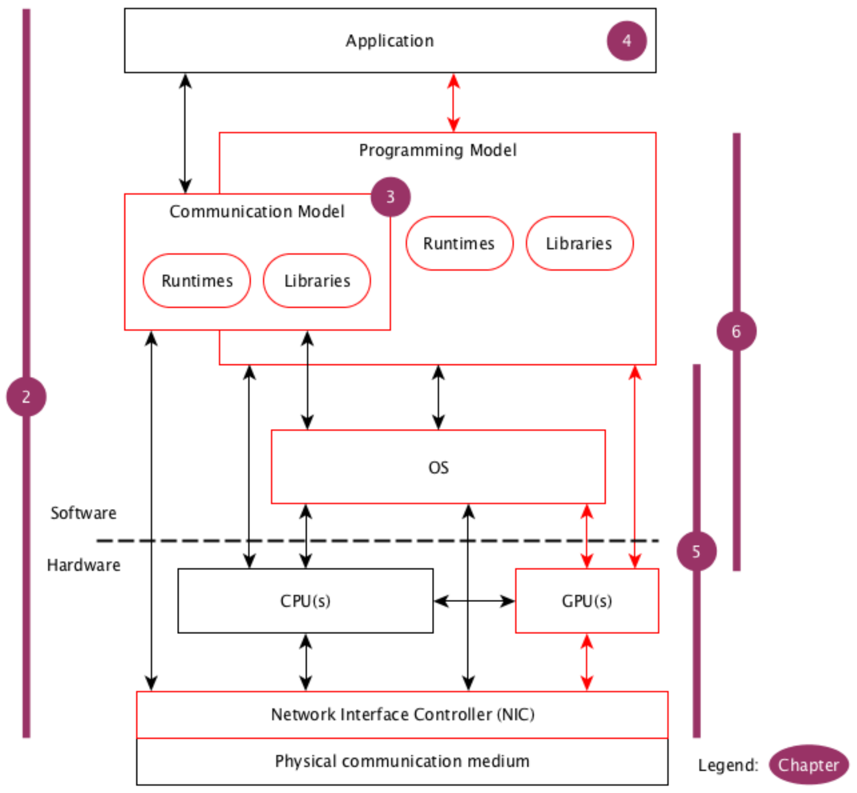


Fig. 1.1 Organization of this work on the basis of different abstraction layers.

Chapter 2

Graphics Processing Units

The first GPU with programmable shaders was introduced in 2001 by NVIDIA, aimed to accelerate image and video processing and allow programmers to implement their own programs on GPUs. Soon after, researches started to write scientific simulations in graphics languages to execute them on the GPU, using languages like *cg* or *OpenGL* [22]. In 2007, NVIDIA started CUDA and allowed GPUs to be programmed in a C-like language. This tremendously increased the number of general purpose GPU applications and opened the door for the GPU to enter the HPC area.

The following provides important background on GPU computing, including its architecture, memory, execution and programming model. More details can be found in the CUDA programming guide [23], or the excellent books of Shane Cook [24] and Nicholas Wilt [25].

2.1 Architecture

An high-level comparison between a CPU and a GPU is shown in Table 1.1. Although the number of cores, respectively processors, is similar, each Streaming Multiprocessor (SM) itself is highly parallel and implements many light-weight cores. This renders the GPU a massively parallel processor with memory that provides extremely high bandwidth. The target application for GPUs has always been graphics, wherein most operations are matrix multiplications to transform

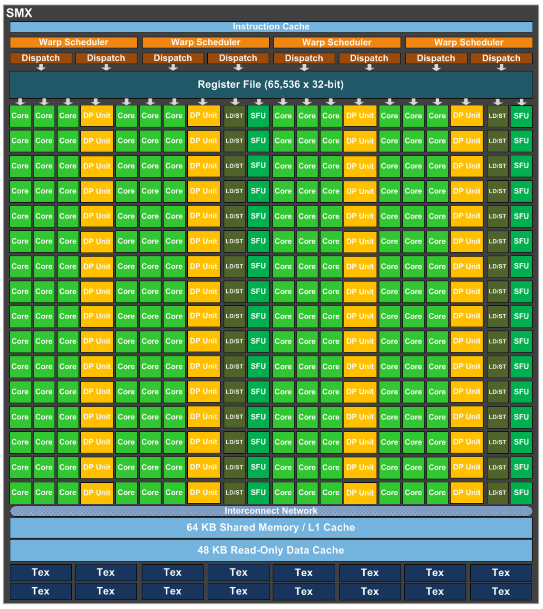

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Fig. 2.1 Architecture of the Kepler SM [26].

and scale vertexes. Most of the computation can be performed in parallel, which is the reason for the highly parallel architecture of GPUs.

2.1.1 Compute Architecture

The Kepler GK110 architecture [26] implements a total of 15 SMs. The SMs are interconnected by an internal network to which the memory is also attached. The architecture of a single SM is depicted in Figure 2.1. There are a total of 192 so-called CUDA cores, 32 load/store units, 64 double-precision units, and 32 special function units. Each core is fully pipelined and able to perform single-precision and integer operations [2, p. 307].

GPUs implement a SIMT model, in which the same instructions are dispatched to a group of threads operating on different data. Currently this group comprises 32 threads and is referred to as *warp*. A Kepler SM provides four warp schedulers with two instruction dispatch units each. Each scheduler selects an available warp and issues two independent instructions to the hardware to be executed. A *scoreboard* [2, pp. 294-306] is used to keep track of warps and provides information on whether a warp is available for execution or awaits completion of previously issued instructions. It is important to have much more warps than execution units in order to hide instructions latencies, especially memory accesses. This principle is known as oversubscription and key to the GPU's high compute performance.

Another important aspect is branch divergence, which occurs when threads within a warp follow a different execution path. As aforementioned, threads of the same warp are given the same instruction. In case of branches, such as *if* statements, the entire warp executes the same instruction, but the results from divergent threads are discarded. This reduces efficiency and should be avoided. Furthermore, synchronization primitives in diverging paths result in an undefined behavior.

On top of the SM's warp schedulers, the GPU implements a Collaborative Thread Array (CTA) scheduler. A CTA is a group of threads and assigned to an SM for execution. Once a CTA is running it completes before another CTA can be brought to execution. Currently, CUDA limits the size of a CTA to 1,024 threads or 32 warps, respectively. Depending on the size of a CTA, multiple CTAs can be executed by one SM concurrently.

The peak performance can only be achieved if the GPU is kept busy and occupancy is high, which means all SMs are fully occupied with executing CTAs. The occupancy mainly depends on the number of threads per CTA, the number of allocated registers per thread, and the amount of allocated shared memory per CTA. For example, if a CTA consumes all of the available shared memory of an SM and the number of threads is low, the SM cannot execute another CTA although compute resources would be available. Goal of every kernel is to maximize occupancy, which sometimes means to launch less threads per CTA to increase the number of CTAs per SM that can be concurrently executed.

Table 2.1 compares three different GPU architectures in regard to their specification. The Kepler K80 and Pascal GTX1080 will be used for experiments later and the Volta V100 serves as a reference for the latest GPU. Volta was announced at the end of this work, thus it was not possible to run experiments on this card as it wasn't available yet. Nonetheless, the comparison shows how much and rapidly GPUs are advancing and progressing.

The Pascal architecture differs from Kepler, whose SM is shown in Figure 2.1, as its SM implements 128 cores, grouped into four blocks of 32 cores each. This allows for better warp scheduling and increases efficiency. More on Pascal can be found in its architectural white paper [10].

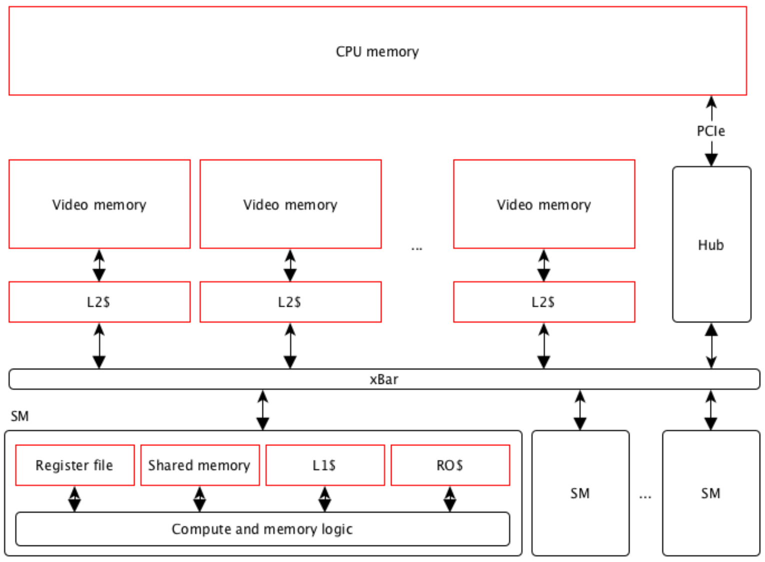


Fig. 2.2 Memory system of a Kepler-class GPU.

2.1.2 Memory System

The memory system of GPUs differs significantly from CPUs. Although much smaller in size, the GPU's video memory, often also referred to as global memory, offers tremendous bandwidth that is an order of magnitude higher than the CPU's Double Data Rate (DDR) memory. The main reason is that video memory is optimized for bandwidth with a much wider data bus and higher voltage at the cost of higher access latency. However, as opposed to CPUs, GPUs can hide longer latencies extremely well due to oversubscription of resources.

Figure 2.2 shows the memory system of Kepler GPUs. There is a two-level cache hierarchy, however, their principle is different from CPUs in which caches are used to minimize latency. In GPUs, the streaming caches are used to reduce traffic on the video memory and increase bandwidth. There is also no coherency between L1 caches. L2 caches cover a certain address range, hence no coherency is required as well. The read-only cache per SM allows for fast access to constant variables and improves texture performance.

Each SM also implements explicitly managed scratchpad memory, called *shared memory*. Shared memory provides low-latency and high-bandwidth access to data, but the user has to explicitly administer the data it holds. Furthermore, the memory is divided into interleaved banks and accesses to the same banks are serialized, hence increasing latency.

Although the memory system allows for high bandwidth, the access pattern

Table 2.1 Comparison of the Kepler, Pascal, and Volta GPU architectures.

	Kepler K80[1]	**Pascal GTX1080**	**Volta V100**
Release year	2014	2016	2017
Market	Tesla	GeForce	Tesla
Base Clock	562MHz	1607MHz	1370MHz
SMs	2x 13	20	84
CUDA Cores	2x 2496	2560	5376
Memory	2x 12GB	8GB	16GB
	GDDR5	GDDR5	HBM
Memory Bandwidth	2x 240GB/s	320GB/s	900GB/s
Shared Memory/SM	2x 64KB	96KB	96KB
Register File/SM	2x 256KB	256KB	64KB
Power TDP	300W	180W	300W
Lithography	28nm	16nm (FinFET)	12nm (FinFET)
Transistors	2x 7.1B	7.2B	21B
Price	$5,000	$600	n/a

[1] The K80 GPU is a dual GPU with two chips per card.

is of great importance. A memory transaction consists of 128B, thus 32 single precision or integer values. Accessed memory addresses should be consecutive in order to allow the memory system to coalesce requests into a single transactions, maximizing utilization and efficiency.

The memory specification of various GPU architectures are also shown in Table 2.1.

2.2 Programming and Execution Model

GPUs are programmed in CUDA, which extends the C/C++ language. Today, there are several bindings for various programming languages, including Python, Fortran, Java, or MATLAB. Nonetheless, this work is entirely based on C/C++ and the following presents how memory is managed and work can be launched on the GPU. Approaches like *OpenACC*, in which the compiler translates CPU code to GPU code based on `#pragma` directives, are not presented. Although this work's techniques could also be applied to these programming models, the following focuses solely on CUDA C/C++.

2.2.1 Memory Management

GPUs have their own on-device memory in which data should reside to yield high performance. Consequently, GPU memory has to be allocated through CUDA's Application Programmable Interface (API), for example `cudaMalloc`. The function allocates memory on the GPU and returns a pointer. The pointer cannot be accessed by the CPU and needs to be passed to the kernel, which comprises all instructions that will be executed on the GPU.

When memory is allocated on the device, data can be copied using `cudaMemcpy`. This instructs the GPU's copy engine to copy data from system to device memory and vice versa. The best performance is achieved when the host data is pinned and thus cannot be swapped to disk. Due to Unified Virtual Addressing (UVA), the GPU can also access pinned memory directly when a specific device pointer is passed to the kernel, obtained by `cudaGetDevicePointer`. Pinned memory is usually allocated by `cudaHostAlloc`.

CUDA version 6.0 introduced *Unified Memory*, with which memory can be simply allocated with `cudaMallocManaged`. The resulting pointer can be used on both CPU and GPU and data is copied between host and device automatically. The Pascal GPU even provides a page migration engine that can trigger page faults, which significantly reduces the effort to program GPUs and manage memory. Upcoming Linux kernels are expected to support this as well, so that memory can simply be allocated by standard `malloc` and be used by both CPU and GPU.

2.2.2 Code Execution

CUDA extends the C/C++ language by additional keywords to inform the compiler about code that is supposed to be executed on the GPU as opposed to the CPU. Kernels are declared with the `__global__` keyword and device functions with `__device__`. These functions are compiled by NVIDIA's *nvcc* compiler and can contain GPU specific intrinsics. The code is compiled to Parallel Thread Execution (PTX), which is a stable and abstract instruction set and hides the hardware specific instruction set, ensuring compatibility across multiple generations of GPU architectures [2, pp. 298].

Kernels are launched from the CPU like regular functions, but require to be passed information about the kernel's configuration, encapsulated in <<< and

>>> delimiters before the opening bracket for the function's arguments. The configuration comprises the size of the grid, respectively the number of CTAs, the number of threads per CTA, and the size of dynamically allocated shared memory per CTA. Once the kernel is launched control returns back to the CPU, which can synchronize with the GPU through `cudaDeviceSynchronize` again.

Many functions in CUDA also have asynchronous implementations, such as `cudaMemcpy` and `cudaMemcpyAsync`. These functions immediately return and synchronization has to be handled explicitly. Asynchronous functions, together with kernels, can be assigned to *streams* to allow for more parallelism. Streams are independently executed in parallel, whereas functions in the same stream are executed sequentially. This also allows to pipeline and overlap data transfers to and from the device with kernels.

CUDA 5.0 introduced *dynamic parallelism*, with which kernels can be launched within other kernels on the GPU. While this allows adaptive applications to avoid returning control back to the CPU, kernel launch latencies are quite high. Besides kernels, many other operations are callable from the device, such as `cudaMemcpy`, `cudaMalloc`, or stream-related functions. Nonetheless, due to their high latency these should be used with care as performance can be reduced significantly [27].

A rather unique feature is *warp vote* functions. As aforementioned, threads within a warp follow the same control path and communication among threads within a CTA requires shared memory. However, warp vote functions allow threads within a warp to use the register file to communicate data. Table 2.2 shows a brief overview of the most common warp vote functions and bit-vector intrinsics. Especially the `__shfl` operation is widely used and provides fast intra-warp communication through the register file. These intrinsics are executed extremely fast compared to communication through shared or global memory.

2.2.3 Multi-GPU Programming

Programming multiple GPUs with CUDA requires explicit device management. This means the user has to declare the device to which all subsequent CUDA calls are assigned. In practice, it is common to statically assign a CPU thread or process, for example using MPI, to each GPU. All operations called by a thread are then always issued to its device.

Table 2.2 Overview of common warp vote functions and math intrinsics, available in Kepler and newer generations of GPUs.

Intrinsic	**Description**
`__shfl(int val, int src)`	Exchanges data within a warp. Data can either be broadcasted by a single thread or distributed by a certain pattern. The function returns the received value.
`__ballot(bool cond)`	The function takes a condition as argument and returns a 32-bit vector wherein each bit represents each thread's evaluation of the condition.
`__any(bool cond)`	Returns true if any of the threads evaluate the condition to true.
`__ffs(int arg)`	The function returns the position of the first '1' within a 32-bit vector that is passed as an argument.
`__clz(int arg)`	The function returns the number of consecutive '0's starting at the MSB within a 32-bit vector that is passed as an argument.

Communication between GPUs requires UVA and is possible through the Peripheral Component Interconnect Express (PCIe) protocol's peer-to-peer feature. Here, devices can access each other without the CPU. UVA allows to map another GPU's memory with access on load/store level. Furthermore, `cudaMemcpy` can be used to copy data between memories allocated on different GPUs.

In distributed systems, communication between GPUs relies on the CPU and its communication primitives, for example MPI. However, there have been efforts to improve data transfers and GPU-NIC interactions. Table 2.3 shows the history of *GPUDirect*, which has steadily been extended to further support inter-node communication. *GPUDirectRDMA* allows a NIC to read from and write to GPU memory without the CPU. However, this feature should only be used if both devices share the same PCIe root complex. In 2016, NVIDIA released *GPUDirectAsync*, which allows the GPU to trigger network operations. Here, the CPU generates a work plan and submits everything to a stream. The work plan can now include certain memory locations that are written by the GPU directly, for example a register that has the network device beginning with the data transfer upon being written. Some MPI implementations are currently looking into adding this feature, with which a CUDA stream could be passed

Table 2.3 History of GPUDirect, which aims to improve GPU-NIC interactions. Note that communication is still managed on the CPU.

Technology	**Description**	**Year**
GPUDirect	The same memory allocation can be pinned and used for GPU and third-party devices, such as NICs.	2010
GPUDirectP2P	Data can directly be copied between GPUs without any host memory staging copies through PCIe.	2011
GPUDirectRDMA	Third-party devices can directly access the GPU's memory through PCIe's peer-to-peer protocol.	2013
GPUDirectAsync	Memory operations can be added to streams, meaning the GPU can trigger events after completing tasks.	2016

to MPI operations. The MPI operation would then be triggered by the GPU without involving the CPU at all.

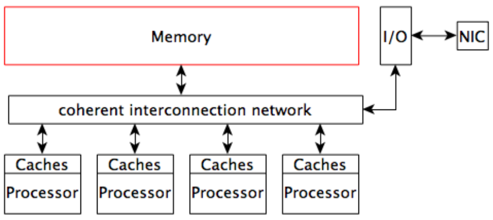
Chapter 3

Communication and Parallel Programming Models

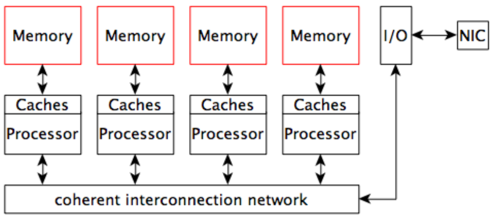
The previous chapter provided a comprehensive introduction to GPUs and this chapter introduces various communication and parallel programming models for large scale systems, which are going to be discussed and evaluated throughout this work with respect to their compatibility with GPUs. While these models are generally described here, the next chapters look at them from a more GPU-centric perspective.

Generally, parallel computing systems can be categorized into two fundamental approaches, depending on how the memory is accessed. An overview is depicted in Figure 3.1. *Shared memory systems*, for example, provide a single address space which can be accessed by all processors. This can be implemented by either having a single memory to which each processor is attached, or by allowing any processor to access another processor's local memory. Here, it is distinguished by the access costs of a memory access. In a Uniform Memory Access (UMA) system each memory access has the same latency, whereas access latencies differ in Non-Uniform Memory Access (NUMA) systems as some processor's memory is farther away than others and thus accesses require more time.

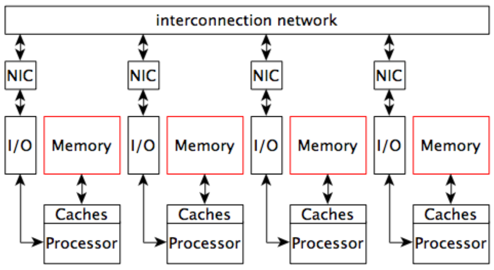
The second class are *distributed memory systems.* As suggested by the name, the memory is distributed across the system and primarily accessed by the local processor. Data transfers between processors require address translation, performed by the NIC at both endpoints.



(a) UMA shared memory system



(b) NUMA shared memory system



(c) Distributed memory system

Fig. 3.1 Illustration of shared and distributed memory systems.

Independent of the underlying system architecture, the programming model defines how a system is programmed and how the user perceives the system. The communication model, on the other hand, defines how data is exchanged.

Since 2003, the largest and most powerful computing systems have always been distributed memory systems [3], with an increasing share of clusters and Massively Parallel Processing (MPP) systems. Therefore, the following is only concerned about programming and communication models for distributed memory systems. More on the system architecture can be found in the book of Hennessy and Patterson [2].

3.1 Characteristics of Programming Models

A very simple classification of programming models is to distinguish by control flow and data. The simplest and sequential model is Single Program, Single Data (SPSD), in which only a single program accesses non-shared data. The other end of the spectrum is represented by Multiple Program, Multiple Data (MPMD). Here, multiple processes are executed, each operating on different data. There are two other models in between these two, namely Single Program, Multiple Data (SPMD) and Multiple Program, Single Data (MPSD). However, the most relevant model is SPMD in which the same program is executed by multiple processes, but on different data.

Other important characteristics of programming models are communication, synchronization, consistency, and coherency, which are briefly described in the following.

Communication Exchanging data is key in every parallel program and the programming model implements communication either explicit or implicit. Implicit communication means that variables can be assigned to other variables even if their data does not reside in the same memory, or even in the same node. When the assignment is executed communication is triggered, for example a message is sent from the process that owns the right operand to the process owning the left operand. Explicit communication, however, requires the user to invoke communication. Here, data locality is explicitly expressed, for example through send and receive operations.

Synchronization Similar to communication, synchronization can be implicit or explicit. While message passing provides implicit synchronization, assigning remote variables to local variables may require explicit synchronization to avoid race conditions. Hence, shared memory models provide synchronization primitives such as atomic operation, mutexes, semaphores, and barriers.

Consistency [2, Section 5.6] If multiple threads or processes share data the order and point in time at which updates become visible to others pose an important challenge, to which the consistency model provides the answer. The most strict model is *sequential consistency* in which everything appears in the order it is posted. However, as memory instructions cannot pass others and the compiler is prohibited from reordering instructions this model limits performance significantly. Thus, a relaxed consistency model allows for better performance, but the user now bears the responsibility to ensure consistency by explicitly invoking memory fences. A memory fence guarantees that all memory operations prior to the fence are visible when the fence operation returns. Consistency is an important challenge and often the reason of faulty parallel programs.

Coherency [2, Section 5.2] While consistency concerns multiple addresses, coherency must ensure that every thread or process perceives all updates to the data. However, this is only an issue if the same address is cached or kept in multiple memories. For example, it is assumed that both threads A and B own a copy of data Φ. When A issues a write operation on Φ, the update needs to be broadcasted to ensure that B perceives the update on Φ. With an increasing number of processes or threads, coherency is a major challenge and often limits performance of shared memory applications at large scale.

The following introduces various parallel programming and communication models that are common in HPC.

3.2 Message Passing

This section introduces message passing, which is perceived as the most dominant communication model in distributed memory systems, such as cluster architectures. The focus lies on MPI since it comprises a comprehensive set of features and is widely used in HPC. In addition, other interfaces are also briefly introduced. The most common MPI implementations are *OpenMPI* [28], *MPICH* [29], and *MVAPICH* [30].

3.2.1 Implementation

Message passing, particularly MPI, has become the de-facto standard for communication in systems with distributed memory. The principle is simple in that communication is performed by sending and receiving data in the form of messages. The application is written using the SPMD paradigm, in which a program is executed by multiple processes, each being assigned a unique identifier, referred to as *rank*. A rank can be used in send or receive operations to address other processes in point-to-point communication. Multiple ranks can be grouped together to a so-called *communicators*. Messages can only be exchanged with ranks within the same communicator, of which multiple can contain the same rank. In MPI, data is identified by naming and ordering. Naming is given by the source, communicator, and a *tag*, which is an arbitrary integer value chosen by the user. If multiple messages are identical in naming, the order in which the messages arrive determines which message is matched with the corresponding receive request.

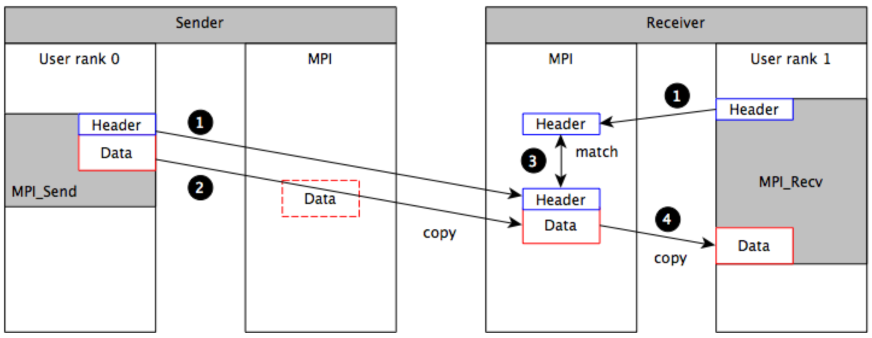
There are two basic protocols for messaging in MPI: *eager* and *rendezvous*. Both protocols differ in their buffering scheme and the selection of the protocol that is applied is usually based on the message size. The principle is illustrated in Figure 3.2.

Eager Protocol In the eager case, the header is written to the receiving process' mailbox (❶), where it is matched with the receive request (❸). The data is either buffered at the sender or, if the the payload is small enough, sent along with the header and kept in the receiver's mailbox (❷) until it is copied to the user buffer (❹). It must be ensured that the receiver provides enough memory capacity for the message to be stored. Thus, it is common to use a credit based system, in which the receiver hands off credits to the sender and when a message is sent a credit is consumed. After receiving some messages, credits are returned back. If there are no credits available, the rendezvous protocol is used as a fallback solution.

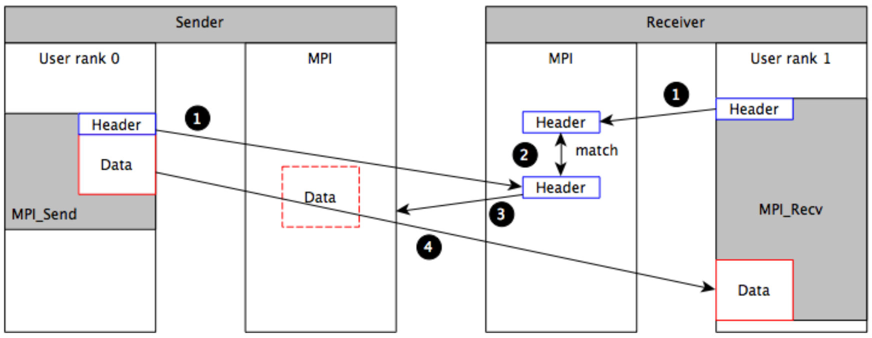
Rendezvous Protocol If a large message is to be sent, the extra copy from the system to the user buffer at the receiver side can significantly add latency and negatively impact performance. Thus, the rendezvous protocol is used for large messages. First, the header is sent to the receiver (❶) where it is matched with the receive request (❷). When a matching receive is found, the receiver's user buffer address is returned to the sender (❸), who starts the copy operation from user to user buffer (❹). Alternatively, the receiver can use a Remote Direct Memory Access (RDMA) transfer to fetch the data from the sender (*get* operation). The MPI implementation may also buffer the data at the sender side to allow the send routine to return more quickly.

If the user relies on the standard *MPI_Send* routine, MPI chooses the protocol based on the message size and available credits. However, the user may also force MPI to use a certain protocol, for example, *MPI_Ssend* (synchronous send) for rendezvous and *MPI_Rsend* (ready send) for eager transfers.

In order to allow communication to be overlapped with computation, MPI also provides non-blocking operations. For example, *MPI_Isend* ('I' stands for immediate) returns immediately and the send buffer must remain untouched until the operation completes. Completion can be queried by *MPI_Test* or ensured by *MPI_Wait*, which blocks until the request is processed.



(a) Eager



(b) Rendezvous

Fig. 3.2 MPI's eager and rendezvous protocol for point-to-point communication.

Message Matching An important aspect is the matching of receive requests and message headers, often also referred to as *tag matching*. The matching is based on the source, tag, and communicator, whereby a wildcard can be used for the source and tag. If these criteria apply to multiple messages, the order in which they are received decides which message is the right match in that the oldest messages are matched first. However, it may occur that a message arrives before the matching receive is posted, or the receive is posted before the message arrives. Thus, MPI maintains special data structures to keep track of receive requests and messages.

Messages that arrive but cannot be matched with already posted receive requests are added to the so-called Unexpected Message Queue (UMQ). Although the name suggests a queue, the most common MPI implementations use lists due to their superior properties regarding removing an element at arbitrary positions. When the receiver posts a new receive request, the UMQ needs to be searched for a match first. If no match is found, the request is appended to the Posted Receive Queue (PRQ). The overall process is depicted in Figure 3.3. The matching is especially crucial to the latency of small messages and synchronizing operations.

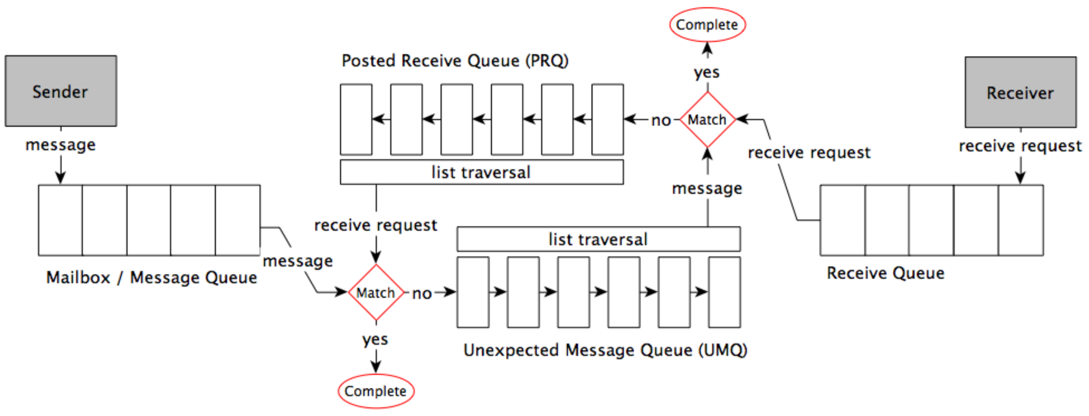


Fig. 3.3 MPI's matching process for incoming messages and receive requests.

It is important that MPI ensures progress of the application, for which there exist three basic strategies. First, an incoming message or receive request could trigger an interrupt, allowing the CPU to enter the MPI library to perform the matching. However, a large number of messages constantly interrupts the CPU and thus limits performance. Second, helper threads could be dedicated to ensure progress, constantly polling on the message and receive request queues. The third approach calls a progress routine every time an MPI routine is entered from the application. This may require more memory as many messages can be received in between two MPI calls, but improves performance.

Collective operations Besides point-to-point messages, MPI also provides collective operations that are called by a group of ranks. For example, a broadcast allows a single rank to send the same data to multiple processes, whereas a collective reduce operation can be used to determine the sum of distributed data. Collectives are optimized within the MPI library and reflect common patterns in scientific computing. The latest MPI, version 3.0, even provides non-blocking collective operations to maximize overlap with computation. One of the most used collective operation is the barrier, which synchronizes all ranks of the same communicator.

3.2.2 Active Messages

Another and more sophisticated type of messages is active messages, in which messages trigger the execution of code. This can be used to move tasks closer to the memory in which the data resides. A common implementation of active messages is the Global Address Space Network (GASNet), mostly used in PGAS environments and Distributed Shared Memory (DSM) systems, which are introduced in the subsequent section. The principle of active messages can be compared to Remote Procedure Calls (RPCs), which are more commonly used in operating systems.

GASNet [31] consists of two layers, namely core and extended API. The core API is network-specific and directly builds upon the network's API, while the extended API provides high-level abstractions to exchange active messages and data, or to ensure synchronization. The user may register handlers that are invoked upon receipt of an active message, which itself contains arguments that are passed to the handler. When the handler has been executed a reply may be sent back to the requesting node, which perhaps triggers the execution of another handler. However, no further reply messages are generated. Besides active messages, GASNet also provides data transfer operations, based on put/get semantics. Remote memory has to be registered using RDMA in advance to avoid local staging copies.

The main application of GASNet is Unified Parallel C (UPC) [32] or Titanium (parallel dialect of Java for HPC) [31], which implement the PGAS model. An important aspect of UPC is compilation as the compiler can try to overlap computation and communication by relying on relaxed consistency semantics automatically. Hence, GASNet is tailored to be used by compilers rather than exposing its API directly to the user. This differs from MPI, which offers an end-user library.

Another more recent application of GASNet is *Legion* [33], which implements a task-based programming model and is going to be introduced later in this chapter.

3.2.3 Other Message Passing Systems

Portals [34] is another network programming interface that allows for efficient message passing, but also one-sided communication. However, it is barely used directly and often used within MPI, GASNet, or OpenSHMEM [35]. Similar to MPI, Portals supports unexpected messages and posted receive requests.

Erlang [36] is a functional programming language that supports messaging. Messages are put into process' queues, in which they are matched with receive requests. However, the matching is much simpler than MPI, for example. Messages can contain any object and be sent to any other process. Erlang is often used for message handling systems and allows to use C-bindings for high-performance.

Akka [37] is a distributed parallel extension to Java/Scala and supports messaging. It supports non-blocking messages, unordered delivery, and message queues can be shared among multiple processes. Message are matched according to a certain pattern, which the user has to provide, for example sender and receiver process ID.

3.3 Distributed Shared Memory

As opposed to transferring data by sending and receiving messages, DSM models allow for fine-grain communication through load/store operations, whereas the granularity depends on the system. While single words can be addressed, the system may decide to move or copy entire pages to optimize for spatial and temporal locality. There are two basic principles in that the address space is either flat or partitioned. Both are briefly introduced in the following.

3.3.1 Virtual Shared Memory

In a shared memory system all processors see the same address space and can access data even if it physically resides in some other processor's memory. This makes communication key and coherency and consistency pose tremendous challenges. However, the programmer's view on a shared memory system seems simpler than message passing, for example. Data can be accessed everywhere and communication is implicitly handled by the system, but it also renders the need for fine grain communication and synchronization inevitable.

One of the most performance critical aspects about DSM systems is coherency. A simple snooping protocol as implemented in many Symmetric Multiprocessors (SMPs) is not scalable as the overhead causes network traffic to increase proportionally to the number of processors squared. Even with the memory bandwidth increasing linearly, the network's bandwidth is effectively reduced by the number of processors. Since the required broadcast semantic accounts for this overhead, directory-based coherency is used in large scale DSM systems [38, Section 1.3.2], avoiding broadcasts by only involving processors that keep copies of the cached data. However, even such system require mechanisms to effectively hide latencies.

Another approach is virtual shared memory, in which shared memory accesses are mapped onto messages. Here, coherency is achieved on page level and Translation Lookaside Buffer (TLB) faults trigger mechanisms to ensure a coherent view on the memory. Since coherency is maintained in software it can be better adapted to the application. However, because of pages being the smallest coherent entity data transfers become large and false sharing is likely to reduce performance [38, Section 5.6].

3.3.2 Partitioned Global Address Space

One significant drawback of a large flat address space is the unawareness of data locality. Thus, the PGAS approach extends addresses by the affinity to the memory in which the data resides. This allows threads to rather work on data that is local as opposed to remote memory, whose accesses entail high latency. There are two main operations to access the memory: fine-grain *load/store* accesses and bulky *put/get* operations.

In PGAS, memory can be allocated in private or shared memory. As the name suggests, private memory can only be accessed by the thread from which the memory was allocated and shared data can be accessed by all threads, however, with varying access costs. The model allows users to treat cluster systems like shared memory systems, easing the programming with implicit communication. In fact, communication is often inserted by the compiler. However, in order to yield high performance the user has to be careful with data placement, synchronization, and consistency.

A common implementation of PGAS is the previously mentioned UPC language, which uses GASNet and its active messages. More recent efforts have given rise to Chapel [39], which has been developed by Cray and provides high-level abstractions, aiming for better productivity and scalability. The user can specify the mapping of data and processes. Accessing remote data is made simple by extending variables by a *locale* characteristic (e.g. *variable.local*) that returns the ID of the process that owns the variable. Communication is implicit by assigning remote to local variables. Although programming is made simpler and high-level abstractions are provided, Chapel currently struggles to keep up with performance demands but is constantly improving in this direction.

3.4 Task-based Programming

Both message passing and DSM have been widely used in HPC, but as computing systems keep increasing and become more heterogeneous their static behavior becomes hard to manage. For example, MPI requires to specify a mapping after which ranks are assigned to processors and cores, respectively. Within the application, messages are exchanged between distinct ranks, which requires the user to ensure the availability of the data to the ranks.

Task programming aims to be more dynamic and the user specifies constraints and relationships between data and tasks, whereas the system decides where tasks are executed. This is usually based on resource availability and data locality. Another important aspect is reliability. Dynamic task programming allows to move tasks away from failing nodes, improving the overall system reliability and availability.

Legion An emerging task-based runtime is *Legion* [33], which stands for "logical regions". A logical region comprises a set of objects and can be dynamically allocated and deallocated. It also contains locality information and may be passed to tasks. Computational independence is expressed by sub-regions and privileges, such as read- or write-only, and coherency, for example exclusive or shared. Based on this information, the Legion runtime decides which and where tasks can be run concurrently. Data is also automatically exchanged or replicated, for example if two tasks do not modify the same region. Nonetheless, the runtime also provides a mapping interface to control how tasks are assigned to processors and regions to physical memory units. The authors state that an application-specific mapping usually provides the best performance.

An important aspect of Legion is the scheduler, which the authors refer to as Software Out-Of-Order Processor (SOOP). They compare SOOP to a physical processor as tasks are pipelined, executed out-of-order, and constrained by dependencies between tasks. Furthermore, Legion uses deferred execution, meaning the execution of a task is decoupled from the time it was issued. This allows to hide latencies and improve processor utilization. The deferred execution requires a low-level runtime event system, namely *Realm* [40]. Realm builds upon GASNet and allows to express dependencies in an efficient way.

Legion's performance evaluation considers various benchmarks, such as circuit simulation, adaptive mesh refinement, and particle simulation. The authors compare to hand-coded versions of these codes and are able to report remarkable speedups with high scalability.

OmpSs Introduced and developed by the Barcelona Supercomputing Center (BSC), *OmpSs* [41] is a task-based extension to OpenMP. The user specifies data dependencies and the data movement is taken care of by the compiler. Similar to OpenMP, everything is based on directives. It also supports accelerators like GPUs.

X10 Introduced and researched by IBM, *X10* [42] follows the Java/Scala syntax with focus on parallelism, distributed systems, and productivity. It claims to implement the Asynchronous Partitioned Global Address Space (APGAS) model [43], an extended PGAS with support for tasks. For example, the user can use RPC or active messages to execute a function on a certain processor. Nonetheless, it still remains to be seen if X10 provides comparable performance, which is a must in order to become established in HPC.

Cilk There have also been efforts to extend the C/C++ language by task parallelism, resulting in the *Cilk* runtime [44], [45]. New tasks can easily be spawned by any thread, whereas data and task placement is handled by the runtime. Here, each thread possesses a work queue to implement load balancing techniques, for example work stealing.

Task parallelism with MPI Chatterjee et al. [46] propose an MPI extension to support task parallelism, which they refer to as *HCMPI*. It combines the *Habanero* task-parallel programming model [47], which itself origins from X10, with MPI. The results show that task parallelism improves scalability over an MPI+OpenMP implementation of the same benchmark, but also improves programmability.

CHAPTER 4

Exascale Application Analysis

The previous chapter presented various communication models for large-scale distributed systems. Here, MPI has become the de facto standard and can be regarded as the most prominent and most implemented CPU-centric model in HPC. Consequently, this chapter analyzes MPI applications in regard to their communication behavior and concludes with an overview of communication requirements of the training of deep neural nets. This study is particularly important as the size of high-performance systems keeps increasing and the number of computing nodes is projected to grow to about one million nodes for an exascale system in 2018 [11]. However, this has later been deferred to about 2023 [48].

Applications can be analyzed in various ways, for example by annotating the source code or reading performance counters of processor and networking hardware. While the first requires a basic understanding of the application and how it is implemented, the latter results in a fine-grain analysis, from which it might be difficult to derive general conclusions. Another approach is to intercept library calls. Here, a wrapper library is linked to the application and every call is logged before the routine is called from the original library. The meta data is then written to trace files, which can be parsed and analyzed later in time.

The U.S. Department of Energy (DOE) compiled a set of such MPI traces [49] for various applications, representing the kind of applications that are going to run on an exascale system. The following analyzes traces in regard to various characteristics, both considering point-to-point and collective communication semantics. The analysis has also been published at the *International Supercomputing Conference (ISC)*, 2017 [17][1].

4.1 Methodology of the Trace Analysis

Investigating applications in regard to their communication behavior requires the applications to be run and profiled accordingly. This includes running applications at different scales in order to consider scaling effects. Since access to a large number of nodes, even with access to a supercomputer, is limited, various academic institutions have provided traces of their most common applications. These traces are publicly available and provided in the *dumpi* format. The *dumpi* tracing framework is part of the Structural Simulation Toolkit (SST) [50], developed and maintained by the U.S. Sandia National Laboratories [2]. When linked against the *dumpi* library, any MPI call of the application is intercepted and logged, including call-specific meta data, such as message size, data type, source, destination, tag, and communicator or, for instance, the operation that is executed on the data by MPI's (all-)reduce operation.

The framework traces each rank separately and generates proprietary binary trace files, which can be converted to ASCII text files by tools the framework also provides. This is the first step of the analysis presented in this chapter. Next, the text files are parsed and any MPI call is stored as an event to a binary file. An event includes meta data as well as a time stamp of when the routine was entered and left, containing all the necessary information that is required for the following analyses.

While any characteristic of the communication analysis can be gathered by parsing the event file, it is mainly required for more complex analyses such as the determination of message queue lengths and search depths. Here, queues are reconstructed for any given rank. The length of the queue and the depth at which the match was found are logged, resulting in large outputs as the queues are searched at any occurring send/recv or *MPI_Wait* call.

A second approach is to parse the ASCII trace files directly. First, it allows to verify results from the event-based analysis, but secondly also allows for much

[1]The paper was nominated for the conference's best paper award
[2]http://sst-simulator.org/

faster processing as each file can be processed independently. This approach is sufficient for most statistics, except any metric that concerns UMQ or PRQ.

Nonetheless, the trace-based analysis has limitations, which leave some questions unanswered. For example, not all traces contain information on the composition of custom data types, which renders it impossible to determine exact message sizes. Furthermore, the traces only comprise MPI calls, which does not allow to assess overlap between computation and communication, for example. However, insights from the traces are still valuable for the remainder of this work.

It is worth noting that not all trace files comprise the entire application run time, but rather cover a single iteration. While it is explicitly stated that traces from the *Design Forward* program contain only a single iteration, other programs are silent on this matter.

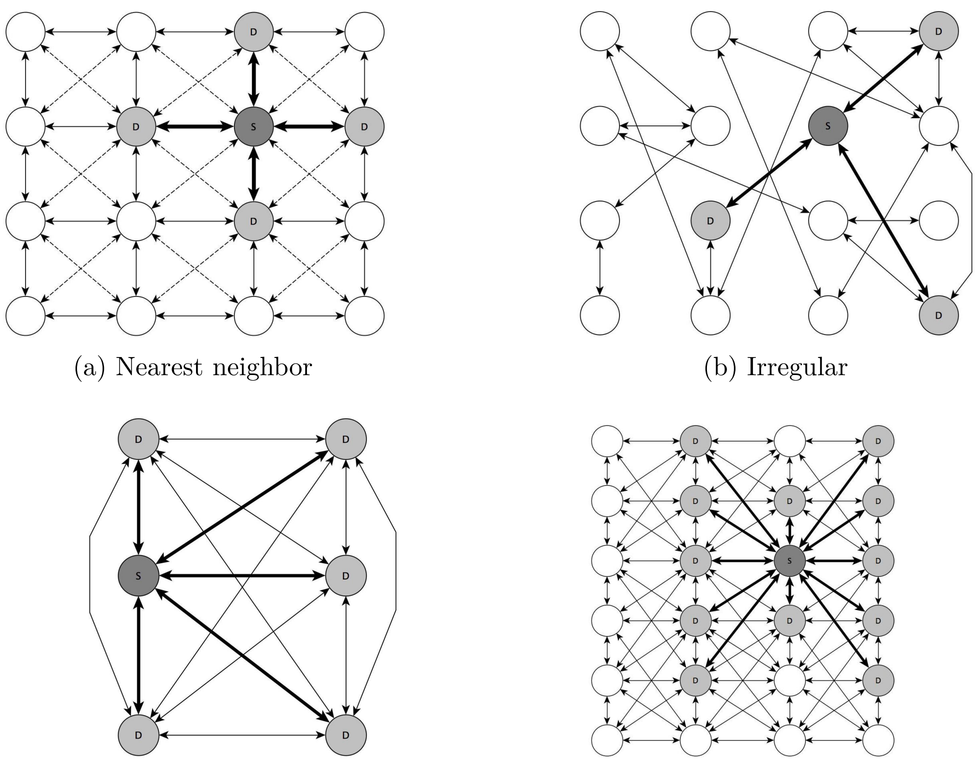
4.2 Overview of MPI Characteristics

Table 4.1 introduces the applications together with a short description and their underlying communication patterns. Interestingly, nearest neighbor communication seems to be the most prominent pattern. Examples for these communication patterns are depicted in Figure 4.1. The nearest neighbor pattern is not necessarily restricted to the immediate neighbor, but can also involve neighbors that are farther away. *Crystal Router* and *MiniFE* follow a *staged all-to-all* pattern, which is actually similar to many-to-many. A stage consists of a subset of processes that exchange data all-to-all. After each stage the subset comprises different processes.

Basic MPI characteristics are shown in Table 4.2 on page 61. As aforementioned, traces are available at various scale, allowing the analysis to also consider scaling effects. The number in brackets of the *Ranks* column indicates the number of threads each process is using, however, not all applications are multi-threaded.

4.2.1 Related Work

There are many papers on MPI characteristics of various applications, mostly focusing on the NAS Parallel Benchmark (NPB) suite. Faraj and Yuan [51] and Riesen [52] found that most applications heavily use point-to-point commu-



(a) Nearest neighbor (b) Irregular

(c) All-to-all (d) Many-to-many

Fig. 4.1 Overview of the most common communication patterns in scientific applications.

nication, whereas collectives contribute only 20% to the total number of MPI operations. Furthermore, messages sizes are rather small and never exceed 64kB on 64 nodes.

Similar metrics as used in this work have been analyzed by Kamil [53] and Vetter [54], who found that the number of peers with which a rank communicates and message sizes are rather small. However, they only looked at a small set of applications. Raponi et al. [55] studied MPI time and the number of calls to various MPI operations. Results regarding the transfer volume are similar to the findings of this work and they also pointed out that point-to-point communication dominates the data transfer. However, only small scale applications are studied.

This work and the analyses in the following are the first to look at exascale applications. Besides common MPI metrics, the queue data structures within MPI are also studied.

4.2.2 MPI and Communication Time

The first metric is the time an application spends in MPI routines, referred to as the MPI time. However, it can also be distinguished between MPI and communication time, which is the sum of all data transferring operations. The application time is determined by the last and first occurring MPI call in the traces. The traces of an application can be reduced to a time-sorted sequence $M = (m_0, m_1, ..., m_n) = (m_i)_{i \in I}$ with index set $I = \{0, 1, ..., n\}$, which contains all occurring MPI calls. Another sequence C only comprises communicating or synchronizing calls, such as send/recv, collectives, and wait(all). The index set J of sequence C is a subset of I.

$$\begin{aligned} t_{overall} &= t_{m_{|M|-1}} - t_{m_0} \\ t_{MPI} &= \sum_{m \in (m_i)_{i \in I}} t_m \\ t_{comm} &= \sum_{m \in (m_j)_{j \in J \subseteq I}} t_m \end{aligned}$$

Based on this, three metrics can be derived that provide insights on how much communication and overhead a given application contains. First, the MPI time is divided by the application time. This allows for an overview of how much time is spent inside the library. Second, the communication time is divided by the time between the first and last communicating MPI call, hence the first and last element of sequence C. The idea is to eliminate administrative MPI calls, such as *MPI_Init*, *MPI_Finalize*, or *MPI_Cart_create*. However, it is still not guaranteed that the first communicating call marks the end of initialization as this phase may also exchange data. However, it allows for a much better estimate than using the application time. Last, the administration share is determined by the administration time divided by the MPI time.

$$s_{mpi} = \frac{t_{MPI}}{t_{app}}$$
$$s_{comm} = \frac{t_{comm}}{t_{c_{|C|-1}} - t_{c_0}}$$
$$s_{admin} = 1 - \frac{t_{comm}}{t_{MPI}}$$

Strong scaling observations Table 4.2 shows the various metrics. It can be seen that both MPI and communication time increase relative to the application time as the number of ranks is increased (strong scaling) in the vast majority of applications. Exceptions are *MiniDFT* and *MiniFE*, in which the share of MPI decreases with the number of ranks. The MPI overhead, which is determined by one minus the communication time divided by the MPI time, seems to decrease at larger scale. As this analysis considers all messages of all ranks a larger scale means more communication, while initialization calls usually remain the same. Consequently, the relative overhead becomes smaller.

Weak scaling observations Weak scaling, in which the problem size is increased at a constant number of processes, shows that both MPI and communication time decrease accordingly. This is reasonable as the application is composed of communication and computation time and an increased problem size increases the amount of computation. However, this can also lead to larger messages or more messages to be exchanged. Hence, there are three consequences that can be observed in the case of weak scaling: first, the share of communication decreases, as applications show in Table 4.2. This is due to computational efforts increase more than the additional communication. Second, communication increases more than computation, thus the share of communication would increase. Third, the share remains the same and the increase in computation offsets the increase in communication. The latter two have not been observed in any application.

MPI overhead On average, an application spends 41% of its time in MPI routines, but only about 21% in communication- and synchronization-related operations. Applications with more than 1,000 ranks show an average MPI time and communication time of 60% and 28 %, respectively. The average MPI overhead amounts to about 33% of the presented MPI time. Some applications show an average overhead of about 90%, such as *Mocfe*. A closer look at the traces revealed that *Mocfe* spends about 70% of its MPI time in a single *MPI_Cart_create* call. As aforementioned, traces from the *Design Forward* program comprise only a single iteration and thus communication phases are much shorter than in the real application.

Unexpected messages Besides the time spent in MPI routines, the table also depicts the number of messages that arrive unexpectedly (see Section 3.2.1). Having too many unexpected messages poses a performance issue as queue structures and search times become large. On average, 36% percent of all message are unexpected, with none of the applications having less than 27% and more than 48%. These numbers mean that the majority of receive requests are already being pre-posted, as also shown in Table 4.2. In fact, almost all applications use non-blocking receive operations exclusively, however, it is not always guaranteed that these are executed before the arrival of the matching message. For instance, this would be required for ready-send operations (*MPI_Rsend*), which were not found except for the *MiniDFT* application.

The last column shows the number of peer processes any given rank is addressing with point-to-point messages. Interestingly, the numbers are rather small and indicate that point-to-point communication is rather local. This offers space for optimizations regarding the process mapping and topology of the network infrastructure. On average, a rank has 23 other ranks, with which it exchanges messages directly.

4.2.3 Message Size

As aforementioned in Section 4.1, the exact size of messages cannot be determined by analyzing *dumpi* traces as not all traces contain information on the composition of custom data types. Nonetheless, Figure 4.2 depicts the message size as elements per message, whereas the size of an element remains unknown for now.

It can be observed that point-to-point messages generally contain more elements than collective operations, which are often called and executed on a single data element. An exception is all-to-all, which shows a median message size of about 5K elements. However, this is mostly dominated by *BigFFT*, which heavily relies on this operation for its matrix transposes. In fact, the overall column of the graph shows the overall message size distribution, but

Table 4.1 Applications, for which traces are provided by the DOE. The programs, under which the applications are developed and maintained, are given in brackets.

Application	Description	Comm. Pattern
MOCFE (CESAR)	Neutronics code.	Nearest neighbor
NEKBONE (CESAR)	Fluid Dynamics code	Nearest neighbor
CNS (EXACT)	compressed Navier-Stokes eqn.	Nearest neighbor
MultiGrid (EXACT)	MultiGrid solver (BoxLib)	Nearest neighbor
LULESH (EXMATEX)	Hydrodynamic simulation	Nearest neighbor
CMC (EXMATEX)	Classic Monte Carlo	Nearest neighbor
AMG (DF)	Algebraic MultiGrid Solver	Nearest neighbor
AMR Boxlib (DF)	Adaptive mesh refinement	Irregular (sparse)
MiniAMR (DF)	Adaptive mesh refinement	Irregular (sparse)
BigFFT (DF)	large 3D FFT	Many-to-Many
Crystal Router (DF)	MPI many-to-many code	Staged All2All
Fill Boundary (DF)	Halo update (BoxLib)	Nearest neighbor
MultiGrid (DF)	MultiGrid solver (BoxLib)	Nearest neighbor
MiniDFT (DF)	VASP electron structure calc.	Many2Many
MiniFE (DF)	Finite element solver	Staged All2All
SNAP (DF)	Neural particle transport	Nearest neighbor
PARTISN (DF)	Neural particle transport	Nearest neighbor

certain workloads exchange far more messages than others and thus dominate this statistic. Nonetheless, due to the large number of applications it provides valuable insights on common message sizes.

Scaling observations Half of the applications show a decrease in message size of point-to-point messages when the scale is increased. On the other hand, messages of a few applications, namely *Crystal Router*, *MiniDFT*, and *DF MultiGrid* become larger at scale. *MiniDFT* shows the same behavior for the broadcast operation, which message size remains constant in every other application. A constant message size can also be found for point-to-point messages in *Mocfe*, *Fillboundary*, and *LULESH.* The most used collective operation is *(all-)reduce* with a constant message size in about two-thirds of the applications and an increase in message size in *AMR* and *MinDFT* again. Last, *all-to-all* messages become smaller in every application.

The fact that messages tend to become smaller at larger scale is less surprising as more ranks usually allow to decompose the problem size into smaller parts. For example, cells contain less elements in a fixed-sized grid if more processes

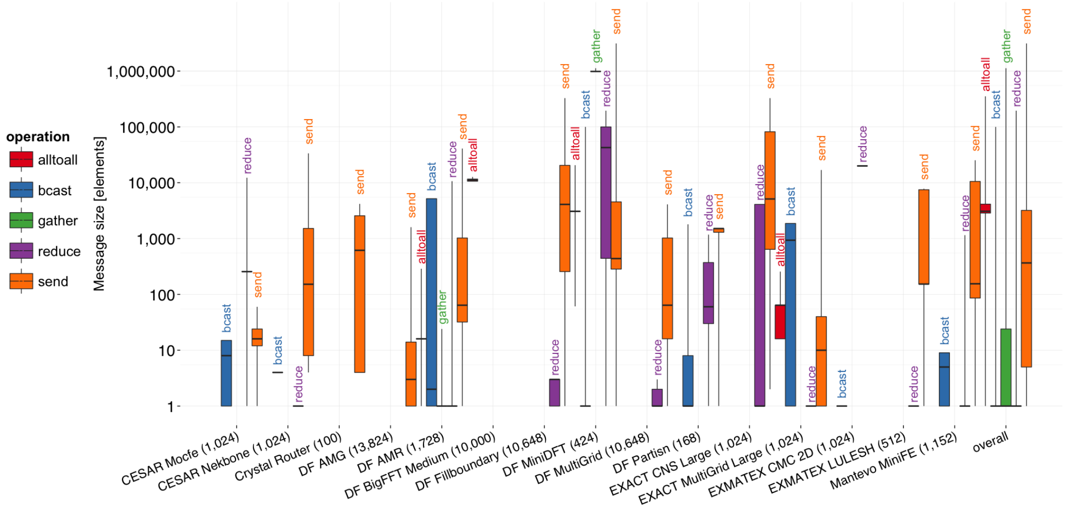


Fig. 4.2 Message size statistics for various applications and MPI operations [17]. The last column shows the average across all applications at varying scale.

are used for the decomposition. If the border elements of a cell are exchanged with neighbor processes, the resulting message also contains less elements. This is a common pattern as nearest neighbor communication is the predominant communication pattern in scientific applications.

Data type distribution While this analysis provides an idea on how many elements a message contains, it remains difficult to compare the results to other applications as no exact size can be determined. However, Figure 4.3 presents the data types that are used by any given application, comprising both point-to-point and collective operations. It is less surprising that most applications use *double* as their primary data type, but also user-specific types are widely used. It can also be seen that 32-bit types are used more frequent than the equivalent 64-bit types. While the majority of applications use the same types for point-to-point and collective operations, *Nekbone*, *Partisn*, and *FillBoundary* rely on different data types for the two kinds of messages.

Besides the size of the message it also important to consider how many messages are sent during an application's run time. There are two important characteristics: communication effort as number of elements sent during an application's run time and the rate at which messages are sent. Both aspects are discussed in more detail in the following.

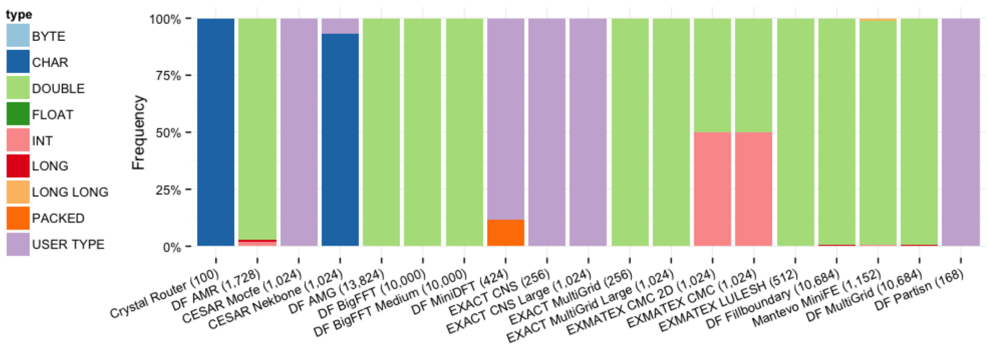


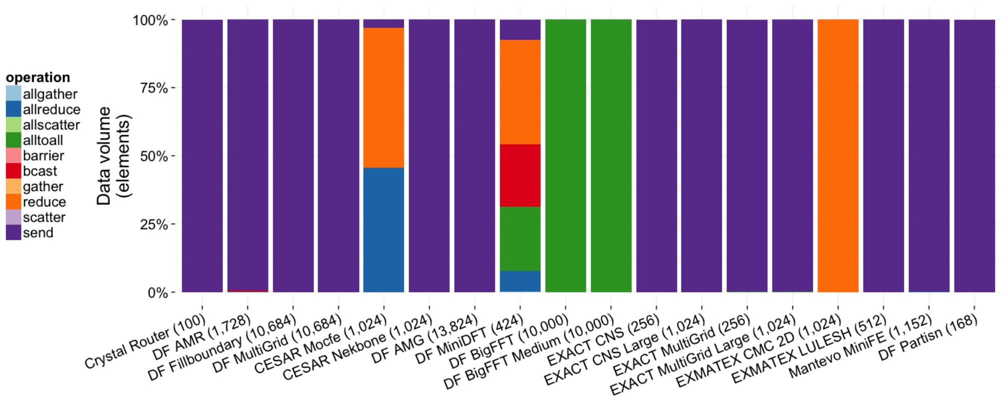
Fig. 4.3 Data type distribution for various applications, considering both point-to-point and collective operations [17].

4.2.4 Transferred Data Volume

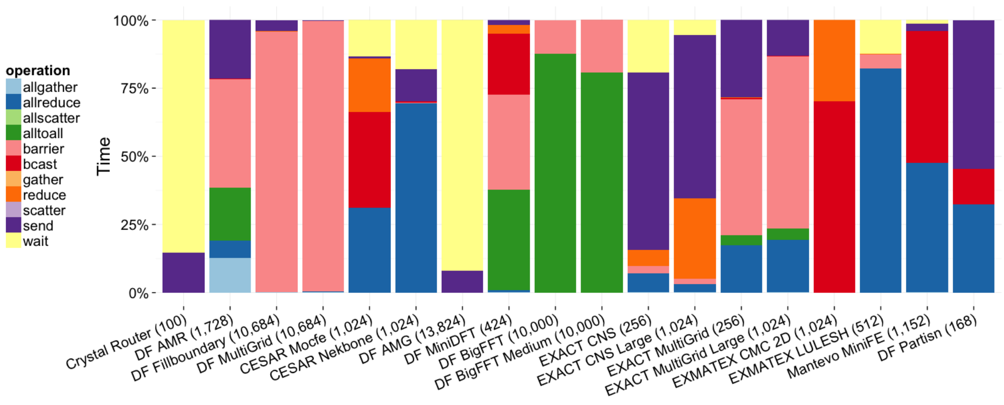
The most data per run time is exchanged by *Crystal Router* with about 5G elements/s, using 100 ranks. This is followed by *BigFFT* at 1,024 ranks and *Fillboundary* at 10,648 ranks with a rate of 3.3G elements/s each. Regarding communication time, the most data exchanging application is *BigFFT* with 112G elements/s at 10,000 ranks. The difference to the second place is quite significant as *AMG* (13,824 ranks) and *LULESH* (512) exchange data at 37G and 24G elements/s, respectively. The lowest observed rate is 1.2K elements/s, yielded by *CMC* at 64 ranks.

The data exchange rate can also be divided by the number of ranks, which provides insights on how much pressure a process is putting on the network infrastructure. As a result, *MiniFE* with 18 ranks shows the highest rate with about 900M elements/s with the communication time being the reference here. *LULESH* with 16 and *Crystal Router* with 10 processes achieve the second highest rate at about 400M elements/s. Interestingly, small scale applications tend to exchange more data per second and rank than large scale configurations, however, the absolute data volume is smaller as well. This might result from messages becoming smaller at large scale, which increases overhead and eventually reduces message rates. However, traces alone are not sufficient to prove this assumption and more profound analyses are required at this point.

The graph in Figure 4.4 depicts a breakdown of the data volume and time into various MPI operations. Except for a few workloads, most data is moved by send/recv operations, while the amount transferred by collectives is comparably



(a) transfer volume



(b) transfer time

Fig. 4.4 Data volume and transfer time for various applications and MPI operations [17].

small. *Mocfe* and *CMC* heavily use (all-)reduce operations to exchange data and *BigFFT* relies almost entirely on all-to-all communication. *MiniDFT*, on the other hand, uses all-to-all, broadcast, and reduce almost equally. Interestingly, the time spent in these operations shows different results. While most data is sent by send/recv, the most time is spent in collectives in many applications. Only *Crystal Router*, *AMG*, *CNS*, and *Partisn* show a large amount of time spent in point-to-point communication, of which *Crystal Router* and *AMG* do not even use collectives. It seems that inherent synchronization of collectives significantly adds to the latency, making them a top priority for optimizations. That being said, it is less surprising that the barrier and collectives that broadcast their results, which contain an "all" prefix, show the highest latencies of all operations. Especially at large scale, small imbalances can significantly slow down the application when many processes need to be synchronized.

4.2.5 Message Rate

Another important metric is the rate at which messages are injected into the network. This metric is especially important for small messages as it indicates the latency of the whole networking subsystem. A simple approach would count all messages during the application's run time, however, it would not reflect the demands regarding the network at certain communication hot spots within an application. For example, the need for high message rates might be significantly higher during communication phases than during computation. Consequently, here we measure the message rate by imposing a time interval and counting all messages that occur during that time. Figure 4.5 shows the results of this analysis. The boxplots show the message rate across all applications, while the orange dots represent the mean count of messages for particular time intervals.

The message count for small time intervals is almost negligible for most applications as only a mean of 2 messages are exchanged within an interval of $1\mu s$, for example. The count steadily increases for longer time intervals, but even considering an interval of $1ms$ does not result in more than 50 occurring message transfers, translating to a median message rate of 25K messages/s.

Another aspect is that the message rates for point-to-point messages are higher than for collectives, presumably due to the collectives' implicit synchronization.

However, message rates can be limited by various factors, such as the end-

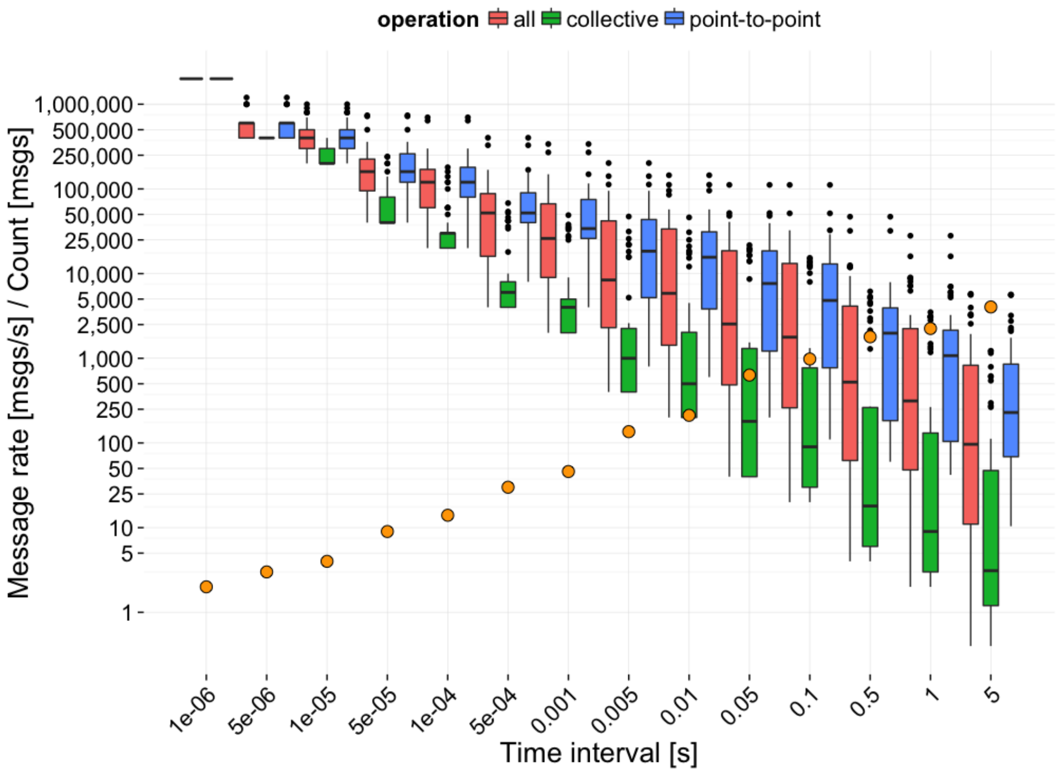


Fig. 4.5 Message rate for certain time intervals across all applications [17]. The orange points indicate the number of messages that are counted during the particular time interval.

point's injection bandwidth or the network throughput itself. Furthermore, it depends on the message size since small messages can usually be transferred at higher rates due to the available bandwidth. Considering a time interval of $100\mu s$, the median message rate amounts to 100K messages/s with a maximum of 500K messages/s. Taking an average message size of 1K elements into account, it appears that a throughput of 400MB/s for single-precision data and 800MB/s for double-precision data is achieved. This equals the PCIe 2.0 bandwidth for that particular message size [56], indicating that message rates are limited by the endpoint's injection bandwidth and PCIe interface. Although this trace analysis cannot prove this statement, PCIe is a bottleneck especially with multiple MPI processes sharing a single NIC. For example, the current PCIe 3.0 standard allows for a maximum unidirectional bandwidth of 16GB/s with 16 lanes, which is shared among all processes that use the same NIC.

4.3 Message/Receive Request Matching

MPI is a prime example for two-sided communication and as such it requires messages to be matched with receive requests at the targeted endpoint. The matching is crucial for the latency and message rate of small messages and highly optimized in all MPI implementations. For example, although MPI calls their data structure for unexpected messages and posted receive requests queues, they are actually implemented as lists due to the list's superior time complexity of the *remove* operation. Removing an element at an arbitrary position withing a queue requires all following elements to be moved one position back, while single-linked lists only require two pointers to be changed. On the downside, finding an element withing the list requires pointers to be chased, whereas memory can be traversed linearly within a queue.

Generally, there are three major aspects regarding the matching: the matching performance, lengths, and search depth of UMQ and PRQ, respectively.

4.3.1 Related Work

Similar to MPI metrics, existing work has focused on the NPB suite. Brightwell et al. [34] reported queue lengths for UMQ and PRQ and concluded that the UMQ length amounts to about 200 entries for applications with up to 140 processes.

However, their analysis also showed that the PRQ is always significantly smaller and search depths never exceed 30 elements. The authors explicitly say that a study of real applications is necessary, as opposed to analyzing benchmark suites.

Large-scale applications were studied by Keller et al. [57], who found that the UMQ length scales linearly with the number of processors for a thermodynamics application. Other applications showed queue lengths in the range of 10 to 30 elements.

4.3.2 Matching Rate

The matching performance is measured in matches per seconds and depends on the matching criteria, the protocol's constraints, and the implementation, such as whether a queue or list is used as primary data structure. In MPI, the matching process has to ensure that messages stay in order and wildcards are supported, thus limiting the choice of data structures.

The benchmark to determine the matching rate has two processes running on the same node, in which one process is referred to as the *sender* and the other as the *receiver*. The sender sends a particular number of messages using the *MPI_Send* routine, followed by a barrier. After the barrier, the receiver posts *MPI_Recv* operations. The time it takes to receive all messages yields the matching time. The fact that both processes run on the same node and are synchronized by the barrier before receiving the messages, minimizes the transfer times. It can be assumed that messages are already placed in the appropriate UMQ when the receiver starts posting receive requests.

The tags used in *MPI_Send* and *MPI_Recv* determine the order in which the messages are consumed. The best case scenario means that the tags are incremental and identical for both send and receive operations. Matching messages for receive requests are then always found at the queue's head. Alternatively, *MPI_ANY_TAG* can be applied too. The worst case, on the other hand, reverses the order of the tags at the receive side, thus matching messages are always found at the tail of the queue. Last, randomly shuffling the order of the tags used in receive requests reflects the average case.

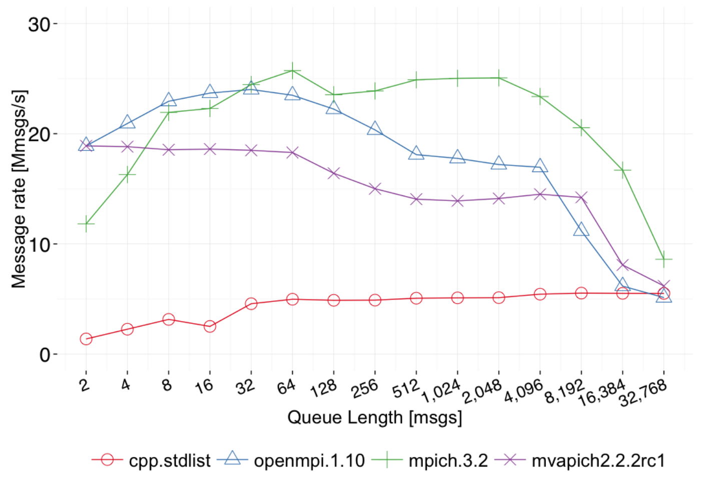
Figure 4.6 depicts the results of this benchmark for different MPI implementations and C++11's list from the Standard Template Library (STL), which serves as a reference to show how much MPI's lists are optimized for the matching process. The MPI implementations are OpenMPI 1.10 [28], MPICH 3.2 [29], and MVAPICH 2.2.2rc [30]. All of them are in wide use and well known. The test system's processor is an Intel Ivy Bridge Xeon E5-2630 with 2.60 GHz core frequency and DDR3 memory at 1600 MHz. Note that the matching's linear time complexity scales well with the core frequency and thus faster single-thread performance has strong impact on the matching rate.

Regarding the best case matching performance, MPICH is superior most of the time, except for queues shorter than 32 elements. The performance remains constant at about 25M matches/s until a queue size of 2K, but starts declining from there until it reaches about 8M matches/s at 32K elements. OpenMPI and MVAPICH perform similar and show the same trends, but at a lower peak performance than MPICH. STL's list, on the other hand, only performs comparably at queue sizes of 16K and larger.

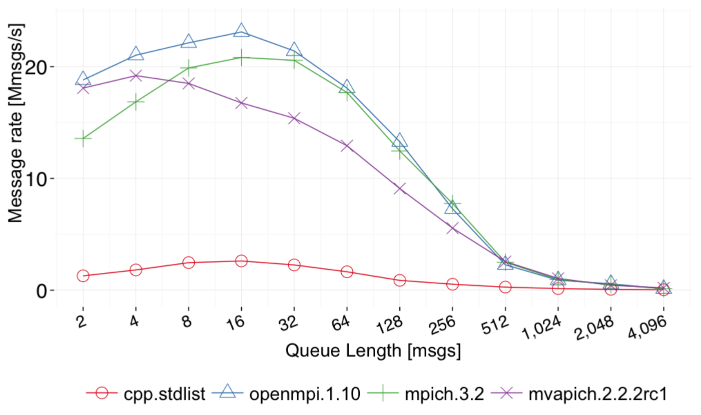
The matching rate of different MPI implementations becomes more similar in the average and worst case scenarios, while STL's list keeps performing significantly worse. The performance in the average case starts to drop at 18 elements and reaches a matching rate of less than 10M matches/s at a queue length of 256. At 1K elements, the matching rate is already below 2M matches/s. The average case is more representative than the best case as it better reflects real applications. The worst case shows a similar trend, but with lower absolute matching rates.

4.3.3 Queue Characteristics

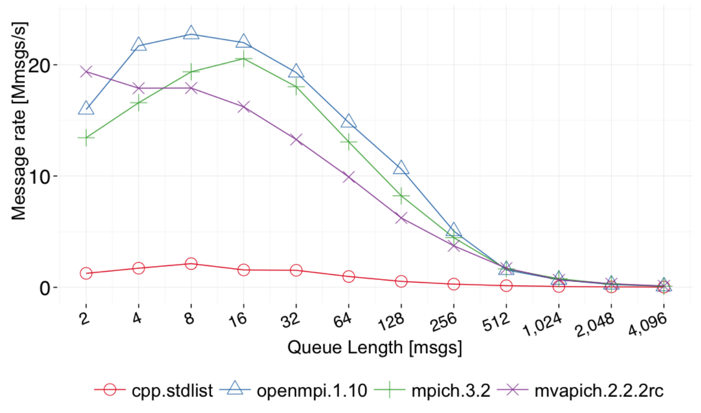
As shown, the matching performance strongly depends on the length of the queue and search depth. For example, if matching messages are always found near the head of the queue, the actual queue length has no impact on performance, but still increases the memory footprint. In order to determine queue lengths and search depths, the queues need to be reconstructed from the trace files. Here, the analyzing script looks at every occurring send/recv and wait routine. For example, a new arriving message has to search the PRQ for a matching receive. If a match is found the receive request is removed from the queue and the length and search depth are written to a file. This continues for every call, resulting in large outputs with the length and depth after every event. However, queues tend to be quite small toward the end of the application and they are often even



(a) best case



(b) average case



(c) worst case

Fig. 4.6 MPI's matching rate for synthetic input data, measured for the best, average, and worst case on the CPU [17].

zero in length. This has an impact on the statistics that are reported in the following and it should be considered that queue length and search depth values may be higher during communication-intensive periods. The results of the queue analysis are shown in Figure 4.7.

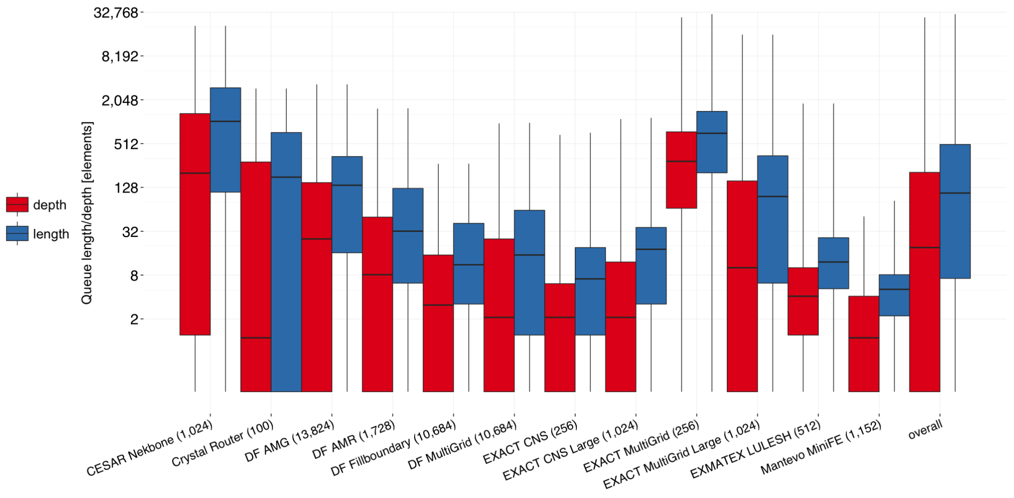
The length of both UMQ and PRQ is similar for most applications, although the UMQ tends to become slightly larger. The queues of *Nekbone* and *MultiGrid* are the longest with a median length of 1,024 elements. This is quite long compared to the overall median length of 128 and a 3$^{\text{rd}}$ quartile of 512.

It is particularly interesting to compare search depth and queue length to identify possible issues with the order in which receive requests are posted. If the search depth is similar to the actual length of the queue then matches tend to be found toward the queue's tail, increasing matching latency and overhead. On the other hand, if the search depth can be kept small, the actual length has no impact on the matching performance and only increases the memory footprint of the MPI library. As the results show, the UMQ search depth is never significantly lower than the length, which is the case for the PRQ in many applications. It suggests that the order in which messages arrive match the order of posted receive requests. Another reason can be that multiple receive requests match the same message, for example if the same tag is used and messages originate from the same process.

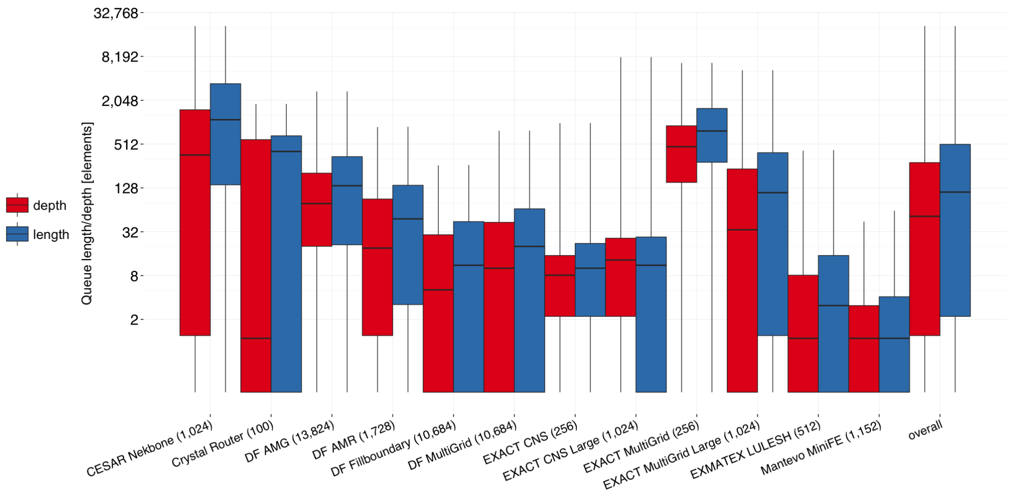
Combining results from the matching rate experiment with the queue analysis shows that a matching rate of about 18M matches/s is achieved 50% of the time, which is about 70% of the peak matching rate. This performance might be even lower during communication-intensive periods as queues tend to be longer. Furthermore, the analysis shows that queues tend to become longer at larger scale. This becomes even more challenging as messages are also becoming smaller and thus the actual data transfer time is low compared to the matching overhead.

4.4 Trace Analysis Conclusion

Various applications have been analyzed in regard to MPI and general communication characteristics. This is important as application developers can follow a similar methodology to identify possible bottlenecks and performance issues, but also learn from the conclusions this analysis is drawing. For example, applications mostly use a single MPI communicator, whereas multiple communicators would



(a) PRQ



(b) UMQ

Fig. 4.7 Length and search depth of UMQ and PRQ [17]. The last bar on the right considers all messages from all applications and configurations and not only the configurations shown as the other bars.

allow for better matching performance as they can be matched independently in parallel. Also, it has been shown that collective operations cause significant overhead due to their synchronization aspect. A better approach would be to use non-blocking collectives as introduced with MPI 3.0 [58].

More importantly to this work, this analysis helps system designers and architects to understand what applications are demanding from the system in regard to communication and message passing. For example, messages tend to become smaller at large scale and thus it seems beneficial to optimize large-scale systems for small message exchanges. This especially renders the matching important and raises the question of whether processors and network controllers could support this even further. It also remains an open question whether MPI's protocol still requires complex guarantees such as wildcards and ordering, or if this can be relaxed to allow for better performance.

Aspects of the trace analysis, especially the message matching, are going to be picked up again later when GPU-centric (Chapter 5) and managed communication (Chapter 6) are discussed in more detail.

4.5 Communication in Deep Learning Applications

The previous sections analyzed traditional HPC workloads that rely on MPI and the CPU. Although these applications represent the vast majority of HPC workloads today, advances in artificial intelligence are made so quickly that current HPC facilities adapt their systems to better support deep learning applications at large scale. Current and future cloud installations are even adapting their systems at a much faster rate. Consequently, exascale systems will not only host traditional scientific applications, much rather they will also be used to train large neural nets [59], [60].

Although machine learning is a wide field with various approaches for different use cases, deep learning has been the main focus and driving force of artificial intelligence lately. Similar to HPC applications, the training of deep neural nets require vast amounts of computational power, but also large amounts of data, on which the network is trained. The main reason why deep learning has received tremendous attention lately is two-fold. First, the computational power provided by recent systems, especially GPUs, has reached a level that makes large nets trainable and second, the available amount of data is seemingly endless and grows rapidly every second. With deep and powerful neural nets being able to be trained now, many applications have been found with more and more showing up every day. Main applications today include image classification, object recognition, speech recognition, and language translation.

The following presents a brief background on neural nets and their training. Then, the parallel training is described and analyzed with respect to communication and synchronization. More details can be found in the book of Goodfellow et al. [61].

4.5.1 Background

As opposed to analytical approaches, in which there is a concrete and well-defined relationship between input and output, a neural net needs to be trained with data to develop this relationship. Neural nets are comprised of so-called *neurons*, which are organized in layers. Neurons take the weighted output of other neurons from previous layers as their input and apply an *activation* function to generate their output. A general and simple net is shown in Figure 4.8. The input vector x, which is the output of the previous layer, is multiplied with the weight matrix W_0, to which a biased b is added also. The results are passed to the activation function of the neurons:

$$a_i = f\left(W_{i-1} \cdot a_{i-1} + b_i\right) \;\; i \in Layers \;\text{ with }\; f(t) = \frac{t}{\sqrt{1+t^2}}$$

The activation function is usually a logistic function, such as the sigmoid function shown in the formula. At the beginning of *supervised training*, the weights, or sometimes also referred to as *parameters*, are usually initialized with small random values. The result of the output layer is then compared to the expected result and a *loss* is calculated. Next, the gradient is calculated and backpropagated to the input so that each layer can adapt its weights accordingly. The next time the network sees a similar input the output will be closer to the desired result. The training needs to be repeated with enough data to eventually yield the desired accuracy or it ends when the network has stopped learning from the data. Instead of learning on each example individually, examples are aggregated into *batches* of size N examples. This improves efficiency as matrix-

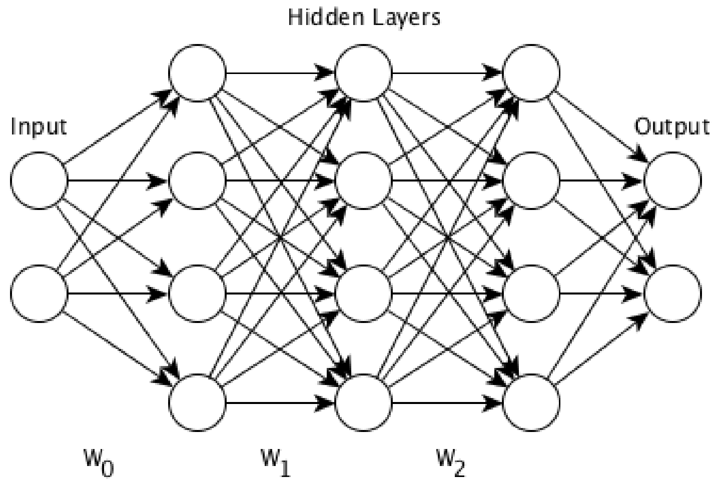


Fig. 4.8 Example of a neural net with one input, one output, and three hidden layers in between. All layers are fully-connected.

vector operations are replaced by more efficient matrix-matrix operations and furthermore, weights are updated $N-1$ less times.

Fully-connected layers The layers shown in Figure 4.8 are referred to as fully-connected as any neuron's output is influenced by every neuron's output of the previous layer. While computing the output of a fully-connected layer is relatively fast, the memory requirements are quite high due to the large amount of weights. The computational effort F in FLOPS for layer l with Ψ neurons and a batch size of N is given by:

$$F_{FC} = N \cdot \Psi_l \cdot (2\Psi_{l-1} - 1)$$

The memory requirements M for a floating point representation of weights and activations (four bytes), on the other hand, are given by:

$$M_{FC} = 4 \cdot \Psi_l \cdot (\Psi_{l-1} + N)$$

The impact of the batch size on the performance is made clear by these formulas. With small batch sizes, the arithmetic intensity, which is calculated by the number of FLOPS divided by the number of memory operations, is also small and thus overall performance is limited by the memory bandwidth. The performance limits can be determined by the *roofline model* [62] and although GPUs provide a high bandwidth memory system, the raw compute power is still much higher.

Convolutional layers While fully-connected layers are one type of many, another important type is the *convolutional layer*. An common application of deep learning techniques is image recognition and classification. However, as the image resolution increases the number of weights also increases dramatically with fully-connected layers and thus *overfitting* quickly becomes an issue. Overfitting occurs when the model is too complex with respect to the number of observations within the data. Eventually the model learns from noise rather than from relationships within the data.

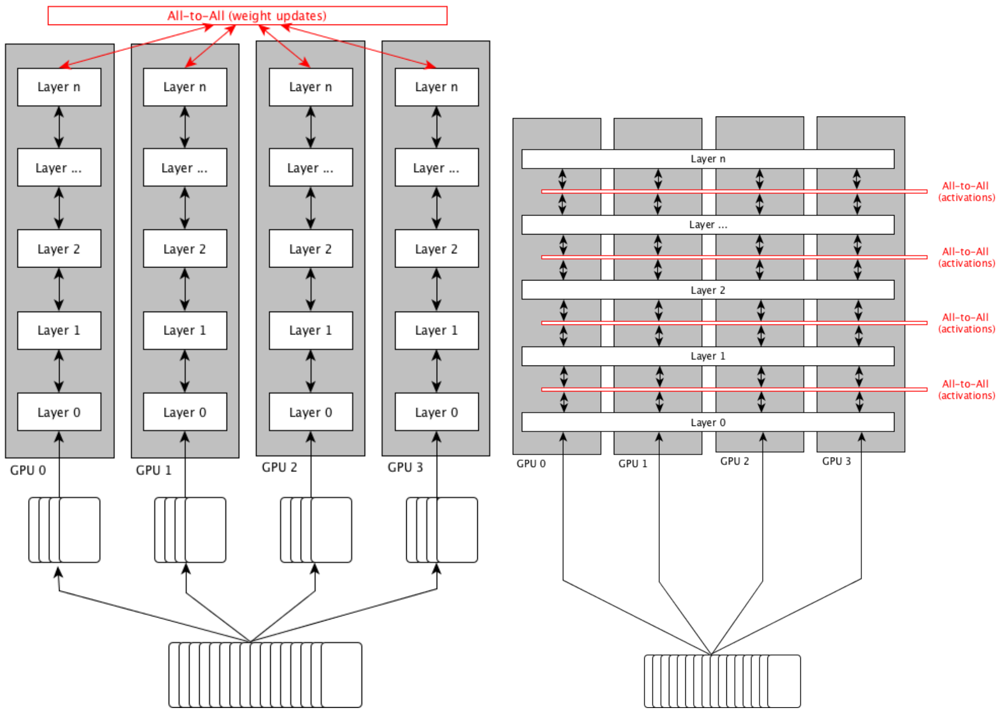
As opposed to fully-connected layers, neurons in convolutional layers do not connect to all neurons of the previous layer, but exploit spatial locality by only connecting to a neighboring subset, commonly referred to as a *kernel*. Kernels share weights with other kernels in order to reduce the number of parameters to be learned, rendering convolutional layers less prone to overfitting. Due to the large numbers of neurons grouped into kernels, the output of convolutional layers is much larger as well, which is why these layers are usually followed by *pooling* layers. The pooling takes a neighboring subset of neurons and reduces their output according to a particular math operation, for example four neurons' activations are reduced to the maximum value only. The memory requirements M for a convolutional layer with χ kernels of size υ are given by:

$$M[B] = 4[B] \cdot (\underbrace{\nu \cdot \chi}_{W} + \underbrace{\nu \cdot \frac{\Psi}{\chi}}_{x} + \underbrace{\chi \cdot \frac{\Psi}{\chi}}_{a}) = (\nu \cdot \chi + \tfrac{\Psi}{\chi}(\chi + \nu)) \cdot 4[B]$$

This results in the following arithmetic intensity λ for convolutional layers :

$$\lambda_{conv} = \frac{N\Psi(2\upsilon - 1)}{4(\chi\upsilon + N\frac{\Psi}{\chi}(\upsilon + \chi))}$$

One of the first convolutional networks that are trained on GPUs is *AlexNet* [63], whose convolutional layers have an arithmetic intensity of 49 to 91 for a batch size of 1, and 62 to 171 for a batch size of 256. This shows how computational intensive convolutional layers are, compared to fully-connected layers, whose arithmetic intensity ranges from 97 to 117 for a batch size of 256, but only amounts to 0.5 for a batch size of 1.



(a) Data parallelism (b) Model parallelism

Fig. 4.9 Principles of data and model parallelism. The bottom shows a batch of training examples.

4.5.2 Parallel Training

The following elaborates on parallelism and communication during the training.

4.5.2.1 Data Parallelism

The idea of data parallelism is simple as the same neural net is trained on multiple processors. The batch of input data is divided among the processors as depicted in Figure 4.9a. During training, weights of the network are exchanged between all processors after each training iteration to update local weights accordingly, allowing to apply what other processors have learned so far. The updates can be performed either synchronously or asynchronously. While the first requires updates to be completed before the network is trained on the next batch of data, the latter continues with the next batch even though updates from other processors have not arrived yet.

A common implementation is the *parameter server* [64], which is a centralized processor that holds the weights of the network. Before each training iteration, the participating processors fetch weights from the server and send their updates back after they completed the training iteration. Then, weights are fetched again before the next iteration begins. Again, processors may continue with the training even though not all updates have been applied to the weights. Sometimes asynchronicity even helps with the convergence of the training as random noise adds to the regulation. However, with an increasing number of processors asynchronous data parallelism might not be applicable anymore as too many updates are getting lost [65]. This is known as the *stale gradient problem*. Thus, synchronous data parallelism could become mandatory at larger scale.

The batch size is another issue that can prevent data parallelism from scaling to a large number of processors. While the local batch size, meaning the batch a single processor is training the network on, should be kept moderate to increase computational efficiency, the effective batch size is the sum of all local batch sizes. For example, if two processors are given a batch of 64 images to train the network, the effective batch size is 128. Consequently, the effective batch size increases linearly with the number of processors, but cannot be chosen arbitrarily large at the same time as it eventually prevents convergence [65].

The most important communication routine during the data parallel training is allreduce with the payload being the weights of the network. Since the same model is replicated on each processor, the payload remains constant with an increasing number of processors. Table 4.3 on page 62 shows the data parallel training using the *caffe* framework on up to eight Kepler K80 GPUs.

While communication efforts increase with the number of GPUs, the time to complete computation decreases. Consequently, the training quickly becomes dominated by communication even at rather small scale.

4.5.2.2 Model Parallelism

The second parallel training approach is referred to as *model parallelism*, in which activations are exchanged among processors after each layer. Unlike data parallelism, communication must be synchronous as succeeding layers depend on the activations of the preceding layer for their calculation of activations. The principle is illustrated in Figure 4.9b.

There are two approaches to implement model parallelism and distribute the model among GPUs. First, layers can be cut through and the more processors are used the smaller the part of the layer per processor. Thus, the overall communication payload remains constant with an increasing number of processors.

While the computation time also decreases at increasing scale, model parallelism seems to scale well in theory.

A second approach assigns each layer to a GPU. When a layer is computed, the activations are passed to the next GPU and the next batch of training data is fetched. This follows a pipeline scheme and is supported by *mxnet* [66], for example.

With current systems lacking support for GPU-controlled communication, the necessary communication between layers has to be controlled by the CPU, hence control is given back and forth between CPU and GPU. The resulting overhead due to copy operations and context switches prevent model parallelism to be used at larger scale. However, the latest Pascal GPU architecture enables fast data exchange between GPUs via NVLINK, enabling model parallelism for GPUs within a node. It still remains to be seen how future systems support this principle and how far it scales in practice.

4.5.2.3 Hybrid Parallelism

As aforementioned, fully-connected layers result in many weights to be trained but the number of neurons and thus activations is comparably small. On the other hand, convolutional layers share weights, but the large number of neurons lead to many activations to be fed into the next layer. Since data parallelism exchanges weights and model parallelism exchanges activations, the type of parallelism can be varied depending on the layer.

Hybrid parallelism applies data parallelism to the convolutional layers, which are placed at the beginning of the neural net, and model parallelism is applied to fully-connected layers, which conclude the neural net. The main advantage of this approach is that communication can be minimized and also well overlapped with computation.

However, the main disadvantages of data parallelism still apply as the batch of training data needs to be divided across all processors.

4.5.3 Comparison and Verdict

The parallel training of neural nets seems to perform well at weak scaling, but struggles at strong scaling. Both types of parallelism, data parallelism as well as model parallelism, rely on collective communication primitives, such as allreduce, broadcast, and barrier synchronization. Point-to-point communication, on the other hand, is rather irrelevant.

In order to scale the training to a large number of GPUs it is necessary to implement fast and low-overhead GPU-to-GPU communication as CPU-controlled communication prevents the application from applying model parallelism. While data parallelism, especially asynchronously, allows to scale efficiently to a small number of processors, a large scale application seems unfeasible with current systems and algorithms. The batch size cannot be chosen arbitrarily large as too many updates are getting lost, eventually hurting convergence of the training. While hybrid parallelism suffers from this as well, model parallelism seems to be better suited to scale to a larger number of processors.

Table 4.4 on page 62 shows example architectures of widely used neural nets, which will be compared using data and model parallelism. While *AlexNet* [63] has many trainable parameters, the number of neurons is comparably low. Most parameters are provided by the fully-connected layers. *VGG* [67] is a much deeper net with many more parameters and neurons, requiring the most computation for the forward pass. *GoogLeNet* [68] uses a different kind of layers, so-called inception layers. These are comprised of convolutional and pooling layers. Although only 22 layers provide parameters, the entire net consists of 100 layers. The last example net is *ResNet* [69] with 152 layers and almost as many neurons as trainable parameters. This is achieved by heavily relying on convolutional layers.

The amount of data transferred per GPU for data and model parallelism is shown in Table 4.5. As can be seen, the data volume is mostly higher for data parallelism at larger scale as weights need to be exchanged, which is independent of the number of GPUs. For model parallelism, the data volume per GPU becomes smaller at increasing scale as activations are distributed across all processors. Furthermore, most neural nets have less activations than trainable parameters. The number in brackets shows the average amount of bytes per layer that need to be exchanged per GPU. Nonetheless, at a scale of 64 GPUs and a batch size of 256, messages can still become as large as 14MB (VGG) with a total transfer volume of 224MB per GPU.

A simplified model is used to evaluate data and model parallelism on two different systems. First, it is assumed that all GPUs are fully interconnected by bidirectional links. Second, a ring communication is assumed. Other assumptions are summarized in the following:

- Compute and communication time only comprise forward pass without backpropagation.
- The batch size is set to 512.
- The GPU has a performance of 9.3 TFLOP/s (e.g. P100) with a network bandwidth of 80 GB/s (e.g. NVLINK).
- Computation is reduced linearly by the number of GPUs and peak performance and bandwidth are always achieved.

The serial computation time is calculated by the batch size N times the number of floating point operations F divided by the GPU's performance P in floating point operations per second. The parallel time takes the number of GPUs n into account, so one has:

$$t_{serial} = N \cdot \frac{F}{P}$$
$$t_{parallel} = \frac{t_{serial}}{n} + t_{comm}$$

The communication time depends on the type of parallelism and the network architecture. For a net with weights W and activations A, represented by 4B float values, and a network bandwidth of B, one has:

	Data Parallelism	Model Parallelism
Fully-connected	$t_{comm} = 4\frac{W}{B}$	$t_{comm} = 4\frac{N}{n} \cdot \frac{A}{B}$
Ring	$t_{comm} = 4(n-1)\frac{W}{B}$	$t_{comm} = 4\frac{N(n-1)}{n} \cdot \frac{A}{B}$

The parallel efficiency p is then given by:

$$p = \frac{t_{serial}}{n \cdot t_{parallel}} \cdot 100\%$$

The parallel efficiency that is achieved with both types of parallelism and varying scale is reported in Table 4.6. As shown, GoogLeNet and ResNet both scale well with data parallelism and a fully-connected communication. The ring communication scales significantly worse, but the bandwidth utilization could be significantly improved by pipelining. Nonetheless, model parallelism scales well for AlexNet and VGG, but shows worse performance for GoogLeNet and ResNet. The reason for that can be found in the nets' architectures. AlexNet and VGG both use fully-connected layers at the end, which provide many parameters but only a few activations. Thus, their weight-to-activation ratio is higher than GoogLeNet and ResNet, whose nets are mainly composed of convolutional layers. As a result, the large number of activations increase the amount of communication for model parallelism.

It is also important to notice that computation does actually not decrease linearly with a reduced batch size. For example, if a batch of 512 elements takes $10ms$ for the forward pass, a batch of 256 takes more than $5ms$ [70]. However, this is difficult to model and was neglected for brevity.

The issue with model parallelism on current systems is the frequent and synchronous data exchange which takes place after each layer. Computing the training on the GPU but relying on the CPU to exchange activations comes along with too much overhead, caused by both context switches and data copies. This issue is being tackled by the just recently introduced NVLINK network that allows a small number of GPUs to exchange data at high bandwidth and comparably low latency. Consequently, upcoming systems might apply model parallelism to GPUs that are connected by NVLINK on node level, while many of these nodes are using data parallelism to train a network.

It is important to mention that research in this field is highly active at the moment which affects both system architectures and algorithms. Thus, it is tough to predict how systems are going to change and how the training will be executed.

4.6 Insights

The insights of the application analysis are summarized in the following:

Insight I: Communication in scientific applications and training of deep neural nets is structured and regular, with nearest neighbor communication being the most common pattern. The training of neural nets relies entirely on collective operations, mostly allreduce and broadcast.

Insight II: The number of ranks any process is communicating with using send/recv semantics is rather small. This makes communication rather local and selective.

Insight III: Send/recv accounts for the most data by volume, while collectives account for the most time spent in MPI. Especially at larger scale, implicit synchronization of collective operations significantly penalizes load imbalances.

Insight IV: Message sizes are rather small and decreasing at strong scaling. Two-thirds of all point-to-point messages across all applications are smaller than 5,000 elements per message. That's about 40KB when `double` is the assumed data type.

Insight V: The matching of messages and receive requests can mostly be performed with queues smaller than 512 entries. Only a few applications occasionally exceed this size. However, queues might become long during communication-intensive parts of an application.

Insight VI: Deep learning might soon explore more model parallelism approaches as nets become larger and cannot be fit into the GPU's memory. This renders GPU-controlled communication necessary in order to allow for low-latency communication and synchronization.

Insight VII: Data and model parallelism are equally important and their application depends on the type of net. For example, deep convolutional nets might benefit from data parallelism, while mixed nets with fully-connected layers prefer model parallelism.

Insight VIII: While data parallelism relies on large messages in an allreduce pattern, model parallelism results in many small messages that are exchanged in an all-to-all pattern.

Table 4.2 Basic MPI characteristics of various Exascale proxy applications [17].

Application	**Ranks**	s_{MPI} (s_{admin})	s_{comm}	**Unexpected Messages**	**Non-Blocking Send/Recv**	**Peers**
MOCFE	64	74 (89) %	25 %	n/a	100/100%	2
	256	86 (93) %	25 %	n/a	100/100%	3
	1,024	92 (90) %	30 %	n/a	100/100%	4
NEKBONE	64	11 (37) %	12 %	40 %	99.9/99.9%	18
	256	34 (68) %	18 %	35 %	99.9/99.9%	8
	1,024	78 (70) %	73 %	45 %	99.9/99.9%	29
CNS	64 (1)	3 (18) %	5 %	28 %	0/92.5 %	26
	256 (1)	22 (24) %	14 %	40 %	0/98.5 %	44
CNS Large	64 (4)	3 (2) %	4 %	30 %	0/60.8 %	26
	256 (1)	11 (9) %	9 %	27 %	0/85.7 %	20
	1,024 (1)	43 (10) %	39 %	34 %	0/98.4 %	72
MultiGrid	64 (1)	6 (61) %	3 %	27 %	0/100 %	14
	256 (1)	16 (23) %	12 %	47 %	0/100 %	37
MultiGrid Large	64 (1)	3 (53) %	2 %	40 %	0/100%	14
	256 (1)	5 (45) %	9 %	31 %	0/100 %	17
	1,024 (1)	22 (16) %	39 %	33 %	0/100 %	20
LULESH	64 (4)	1 (36) %	17 %	21 %	100/100%	14
	512 (2)	8 (4) %	13 %	29 %	100/100 %	19
CMC 2D	64	76 (0) %	27 %	n/a	n/a	n/a
	256	78 (0) %	29 %	n/a	n/a	n/a
	1,024	84 (1) %	34 %	n/a	n/a	n/a
AMG	216	3 (0) %	33 %	44%	100/100%	57
	1,728	1 (0) %	39 %	46 %	100/100 %	79
	13,824	0 (0) %	44 %	48 %	100/100 %	92
AMR Boxlib	64	9 (47) %	6 %	27%	0/99.9%	18
	1,728	12 (17) %	15 %	37 %	0/99.9 %	35
BigFFT	100	99 (98) %	91 %	n/a	n/a	n/a
	1,024	99 (97) %	98 %	n/a	n/a	n/a
	10,000	99 (99) %	99 %	n/a	n/a	n/a
BigFFT Medium	100	72 (60) %	53 %	n/a	n/a	n/a
	1,024	81 (77) %	63 %	n/a	n/a	n/a
	10,000	99 (99) %	72 %	n/a	n/a	n/a
Crystal Router	10	23 (0) %	27 %	31%	0/100%	3
	100	63 (0) %	64 %	46 %	0/100 %	6
Fill Boundary	125	40 (31) %	28 %	34%	0/100%	16
	1,000	52 (15) %	51 %	30 %	0/100 %	20
	10,648	72 (2) %	82 %	32 %	0/100 %	23
MultiGrid	125	40 (57)%	32 %	41%	0/100%	14
	1,000	66 (12) %	70 %	39 %	0/100 %	10
	10,648	70 (2) %	85 %	38 %	0/100 %	8
MiniDFT	125	15 (1) %	15 %	n/a	32/3.4%	19
	424	11 (0) %	11 %	n/a	31.3/2.2 %	30
MiniFE	144	7 (5) %	12 %	n/a	0/100%	12
	1,152	7 (4) %	6 %	n/a	0/100 %	15
PARTISN	168	51 (3) %	61 %	n/a	0/0%	1
Average	n/a	41 (33) %	35 %	36%	n/a	23

Table 4.3 Training of AlexNet using different numbers of GPUs. The training is based on synchronous data parallelism and is executed on Kepler K80 GPUs.

	2 GPUs	**4 GPUs**	**8 GPUs**
Effective batch size	64	32	16
Communication/batch [ms]	61	183	427
Computation/batch [ms]	760	488	355
Computation/Communication	12.48	2.66	0.83
Speedup without communication	1.73	2.70	3.72
Speedup with communication	1.60	1.97	1.69

Table 4.4 Example architectures of commonly used neural nets.

Net	**Parameters**	**Neurons**	**Layers**	**FLOPS**
AlexNet [63]	60M	0.7M	8	0.72G
VGG-16 [67]	138M	14M	16	15.3G
GoogLeNet [68]	6.8M	4.5M	22	1.5G
ResNet-152 [69]	26M	20M	152	11.3G

Table 4.5 Communication size for various nets and parallel approaches. The numbers represent the amount of data per GPU and the number in brackets provides the average amount of data per layer and GPU. All numbers are in Bytes and a batch size of 256 is assumed.

	Data Parallelism			**Model Parallelism**		
GPUs:	**4**	**16**	**64**	**4**	**16**	**64**
AlexNet	240M	240M	240M	179M (22M)	45M (6M)	11M (1M)
VGG	552M	552M	552M	3,584M (224M)	896M (56M)	224M (14M)
GoogLeNet	27M	27M	27M	1,152M (52M)	288M (13M)	72M (3M)
ResNet	104M	104M	104M	5,120M (34M)	1,280M (8M)	320M (2M)

Table 4.6 Parallel efficiency in percent [%] of the example nets at scale with data and model parallelism.

	Data Parallelism						**Model Parallelism**					
GPUs:	**16**		**64**		**256**		**16**		**64**		**256**	
Topology:	FC	R	FC	R	FC	R	FC	R	FC	R	FC	R
AlexNet	45	5	17	0	5	0	70	14	70	4	70	1
VGG-16	90	35	67	3	34	0	70	14	70	4	70	1
GoogLeNet	94	50	51	6	48	0	42	5	42	1	42	0
ResNet-152	97	67	88	11	65	1	55	8	55	2	55	1

Chapter 5

GPU-centric Communication Methods

The previous chapters have introduced communication models and provided an overview of MPI characteristics of various applications intended to run on large scale computing systems. This chapter discusses communication from a GPU-centric view, thus data transfers between accelerators and independent of the CPU. Various communication models are considered to analyze which one complements the GPU's execution model best.

The remainder of the chapter is structured as follows. The first section provides some background information, including a review of communication offloading and onloading concepts and other work in this area. Also, typical network architectures are presented. This is followed by different GPU-centric communication models and their interactions with the network. The last section presents benchmarks and results in terms of performance and energy efficiency.

This chapter extends and summarizes various contributions to international conferences and workshops [18]–[20], [71].

5.1 Background

Before specialized communication models are presented later in this chapter, some background information is provided in this section. It begins with communication offloading and onloading concepts and continues with the introduction of three different network architectures, namely Infiniband, EXTOLL, and NVLINK.

5.1.1 Review of Communication Offloading and Onloading Approaches

Similar to how computation is offloaded to specialized and highly parallel accelerators, communication can also be offloaded to specialized hardware. Here, a NIC can perform copy operations while the CPU only needs to trigger and complete transfers. The tighter integration of the network interface and the processor has led to a debate on whether to offload communication, mainly between processor and network vendors such as Intel and Mellanox. The following introduces both principles and reviews current research in this area.

5.1.1.1 Offloading

Offloading communication to dedicated hardware aims to reduce communication costs and increase processor availability. The NIC takes over several tasks from the processor, including data copy, address translation, message matching, and accelerated synchronization primitives like barriers, atomics, and locks.

The data copy operation performed by the NIC is executed through Direct Memory Access (DMA). The NIC has direct access to the processor's memory and receives source and target addresses from the processor, embedded into so-called work requests. As soon as the work request is handed off to the NIC, the processor can either continue with other tasks or idle until it is woken up by the NIC again.

Work requests contain information on where the data is currently located and to where it needs to be copied. For message passing, the NIC is told the source address and a destination node identifier. The NIC reads the data and sends network packets to the destination NIC, which either copies the data directly to the user buffer or caches the data in a system buffer until the user address comes to be known.

One-sided communication is different in that the source processor passes source and destination address to the NIC. The NIC translates the addresses into network addresses and the destination NIC translates network addresses to local physical addresses and copies the data directly to user space. In order to perform the address translation, memory regions have to be registered in advance.

Besides receiving work, the NIC also needs to notify the processor about new data available or successfully processed work requests. These signals, generally referred to as notifications, can be an interrupt to wake up the processor or another approach that could let the processor poll on Memory Mapped I/O (MMIO) addresses provided by the NIC. The latter generates significant Input/Output (I/O) traffic and can be optimized by replicating the I/O registers in system memory, which are then held in the processor's first level caches. This minimizes polling latency and reduces I/O and memory system traffic.

Including the NIC in the system's coherence domain has been common especially in Hyper Transport (HT) systems. As opposed to PCIe, HT's protocols is designed for processor interactions, naturally optimized for coherency traffic.

If the NIC is part of the coherence domain it can also update the caches directly, hence called Direct Cache Access (DCA) [72]. This approach is beneficial for network I/O intensive applications with spatial and temporal relationships between processor and I/O memory accesses. Depending on the workload and based on 10Gbps Ethernet, Intel reported speedups of about 15 and 40%s for DCA. This study is motivated by the argument that a simple NIC is sufficient if the processor, a CPU in this case, has sufficient capabilities for communication on-loading, replacing expensive and feature-rich NICs.

5.1.1.2 Onloading

Onloading communication means that communication tasks are performed by the processor as opposed to offloading these operations to other hardware entities. This is mostly relevant to systems with tight network integration and short data paths between the processor and the network. Prominent examples include Intel's *OmniPath* for CPUs [73] and NVIDIA's *NVLINK* [10] for GPUs. However, both vendors follow different strategies as Intel integrates an Infiniband-like network access and NVIDIA allows GPUs to directly access other GPUs' memory, thus implementing a shared memory model. Although the following examples implement message passing, other communication models could also be on-loaded onto the processor.

Vaidyanathan et al. [74] studied an on-loading approach for MPI. In a multi-threaded MPI application, one thread is dedicated to communication and handles all related tasks, while other threads are free to perform computation. A lightweight lock-free command queue is used to instruct the communication thread about messages to be sent or received. Both, blocking and non-blocking calls are supported. In case of a blocking call, the application's compute threads poll on a flag, which is set by the communication thread upon completing the associated MPI operation. Non-blocking calls, however, are processed differently. First, compute threads submit requests to the command queue and receive pointers to *MPI_Request* structures in return. These structures are allocated from a pool and it is required to dynamically allocate as many structures as non-blocking calls are invoked. Progress is guaranteed by the communication thread. In order to assess performance, three different HPC applications, namely Quantum Chromodynamics (QCD), Fast Fourier Transformation (FFT), and a Convolutional Neural Net (CNN), are benchmarked and compared to state-of-the-art multi-threaded MPI. The maximum speedup for QCD is 1.2x, 1.3x for FFT, and up to 2x for the CNN. While their approach is never slower, almost no performance increase is observed at small scales, e.g. less than 32 nodes.

Similarly, Lai et al [75] proposed *ProOnE*, an onloading protocol for multi- and many-core architectures. Unlike the aforementioned approach, ProOne dedicates a subset of cores to communication, executing a daemon process. This suggests two communication paths: intra-node communication between the application and the daemon process, and inter-node communication between ProOne processes. While inter-node communication is implemented in MPI, intra-node communication uses queues again, one for send and receive requests each. The matching is augmented by a sequence number and destination rank to avoid false matching. This is necessary as ProOne sends additional control flow messages, which can mix up the ordering of MPI messages. However, they do not support wildcards with that approach. Results from two benchmarks, namely a 2D Jacobi sweep (stencil code) and a matrix multiplication, show that ProOne allows for better overlap between computation and communication, improving performance by about 2.5x for the Jacobi sweep and 1.2x for the matrix multiplication.

5.1.1.3 Technology Trends

Directing the focus on communication is important in such parallel computing environments as we find them in HPC, for example. Interconnection networks are becoming faster and faster and PCIe, for instance, cannot keep up with bandwidth and latency. As a result, processor vendors increase the effort toward integrated solutions, in which processor and NIC share the same package [73]. However, the main communication processing is delegated to the processor itself, augmented with networking hardware to accelerate distinct networking tasks. This leads away from typical network offloading strategies, arguing the need for complex NIC extension cards. On the other hand, networking equipment vendors like Mellanox argue that offloading communication is beneficial in most cases to maximize overlap between communication and computation [76]. In their opinion, processor availability is crucial to many applications and requires dedicated networking hardware.

Although there is no ground truth of what is the better approach, trends point toward many core processors, in which some cores can be easily dedicated to communication tasks. The main issue with peripheral network cards is the data path between the processor and the NIC. PCIe, for example, seems to limit the node's network injection bandwidth. A 16x wide PCIe 4.0 interface provides 32GB/s per NIC, whereas an Infiniband HDR link provides 6GB/s, adding up to about 72GB/s for an Host Channel Adapter (HCA) with 12 links. With CPUs implementing more and more cores, the demand for network injection bandwidth per processor is increasing significantly, thus questioning whether a PCIe-based NIC is sufficient to satisfy these growing demands. Certainly another aspect is memory bus bandwidth, which needs to gain upon bandwidth demands. For example, *DDR4* at 2,133MHz provides up to 25GB/s per channel, thus limiting overall transfer bandwidth. Here, memory technologies like High Bandwidth Memory (HBM) and Phase Change Memory (PCM) seem promising to increase capacity and bandwidth significantly.

Summarizing it can be said that a tighter coupling of processors and networking hardware is inevitable to cope with increasing demands of network performance. Thus, many communication tasks will be performed on general purpose processors, shifting much of the functionality from hardware to software. In fact, Chapter 6 of this work elaborates on the GPU's capabilities to manage communication.

5.1.2 Network Architecture

The communication architecture comprises many layers, implemented in hardware and software. The NIC is one of the most important hardware components, while the operating system and communication library are essential to the communication software stack. David Culler et al. [77] describe the communication

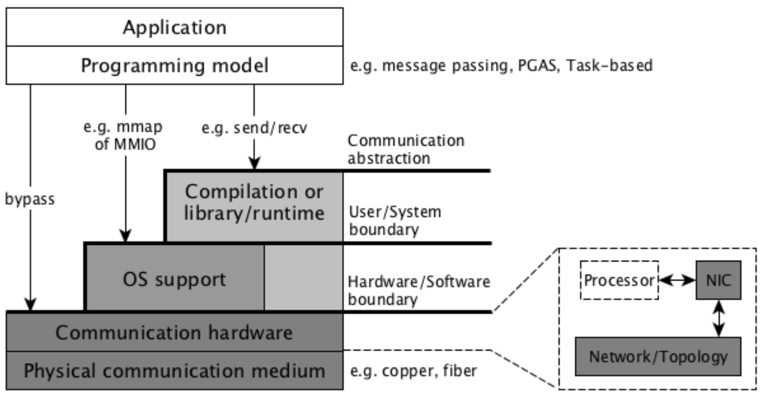


Fig. 5.1 Generic communication architecture in accord with David Culler et al. [77].

architecture as depicted in Figure 5.1.

The top level is represented by the application and programming model, accessing the system through a communication abstraction. The abstraction varies and depends on the communication model and system. For example, a message passing model on a distributed memory system offers send/recv operations to exchange data, while the application can rely on put/get operations in a PGAS model. The abstraction is an interplay between the hardware, the operating system, and the communication library.

The library varies between communication models and builds upon the operating system and the hardware. The level of abstraction is a trade-off between usability and performance. Higher abstraction means more generalization with less control given to the user and thus less application specific optimizations. However, less abstraction leaves many decisions up to the user who must consider system specific peculiarities to tweak the application and system for the best performance.

Another trade-off must be made at the hardware/software boundary. Modern NICs provide many capabilities to support the runtime with specialized hardware, reducing software overhead. However, due to integration and shorter paths between processor and NIC, some functionality is also moved back to the processor by an onloading protocol. Hence, the software/hardware boundary is shifted toward the bottom as more functions are performed in software.

The following elaborates on the various layers and components in more detail. Note that communication and programming models have been discussed in more detail in Chapter 3.

5.1.2.1 Generic NIC Architecture

Traditionally, computing systems implement a processor, memory, and a network controller which is handling communication. The network controller is provided access to the system's memory through DMA. It is important to assess where to attach the network controller and how to access it. Many modern processors even integrate the network controller into the processor.

The NIC is an important component to scale systems to hundreds and thousands of nodes. It accelerates communication and increases processor availability by allowing the CPU to offload data transfer and synchronization tasks to specialized hardware. It serves as interface to the network from the processor's point of view and must provide fast access and delivery of both small (e.g. control) and long (e.g. bulk data) messages. Besides low-overhead access and high-throughput message transfer, the NIC has to ensure that multiple processes can independently access the NIC without notable performance or security complications [78, section 20.1].

The placement of the NIC in the system is an important design choice. In order to minimize latency it is recommended to attach the NIC as close to the processor as possible. In the simplest approach, the processor sees two registers: an input and output register, referred to as two-register interface [78, section 20.1.1]. Reading and writing these registers de- and enqueues from the message queues, residing on the NIC. Longer messages are sent by writing word by word to the output register, prohibiting the NIC to access memory directly through DMA. Another issue is multi-process support, since one processor could be delayed infinitely and not write the end of the message to the output register, tying up resources and blocking the network interface.

The latter issue can be resolved by dedicating a set of registers to the network interface. The message is written to multiple registers and when the data transfer is triggered by the processor, the NIC reads the message registers atomically and starts processing the message. While this renders the message atomic, a DMA transfer is still not supported by this approach.

Instead of placing the message in registers and having the NIC picking it up, the processor can directly write instructions to the NIC. The instructions are embedded in descriptors, which contain the type of command, addresses, and immediate data. Small messages, for example, can be sent by embedding the data into the descriptor and writing it to the NIC. Larger data, on the other hand, cannot be embedded into descriptors and addresses are passed to the NIC, which reads the data from the memory directly through DMA. This approach supports multiple processors and also supports DMA transfers, increasing processor availability by offloading communication to the NIC.

In the following, three different NIC architectures are introduced, of which two are used in the remainder of this chapter to evaluate GPU-centric communication models. The third, NVLINK, has just been released and thus it was not possible to evaluate its performance and architecture in this work. However, this would only affect raw performance numbers as the same principles are still evaluated with EXTOLL's Shared Memory Functional Unit (SMFU) later in this chapter.

5.1.2.2 Infiniband NIC

Infiniband [79] is a standard that was compiled by a consortium of leading computer hardware vendors to replace both Peripheral Component Interconnect (PCI) and Ethernet in high-performance computing systems. While Intel stepped back and focused on the development of PCI, Infiniband has replaced Ethernet in many computing systems with low latency and high bandwidth requirements.

Today, the vast majority of Infiniband implementations are indirect and switch-based networks, with fat tree and dragonfly being widely implemented topologies.

The processor interface is based on the descriptor approach as mentioned in section 5.1.2.1. Descriptors are written to queues by the processor. The NIC, often referred to as HCA, queries the queues for new entries to receive work and writes to a queue to signal completion of work items.

The system integration and processor interface are depicted in Figure 5.2. The NIC is attached to a PCIe switch, which is connected to the CPU's root complex. This is a common setup, especially in accelerated systems in which the PCIe root complex cannot provide enough lanes for all peripheral devices, such as accelerators and network interfaces.

During initialization, a so-called Queue Pair (QP) is allocated, which contains one queue for send and receive work requests. If the processor wants to send data to another node, for example, it creates a work request (descriptor) and enqueues it into the send queue. The NIC does not query the queue periodically, so that the processor needs to write to the doorbell register in order to signal the NIC

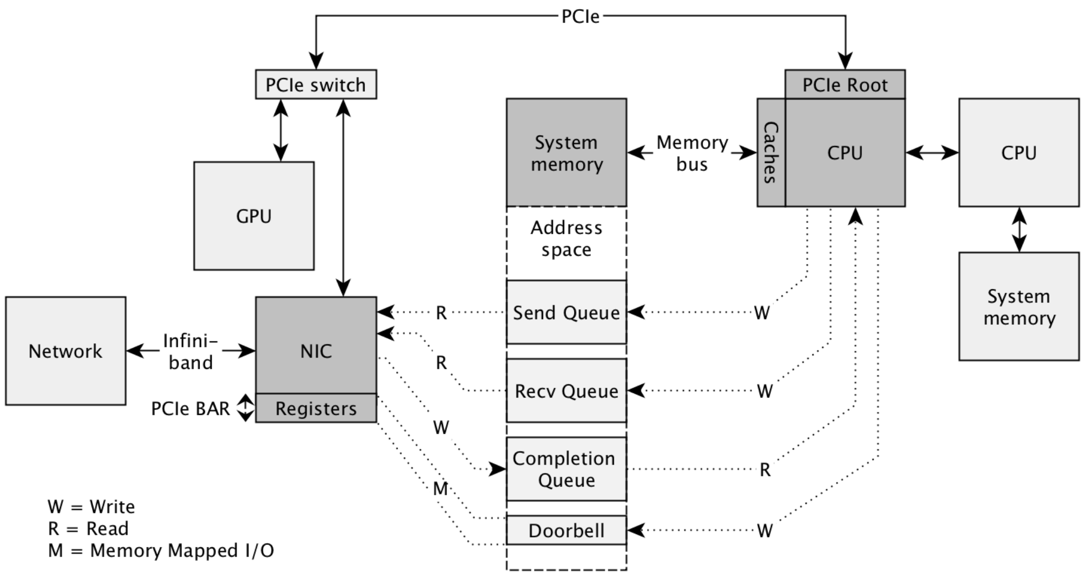


Fig. 5.2 Infiniband NIC system integration and processor interface.

that new work items are available. Upon ringing the doorbell, the NIC reads and executes the work request from the send queue through DMA. After completion, a notification is generated by the NIC and enqueued into the completion queue, which is frequently queried by the processor. Receiving data follows the same procedure as work requests are first enqueued into the receive queue, followed by writing the doorbell register.

Separation of submitting and execution of work requests allows for a few optimizations. One is that the NIC can fetch multiple work requests from the queue, increasing efficiency of PCIe transfers. Furthermore, it reduces the memory footprint on the NIC and avoids polluting the PCIe network by polling on system memory for new work requests becoming available in queues. Nonetheless, Infiniband supports work requests that are directly written to the NIC, which is referred to as 'blue flame'. However, this is only applicable when the embedded data is rather small and only a few requests are issued.

The actual location of the QP is flexible and could be either system or accelerator memory. From the NIC's point of view it does not matter since queues are accessed through PCIe in either case. On the other hand, accelerators can map the doorbell register in their address space to trigger data transfers without the interference of the host processor. This principle will be shown and evaluated later in the course of this chapter. More on Infiniband itself can be found in [79].

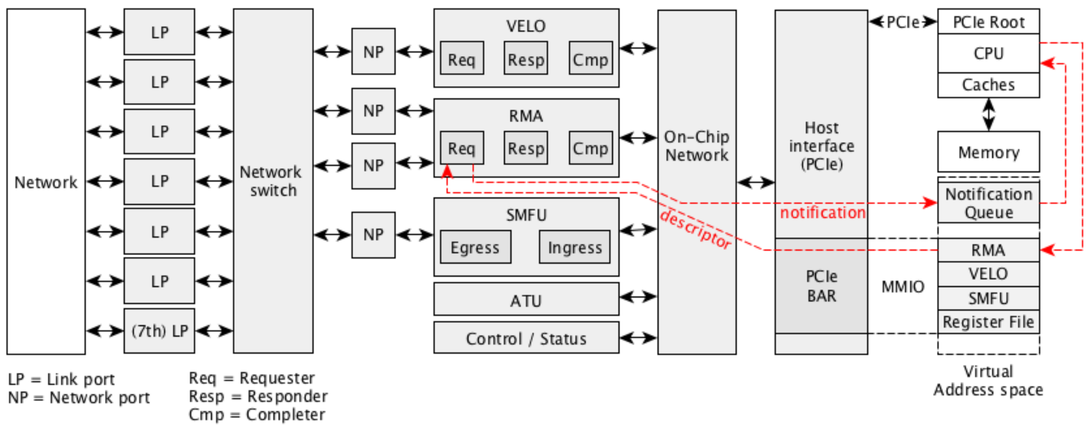


Fig. 5.3 The EXTOLL NIC architecture, system integration, and processor interface. An example Remote Memory Access (RMA) access with requester notification is given with the dotted red lines.

5.1.2.3 EXTOLL NIC

While Infiniband is a well-established and known high-performance interconnect, EXTOLL started off as a research project at the Ruprecht-Karls University Heidelberg (Germany). The first Application Specific Integrated Circuit (ASIC), called Tourmalet, was released in 2016. This section introduces the technology and focuses on the differences compared to Infiniband, for example.

The most obvious difference to Infiniband is the direct network approach, in which the NIC also implements switching logic. The processor interface is, just like Infiniband, descriptor-based. The NIC receives work requests from the processor and handles the communication independently. However, unlike Infiniband, work requests are written to the NIC directly as opposed to queues, allocated in system memory. The system integration and interface is show in Figure 5.3.

An EXTOLL NIC provides seven link ports to connect to other NICs, although the seventh is intended for network attached memory or accelerators, leaving six link ports free to build a 3D torus network. The link ports are connected to a crossbar that either routes messages between the link ports, or from the network to the functional units.

The NIC comprises three major functional units: RMA unit, Virtualized Engine for Low Overhead (VELO) unit, and SMFU, each supporting different communication models. This is complemented by the Address Translation Unit (ATU), which translates between a node's physical address and the Network

Logical Address (NLA).

RMA As the name suggests, the RMA [80] unit provides an interface for one-sided communication. There are two different addressing modes that can be selected: physical and network addresses. While it should be avoided to use physical addresses in user space, memory regions can be registered with the ATU to obtain a network address that can be used for RMA. When an NLAs is obtained, the memory region is pinned so it cannot be swapped out of the main memory. Furthermore, the use of network addresses ensures inter-process security.

During initialization, the work request queue, which resides on the NIC, is mapped to the processor's Virtual Address Space (VAS). Unlike Infiniband, a doorbell is not required and the NIC begins with the execution of work requests as soon as the last word of the descriptor is received. Notifications, on the other hand, are written to a queue that resides in system memory to have them closer to the processor that is polling on pending notifications. Consequently, the design keeps I/O traffic to a minimum.

Descriptors comprise 192-bit and are written as three 64-bit words to the NIC. Upon receiving the last word the work request is executed. Besides source and destination address, the descriptor contains the size, destination node ID, a command specifier, and a Virtual Process ID (VPID).

Notifications are composed of two 64-bit words. They are written to a double-linked list structure, which is accessed like a queue, in system memory by the NIC. The processor can query the queue for new notifications, but has to explicitly remove it from the queue. Notifications have to be consumed in a First In First Out (FIFO) order and cannot be taken from arbitrary positions within the list.

The RMA unit is capable of generating different notifications for various events. For example, a notification can be generated when the NIC has processed the work request or when new data is received. This mechanism allows to generate notifications at both sending and receiving side. Possible combinations are shown in Table 5.1 [81]. The RMA unit itself is comprised of three sub-units: requester, responder, and completer. The requester receives work requests from the processor and generates network packets that are delivered to the receiver. The responder creates responses at the receiving side. For example, a 'get' operation requires the requester to send a request to the receiving side's

Table 5.1 Possible combinations for the generation of notifications [81]

Command	Requester	Completer	Responder	
PUT	0	0	0	No notifications
	1	0	0	One-sided PUT
	1	1	0	Two-sided PUT
GET	0	0	0	No notifications
	0	1	0	One-sided GET
	0	1	1	Two-sided GET

responder, which responds to the request with the data. The completer executes DMA transfers on both sending and receiving sides, depending on where the data is written to memory. For example, the completer in a 'put' operation is only involved on the receiving side and writes the data to the receiver's memory.

The capability of each sub-unit to generate notifications allows to mimic various communication models with the RMA unit. While traditional one-sided communication would only require notifications from one sub-unit, a message passing scheme can be implemented with notifications being generated on both ends.

SMFU While most interconnection networks implement hardware to support one- and two-sided communication semantics, EXTOLL's SMFU [82] provides hardware support for a virtual shared address space across a physically distributed system. PCIe packets are forwarded through the network, allowing to virtually access remote devices.

Like other functional units, the SMFU provides a PCIe Base Address Register (BAR), which can be mapped into the processor's VAS through MMIO. The BAR is divided into intervals that can be individually configured and be assigned to a specific node within the network. If the processor issues a store operation to an address within an interval, for example, the SMFU forwards the PCIe packet to the target node. Upon receiving the network packet the target SMFU looks up the physical address assigned to the interval the network packet is addressing. The store operation is then executed on that address with the offset given by the received network packet. A load operation works similar and a response is sent back to the source.

The SMFU unit has many advantages as it allows to apply a shared memory programming model to cluster systems. Remote node's memory appear in the VAS as if they were part of the system's local address space. However, NUMA effects have to be taken into account, rendering data locality important. This particularly applies to CPUs, which cannot tolerate long latencies as well as GPUs. For example, a database application that uses the SMFU is presented in [83].

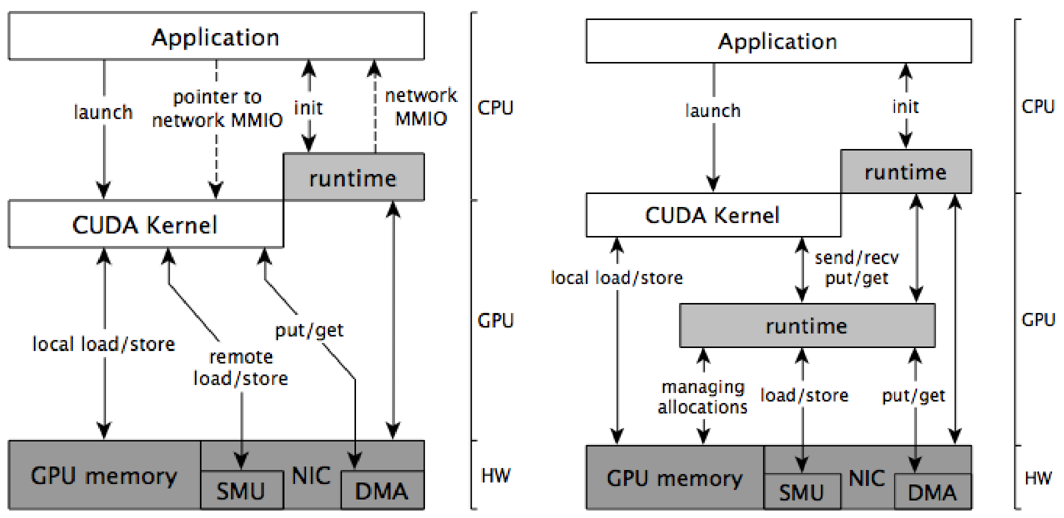
A common issue is that accesses to false addresses can have the system to deadlock. For example, if one node issues a load to a false address and waits for the response that will not be generated on the target side. Consequently, the user has to be careful to ensure addresses are valid. Recovering from deadlocks requires the system to restart.

5.1.2.4 NVLINK

NVIDIA's NVLINK has been designed and proclaimed to be a replacement for PCIe and it's the first network that is optimized and designed for GPU-to-GPU communication. Although it aims to replace PCIe, it is still required for control flow while data is transferred by NVLINK. Nonetheless, less PCIe lanes are needed and thus enabling a higher numbers of GPUs per CPU without additional switches.

In NVLINK, the NIC is integrated into the GPU and attached to the High Speed Hub (HSHUB), which also connects the PCIe interface and the copy engines to the internal crossbar. Similar to PCIe-based GPU systems and UVA, a single large address space is globally shared between all GPUs. This allows for fine-grain communication on a small granularity with load/store operations or high-bandwidth bulk transfers using the GPU's copy engines [10]. Besides load/store operations, NVLINK also supports atomic operations and memory fences, enabling large SMP-like clusters of GPU.

In terms of network architecture, NVLINK can be seen as a direct network with point-to-point connections to other GPUs. Each P100 Pascal GPU provides four links with a bidirectional bandwidth of 80 GB/s. NVIDIA's example configuration in [10] shows eight GPUs with four GPUs being fully connected. It says that "[...] each half of the 8-GPU Hybrid Cube Mesh can operate as a shared memory multiprocessor, while the remote nodes can also share memory with DMA through peers" [10]. It seems that GPUs cannot route traffic and



(a) Unmanaged (b) Managed

Fig. 5.4 Top level diagram of unmanaged and managed communication approaches.

NVLINK connections are truly point-to-point, making managed communication inevitable as routing must be done in software.

Although the current use of NVLINK is limited to a single node and only a small number of GPUs, it is fair to assume that this number is going to increase as deep learning applications seem to benefit from a large GPU to CPU ratio. However, it remains unclear in what way these GPUs will be connected at larger scale and what communication model will be supported.

Currently the two main NVLINK-enabled systems are NVIDIA's DGX-1 [10] with 8 GPUs and IBM's Minsky system [84], implementing 4 GPUs. The DGX-1 system comprises two sockets with each socket being connected to a quad of GPUs. GPUs within a quad can access each others memory, but communication between quads is limited and may need to fallback to PCIe instead of NVLINK.

5.1.3 Software Abstraction and Runtime

The software architecture provides the application with access to the hardware by implementing an interface, through which the application can exchange data. The actual architecture depends on the networking hardware, but also the communication model and the operating system. Furthermore, it can be distinguished between managed and unmanaged communication, which is compared in Figure 5.4.

In an unmanaged communication scheme the application is provided with a direct interface to the hardware and the operating system. Communication is processed by the application, possibly supported by the NIC's offloading capabilities. The example in the figure shows a Global Address Space (GAS) approach, in which the application maps the network's memory into the local VAS. As soon as the system is set up on the CPU, the pointers are passed to the GPU which can directly access data within the network through the NIC's shared memory unit. Additionally, put/get descriptors can be written to the NIC for data transfer offloading. However, the memory buffers are managed by the application and the user. This approach is well supported by EXTOLL, NVLINK, and partially by Infiniband's RDMA capabilities. While shared address space models directly map to unmanaged communication, message passing requires additional management.

Managed communication, on the other hand, adds an additional software layer in between the application and hardware to assist the application with communication. This allows for higher abstraction and communication models, such as message passing or task-based programming models. However, it also imposes additional overhead and occupies more GPU resources as the runtime functions need to be executed on the GPU either within the application kernel or as separate grid. Managing communication is discussed and analyzed in Chapter 6 of this work.

The following provides more detail on the how the NIC is accessed by the GPU.

5.2 GPU Global Address Space

GPU Global Address Space (GGAS) [15], [85] is enabled by EXTOLL's SMFU, which allows to span a global address space across physically distributed nodes. A similar approach was MEMSCALE [83], in which distributed CPUs store a large in-memory database and share a common and non-coherent address space. Database accesses are forwarded by the SMFU on load/store granularity. Although the performance is superior to Solid State Disk (SSD) and hard drive solutions, the approach eventually suffered from the small number of outstanding memory operations of typical x86 processors.

As opposed to CPUs, GPUs do not attempt to minimize latency but rather try to hide accesses, to which a large number of outstanding memory operations is key. Consequently, the SMFU approach to allow a large shared address space across many GPUs seems to be more compatible with the GPU's architecture.

5.2.1 Architecture

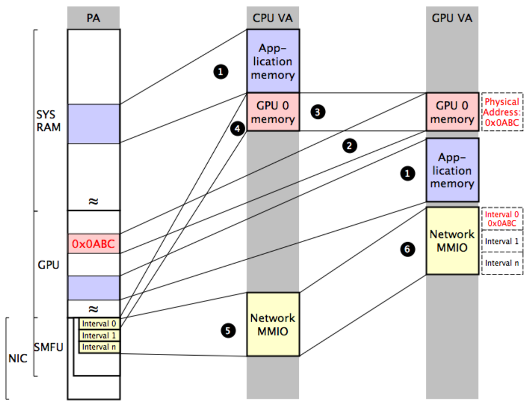
In order to use the SMFU the intervals need to be set accordingly, hence pointing to GPU memory. The memory mapping is shown in Figure 5.5, along with the software architecture which is discussed afterwards.

5.2.1.1 Memory Mappings

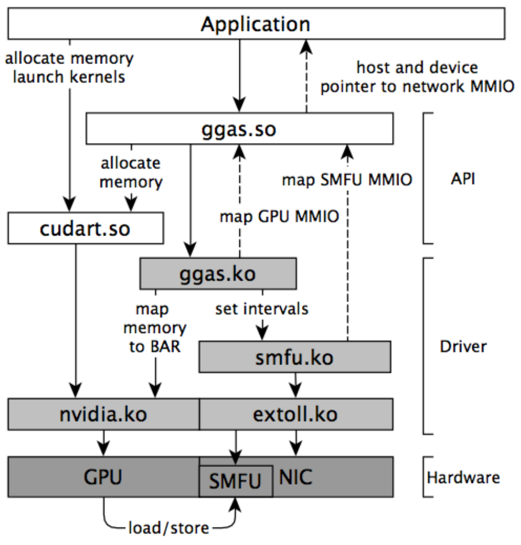
The diagram shows the physical and virtual address space of both CPU and GPU. First, the application allocates memory on either or both processors (❶). In order to initialize GGAS and set up the mapping, a portion of GPU memory is allocated which is then mapped to the GPU's PCIe BAR (❷). After the memory is allocated with CUDA's `cudaMalloc`, the GGAS driver is passed the unique identification of the memory, using three components: the GPU virtual address, a virtual address token, and a peer-to-peer token. This is necessary for the CUDA driver to map these addresses to the PCIe BAR. The BAR can then also be mapped into the CPU's address space through MMIO (❸). The next step requires to set the SMFU intervals accordingly. Here, one interval is assigned to each GPU. The physical address of the GPU's PCIe BAR is returned to the GGAS driver and passed to the SMFU driver, where the physical target address of the particular interval of the GPU is set. All incoming memory accesses hitting this interval are then forwarded and completed at this address, hence the GPU memory (❹). In order to allow the GPU to access the SMFU directly without going through the CPU, the intervals need to be mapped into the GPU's virtual address space. This has required to enhance NVIDIA's CUDA driver by MMIO, but recent drivers support this naturally (❺). As intervals are now accessible by the GPU, any memory accesses are forwarded to the appropriate nodes and completed remotely.

5.2.1.2 Address Translation

Remote memory accesses require a few steps of address translation. This is illustrated in Figure 5.6. If multiple threads of the same warp access consecutive



(a) Memory mapping



(b) Software architecture

Fig. 5.5 Memory mappings and software architecture of GGAS as an example for a shared address space model. The ending *.ko* indicates a kernel driver module, while *.so* stands for dynamic shared libraries.

addresses, requests are combined into a single transaction as shown in Figure 5.7. EXTOLL's SMFU provides performance counters that show how many accesses have been made to certain intervals. This allows to obtain information on how many transactions are forwarded by the GPU's memory controller. Increasing the stride with which memory is accessed allows for less accesses to be combined. Interestingly, the memory controller combines write accesses much more than load operations. Load operations are only coalesced for consecutive accesses and any stride greater than one leads to separate transactions. Write accesses, on the other hand, are combined up to a stride of four.

The next step is to translate the virtual address to a physical address which points to the SMFU's intervals, which is performed by the memory controller. Here, the NIC translates the local physical address to a global address, pointing to the interval, and an offset. This is transferred to the target NIC. On the receiving side, the NIC receives the network packets and determines the interval based on the received global address. The interval is set with a physical target address, which points to the GPU PCIe BAR. The received offset is added to this address and the memory operation is completed. In case of a load operation, the data is read from the memory, sent back to the source, and written to the source GPU's memory by the source SMFU.

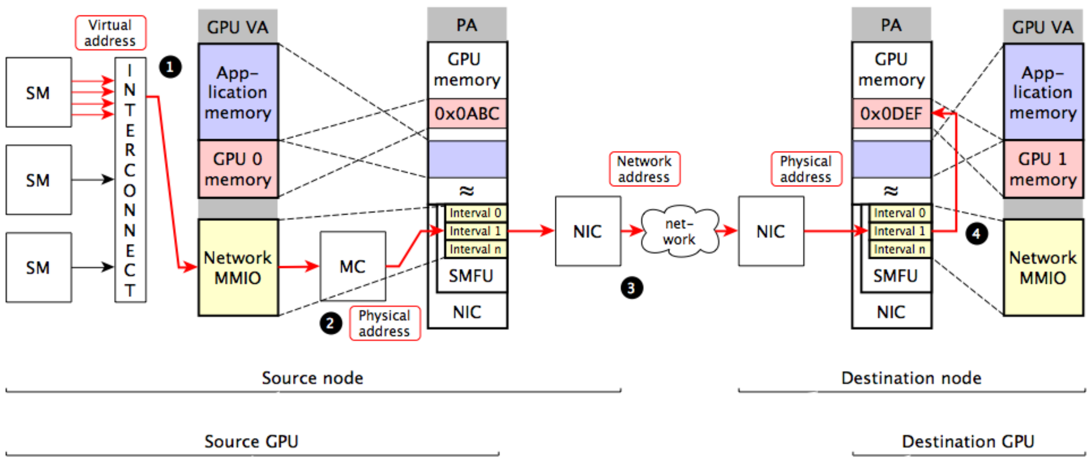


Fig. 5.6 Address translation as required by GGAS. The graph shows every translation step that is required from issuing the memory operation to completion on the receiving side.

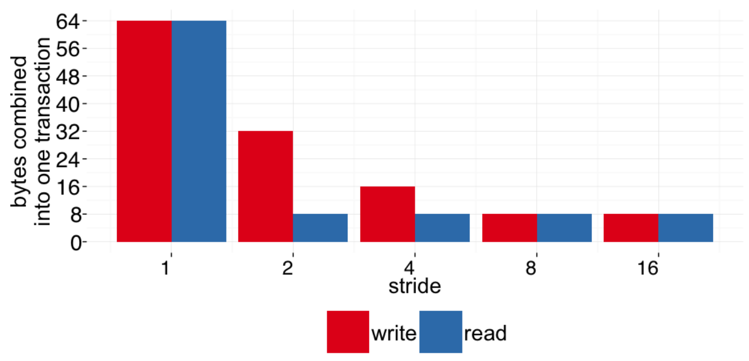


Fig. 5.7 Hardware coalescing of memory accesses for read and write operations. The coalescing is performed by the GPU's memory controller and reduces memory bus transactions.

5.2.1.3 Library Interface

The software architecture of GGAS is depicted in the second graph of Figure 5.5. It comprises a kernel driver and a user-space library as an interface to the application. The *ggas.ko* kernel module bridges the SMFU and NVIDIA driver. The library essentially provides an initialization routine, in which memory is allocated and mapped to the PCIe BAR of the GPU. The user can retrieve host and device pointers to the mapping of both GPU PCIe BAR and SMFU intervals. The device interface of the GGAS library allows to obtain a pointer to a particular node's memory. The library returns the pointer to the SMFU intervals and adds the appropriate offset to target the particular node. The following code shows an example of a naive broadcast implementation. First, references to remote memories need to be obtained (line 11), to which each threads writes the data in the second step (line 12).

Listing 5.1 Example code for a naive broadcast kernel using the GGAS communication model.

```
 1 __global__ void ggas_broadcast( double * input )
 2 {
 3   int tid = threadIdx.x ;
 4   double * remote_memory[ NUM_NODES ] ;
 5
 6   for( int i = 0 ; i < NUM_NODES ; ++i )
 7   {
 8     if( i == __ggas_local_node_id( ) )
 9       continue ;
10
11     remote_memory[ i ] = _ggas_get_pointer_of_node( i ) ;
12     remote_memory[ i ][ tid ] = input[ tid ] ;
13   }
14
15   return ;
16
17 }
```

5.2.2 Synchronization and Collectives

Besides writing and reading from remote memory, GGAS also provides a barrier and a collective reduce operation. The barrier's implementation is based on the work from Fröning et al. [86], who proposed a scalable and fast barrier for distributed CPUs with a non-coherent shared address space. As atomic operations on remote memory are not supported a hierarchical approach is followed in which the barrier data structure is only written by a single thread in a store-only fashion. Hence, synchronization is divided into two phases. First, one thread per GPU sets a flag that resides in the global address space and one designated master GPU waits for all GPUs to have set their flags. The second phase is reached when all flags are set and the master GPU sets another flag for each other GPU to signal completion of the barrier. The necessary intra-GPU barrier, which is required before entering the global barrier, is implemented as shown by Xie and Feng [87]. The resulting latency amounts to about 11 μs with 12 nodes, each equipped with an EXTOLL FPGA and a Kepler GPU. The design indicates good scalability as a barrier between 2 nodes requires about 9 μs.

An important aspect of the GGAS barrier is the GPU's CTA scheduling. Once scheduled, CTAs will run to completion until others can be scheduled. Thus, a barrier is impossible as running CTAs cannot be preempted. Overcoming this issue requires any of the following models to be applied [15], however both significantly alter the global shared address space programming model and pose significant drawbacks:

1. An application can only use as many CTAs as the hardware can execute concurrently. For example, a Kepler K20 GPU allows for about 52 CTAs to be scheduled concurrently if shared memory and register usage is kept small.
2. The application is divided into compute kernels which are launched from a single CTA master kernel. The master kernel synchronizes compute kernels via CUDA's *cudaDeviceSynchronize* or streams and the barrier synchronizes master kernels from each GPU within the network.

The second collective operation is the reduce and all-reduce, respectively [71]. Due to the significantly low bandwidth of remote read operations when transactions need to pass the CPU's root complex the reduce is implemented with remote write operations only. Hence, all GPUs transfer their data to a designated root GPU, where the data is reduced and the result broadcasted in case of an allreduce operation. The aforementioned barrier is used to ensure all GPUs have sent their data to the root and after the broadcast to comply with the collective's synchronization guarantees. However, for small data sizes the synchronization overhead is too significant, thus all GPUs send their data to every other GPU where it is reduced locally. Large reductions, on the hand, suffer from the data distribution, which is why GGAS provides two different strategies in this case.

1. Large installations with large numbers of GPUs should span a tree structure in which the reduction is performed. This has been extensively and successfully studied for large-scale CPU clusters [58], [88] and can also be applied to GPUs.

2. Instead of an allgather, in which any GPU sends its data to every other GPU, an all-to-all operation can be used to split the data and send it as chunks to other GPUs. Each GPU reduces its data and transfers the data back to the root or to all GPUs for an allreduce operation.

Compared to a CPU-controlled MPI reduce operation on data residing on the GPU, GGAS's reduce is about 1.8 times lower in latency for small amounts of data [71].

5.2.3 Summary

In summary, GGAS is representative for a NIC-assisted PGAS model with great performance for small messages, both in terms of latency and bandwidth. However, there are also some drawbacks of which one has to be aware. For example, remote reads perform poorly if the CPU's PCIe root complex is passed and thus a remote store only paradigm needs to be applied. Furthermore, it is not possible to address the entire GPU memory through the PCIe BAR. In order to address large amounts of memory, peripheral devices need to be mapped above the 32-bit addressable space. As the user usually has no control over the placement of the system's PCIe mapping, the system may place the BAR above 40-bit, which means the GPU cannot address this memory through MMIO anymore. Nonetheless, this is a technical limitation that can and should be overcome with future GPUs architectures and host systems. Another drawback are the resources that are occupied by GGAS for data transfers. If a large amount of data is to be transferred GGAS requires many SM resources and load/store units to perform the copy operations. In some cases this reduces the processor availability to the application and does not allow for communication to be overlapped with computation. This is going to be analyzed in more detail at the end of this chapter when performance is discussed on application level.

Another important aspect of a shared memory model is consistency, which determines the order of instructions. The CUDA programming guide describes the memory model as *weakly-ordered* [23]. Within a single GPU, CUDA provides memory fence instructions at various scopes. For example, `threadfence_block` concerns threads of the same CTA only, while `threadfence_system` also comprises system memory and peer memory in addition the GPU's own memory. Fences are necessary to ensure memory operations are issued before the fence returns. For example, instructions can be reordered by the compiler and fences prohibit the compiler from moving instructions across the fence. Furthermore, the hardware has to flush all reorder buffers to ensure operations are executed before the fence returns. The programming guide [23] also states that the *volatile* specifier should be used for shared data to avoid stale copies in caches.

In GGAS, fences are necessary if flags are written along with data. In such case, a system-wide fence needs to be inserted between data and flag to ensure the flag is written after the data. However, the flag must be declared volatile and polling is still required. Nonetheless, a comprehensive study [89] revealed several false statements and assumptions regarding the GPU's memory model. For example, it was discovered that *volatile* statements do not prevent the compiler from reordering load instructions in CUDA version 5.5. Nonetheless, none of these issues have been observed with GGAS and recent CUDA releases, but one has to be careful in this regard.

Note that this shared memory approach is naturally supported by NVLINK, but currently not supported by Infiniband. NVLINK enhances the model by providing atomic operations and memory fences across the network. Furthermore, the entire remote GPU memory can be mapped as opposed to GGAS, in which only a part can be accessed due to the aforementioned 40-bit limitation of addressing MMIO memory.

5.3 Remote Memory Access

A PGAS model has the two fundamental communication operations, namely *put* and *get*. Instead of using GPU resources to copy data the transfer can be offloaded to the NIC's DMA engines, increasing processor availability and overlap between communication and computation. The following presents RMA, also referred to as RDMA, approaches for GPUs, based on Infiniband's and EXTOLL's network architectures. NVLINK also offers RDMA capabilities in the form of `cudaMemcpy`, which performance is discussed in Section 6.3.1 in more detail.

5.3.1 Infiniband

Although Infiniband's main focus is message passing it also provides an RDMA VERBS API for one-sided communication. In order to transfer data between two GPUs' memories, the source and target memory regions need to be registered prior to any data transfer. The memory registration returns two keys, a local key for local access and a remote key for remote access. Furthermore, an entry in the memory translation table is created, in which the corresponding physical address can be found for any registered virtual address and local/remote key.

An RDMA transfer is initiated by first generating a work request and then enqueuing it to the send queue as part of Infiniband's QP. The transfer is triggered when the HCA's doorbell register is written. The user can specify in the work requests if a notification is generated after the work request is processed. The notification is written to a separate completion queue that can be shared by multiple QPs.

Besides traditional put/get operations, the VERBS API also provides two-sided semantics and atomic operations. These differ from MPI, for example, as they require receive requests to be posted before the send operation.

A GPU implementation of Infiniband's VERBS has first been proposed by Oden and Fröning [90]. While data is transferred directly between GPUs, communication is also controlled on the GPU and without any involvement of the CPU. Similar to GGAS, the GPU's PCIe BAR is mapped to GPU's VAS and the physical address is passed to Infiniband to register it for RDMA transfer. However, this requires to modify kernel drivers as the registration would fail with

Table 5.2 Infiniband VERBS latency for queue pairs residing in host and GPU memory [90].

Operation	CPU-controlled	GPU-controlled	
	system memory queue	system memory queue	GPU memory queue
IBV Post Send	0.01 μs	12.5 μs	10.7 μs
IBV Poll CQ	0.01 μs	15.7 μs	8.3 μs
IBV Post Send (opt.)		7.5 μs	7.0 μs
IBV Poll CQ (opt.)		5.5 μs	3.1 μs

MMIO addresses. In order to enable the GPU to control data transfers between the registered GPU memory regions the HCA needs to be made accessible too. Again, this is similar to GGAS as QP and doorbell register are mapped to the GPU's address space. In fact, instead of mapping the queues address, the queues can also be entirely placed in GPU memory and thus access costs are reduced significantly. The corresponding latencies are shown in Table 5.2.

It becomes immediately obvious that GPU-controlled communication shows a significantly higher latency than its CPU counterpart. However, moving the queues from system to GPU memory reduces latency by 14% for posting the send and almost 50% for polling on the completion queue. The authors blame the single-threaded work generation for the significant higher latency. Creating new work requests require multiple memory accesses, which require much more processor cycles than comparable memory accesses on the CPU. Furthermore, the GPU's clock frequency and thus single-thread performance, is also significantly lower.

Table 5.2 also shows an optimized version of both operations, in which some steps are omitted that are usually performed in the CPU version. These steps include checking if queues overflow, interpretation of the completion element, and so-called stamping in which work requests are marked during their generation to avoid prefetching from the HCA. As a result, the latency is reduced significantly as less instructions are required that are executed and which performance is limited by a single thread.

More details on Infiniband VERBS on GPUs can be found in Lena Oden's dissertation [15], including code examples and profound performance analyses.

Daoud et al. [91] follow a similar approach, in which the GPU is able to control the Infiniband NIC and the QP is allocated in GPU memory. The doorbell register is mapped to the GPU's address space as well. Overall, the results shown in their work are similar to the just shown VERBS implementation.

Infiniband is also used in the work of Gysi et al. [13]. Here, MPI's RDMA features serve as a model for their GPU-sided RDMA approach. However, actual communication is executed by the CPU, while instructions come from the GPU through command queues in host memory. As this approach is not limited by the GPU's single thread performance the resulting latency and bandwidth are superior to the approaches shown above. Nonetheless, the CPU is still needed to perform communication.

5.3.2 EXTOLL

Similar to GGAS, controlling EXTOLL's RMA capabilities from the GPU requires to map NIC resources to the GPU's address space. Here, the requester BAR needs to be mapped to trigger operations and the notification queue needs to be mapped to receive status information. While the BAR is an MMIO address again, the notification queue resides in kernel space and system memory.

Prior to any data transfer, the communication has to be set up on the CPU before the GPU is enabled to control the NIC's RMA unit. Before any data transfer can take place, the application needs to establish a *connection*, for which a *port* needs to be opened first. The port is based on the node ID and represents a point-to-point path to another node. A connection, on the other hand, is a virtual path between processes or threads. Multiple connections can be established using the same port and each connection is identified by its VPID. When a connection is established particular memory regions can be registered, for which the ATU will create an NLA. However, this is not necessarily required as the RMA unit can also deal with physical addresses, but it is recommended to use an NLA instead to comply with security standards. The NLA needs to be exchanged with other processes that are allowed to access the memory region remotely, which completes the initialization phase.

Once a port is open, a connection established, and a valid NLA of remote memory obtained, the data transfer can be initiated by either *put* or *get* operations. In order to hand off control to the GPU, the port is mapped to the GPU's address space, together with the notification queue. Furthermore, the node ID, VPID, and NLA are also passed to the GPU kernel. A data transfer is started by writing a descriptor to the mapped port. EXTOLL's descriptor format is kept lean and the NIC immediately starts with the transfer once the last word of the descriptor

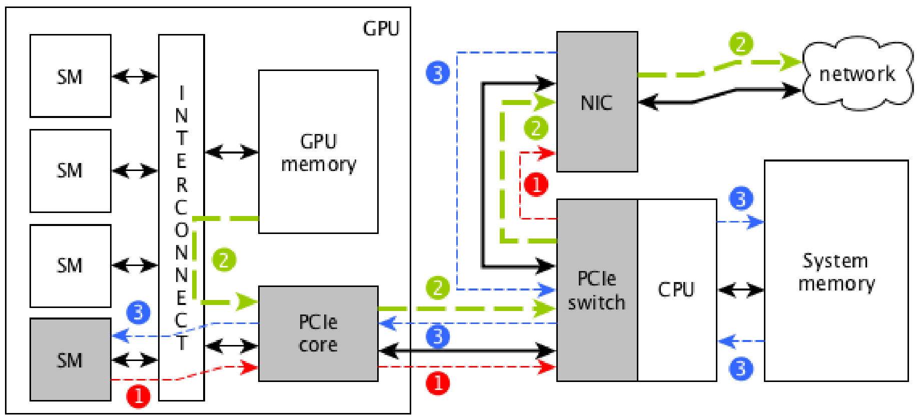


Fig. 5.8 Architecture and system integration of the EXTOLL RMA for direct GPU to GPU communication.

is received.

While the software architecture is similar to GGAS, depicted in Figure 5.5, the communication procedure differs and is shown in Figure 5.8. First, the GPU writes an descriptor, sometimes also called work request, to the NIC device through PCIe. The descriptor contains three 64-bit words and the NIC immediately starts with the data transfer when the last word is received. Copying the data from the local to the remote GPU is executed by the NIC's DMA engines at the source and receiving side. After the transfer is completed the NIC generates notifications and places them into the appropriate queue in system memory. Both sender and receiver can then query the queue for the notification and consume it.

A code example of how the communication is set up on the CPU is shown in Listing 5.2. First, the port is opened and a connection is established the same way it would be done for the CPU. Registered regions need to be exchanged with communication peers as they are directly addressed by put and get operations. In line 12, the information is copied and made accessible to the GPU before the kernel is launched. The kernel is shown in Listing 5.3. The data transfer can be triggered by a single thread. The put operation takes the port, connection, and destination nodes as parameters, hidden in the `info` data structure. Furthermore, the source and destination regions need to be passed as well as the size of the data. The notification requires polling, hence encapsulated into the do-while loop.

Listing 5.2 Example host code for setting up EXTOLL's RMA on GPUs environment.

```
int main( ... )
{
  /* ... */
    rma2_open( &port ) ;
    rma2_connect( port, dest, ... , &handle ) ;
    rma2_register(port, buffer, size, &send_region) ;

    exchange_regions( dest, &send_region, recv_region ) ;

    grma_copy_data2region( &data, &send_region, size ) ;

    grma_get_info( &dev_rma_info, dest, port, \
                   handle, send_region, \
                   recv_region, size ) ;

    grma_send<<<grid,block>>>( dev_rma_info ) ;
    /* ... */
}
```

Listing 5.3 Example device code for copying data using EXTOLL's RMA API on GPUs.

```
__global__ void grma_send( grma_info_t *info )
{
  int tid = threadIdx.x ;
  int bid = blockIdx.x ;

  grma_noti_t noti ;

  if( 0 == tid && 0 == bid )
  {
    grma_put( info, info->dest, info->size, \
              &info->send_region );
    do
    {
      grma_get_noti( info, &noti ) ;
    }while( !noti.valid ) ;
  }
  __syncthreads() ;

  return ;
}
```

5.3.3 Comparison

Although both Infiniband and EXTOLL provide RDMA capabilities, their implementation and interface are different, allowing to compare them accordingly. This eventually helps to design a tailored and optimized RDMA interface for direct GPU communication.

The main difference between Infiniband and EXTOLL is the way data transfers are triggered. Using Infiniband, the work requests are placed in a queue and the data transfer begins when the doorbell register is written. Then, the NIC starts to process work requests, which it fetches from the queue. This approach optimizes the processing of multiple work requests as they are placed in the queue first and then a single write to the doorbell register starts the transfer. Contrary, EXTOLL's work requests are written to its PCIe BAR and the transfer starts when the last word of the work request is received. Here, multiple work requests require multiple PCIe BAR accesses, which cannot be coalesced by the GPU's memory controller and thus become serialized.

5.3.3.1 Bandwidth and Latency of EXTOLL's RMA

EXTOLL's performance is determined for various mechanisms and approaches, which are introduced in the following.

1. *direct*: the sender GPU issues a put operation to the NIC directly and polls on system memory to consume the requester notification, which is generated when the request is processed. The receiving side waits for the completer notification to complete the data transfer.

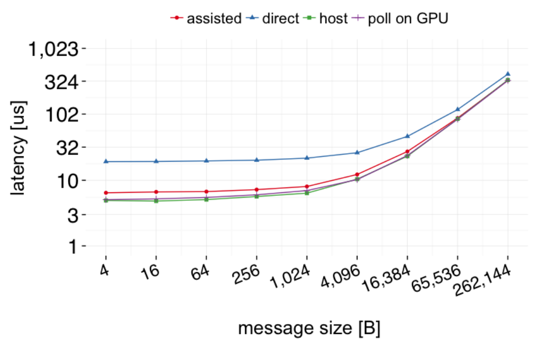
2. *polling on GPU memory:* the receiver GPU polls on the data directly to ensure everything is received. This avoids accesses to system memory as no notifications are generated and consumed.

3. *assisted*: data transfers are triggered on the GPU, but executed on the CPU. Both processors synchronize through a flag mechanism that resides in system memory and is mapped to the GPU's address space.

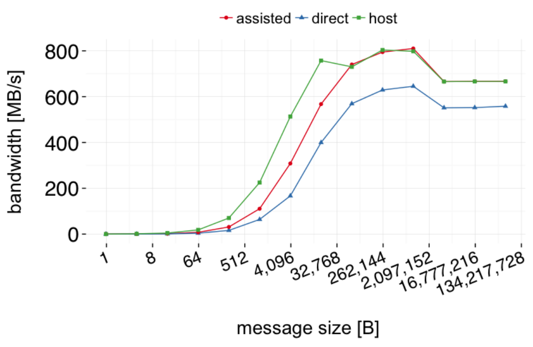
4. *host*: The entire data transfer is handled by the CPU, serving as a reference to GPU-controlled mechanisms. Data is still copied from GPU to GPU.

Half-roundtrip latency and bandwidth for these approaches and EXTOLL's RMA engine are shown in Figure 5.9. The experiments refer to an EXTOLL Field Programmable Gate Array (FPGA) NIC with a clock frequency of 157MHz and 64-bit data paths.

The latency of the *assisted*, *polling on GPU memory*, and *host* approach does not differ significantly among each other, while the *direct* approach's latency is about twice as high as the host-controlled approach. The main difference between the *direct* and the other GPU-controlled approaches is that host memory is polled for new notifications, which seems to degrade performance significantly. This is supported by GPU performance counter, shown in Table 5.3. The bandwidth is similar with host-controlled bandwidth being about 30% higher than the device controlled approach. However, it seems to magically drop for data larger than 1MB. This is due to an issue with PCIe's peer-to-peer protocol and read accesses



(a) Half-roundtrip latency



(b) bandwidth

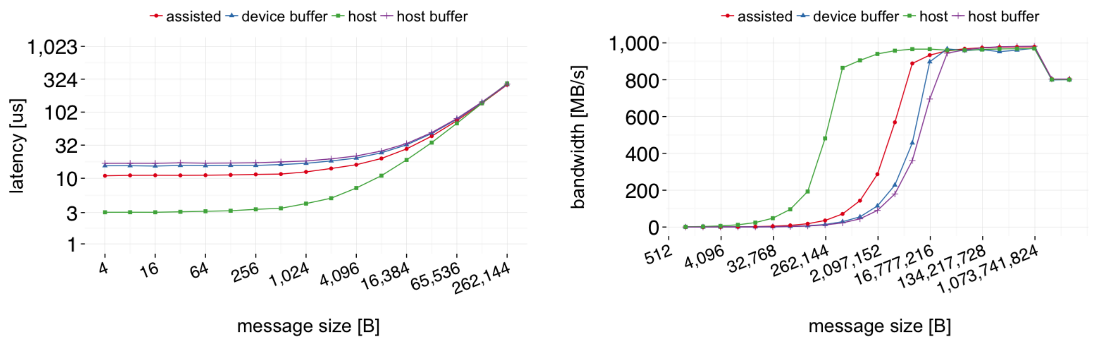
Fig. 5.9 Latency and bandwidth of the EXTOLL RMA for data transfers between GPU memory [19].

Table 5.3 GPU performance counters for RMA transfers for various polling locations [19].

Metric	Performance Counter	
	System memory	Device memory
System memory read transactions (32B)	4,368	0
System memory write transactions (32B)	2,908	303
Global memory read transactions (64b)	0	1,314
Global memory write transactions (64b)	500	400
L2 read hits	0	3,143
L2 read requests	4,822	2,970
L2 write requests	5,268	404
Memory accesses (read/write)	6,788	1714
Instructions executed	46,413	22,491

that are routed through the CPU's PCIe root complex. This has been improved in newer CPU generations starting with Intel's *Ivy Bridge*.

When system memory is polled the overall instructions that are executed are doubled compared to polling on device memory. This is mostly due to long-latency system memory accesses and the GPU not being able to cache them adequately. Contrary, polling on device memory allows read accesses to be cached and hence accesses are served much faster, significantly reducing the overall number of instructions. However, polling on the data to be received requires to know what data is expected and to check every data element. If large chunks of data are moved the GPU's consistency model does not guarantee any order in which the data is received. Consequently, it is not sufficient to poll on the last element. Alternatively, a flag mechanism can be used in which a memory



(a) Half-roundtrip latency (b) bandwidth

Fig. 5.10 Latency and Bandwidth for Infiniband VERBS on GPUs [19]. Data is transferred between GPU memories.

barrier is called in between writing the data and the flag.

5.3.3.2 Bandwidth and Latency of Infiniband VERBS

Figure 5.10 shows latency and bandwidth for the Infiniband's VERBS implementation on GPUs with similar approaches as listed before for EXTOLL. Instead of polling on data elements on the GPU the entire QP is placed in GPU memory, referred to as *device buffer*. The test system for the experiments contained one 4x FDR HCA per node and the *openstack* subnet manager version 4.0.5.

Interestingly, neither latency nor bandwidth seem to benefit from the QP residing in GPU memory. Their latency is roughly the same regardless of the queue placement and about 6 times higher than comparable host-controlled data transfers. The same is observed for the bandwidth.

The reason for the latency not differing significantly for device and host buffered queues can be found in the performance counters again, shown in Table 5.4. Although system memory accesses are reduced by 70% if queues are allocated in GPU memory, the overall instruction count is almost the same. Hence, the queue placement is not the limiting factor, but rather the complex single-threaded work request generation. An example of its complexity is the conversion from little-endian to big-endian values as it is required by the Infiniband NIC.

The implication of Infiniband's complex work request generation becomes even clearer when its counters are compared to EXTOLL. Overall, more than twice as many instructions need to be executed. EXTOLL's lean work request format and the immediate start of data transfers upon receiving the last word

Table 5.4 GPU performance counters for Infiniband's queue pairs residing in host and device memory [19].

Metric	**Performance Counter**	
	Host buffer	**Device buffer**
System memory read transactions (32B)	772	80
System memory write transactions (32B)	670	316
L2 read misses	999	1,405
L2 read hits	16,647	14,575
L2 read requests	16,657	15,110
L2 write requests	1,990	1,885
Memory accesses (read/write)	59,937	58,905
Instructions executed	123,297	110,463

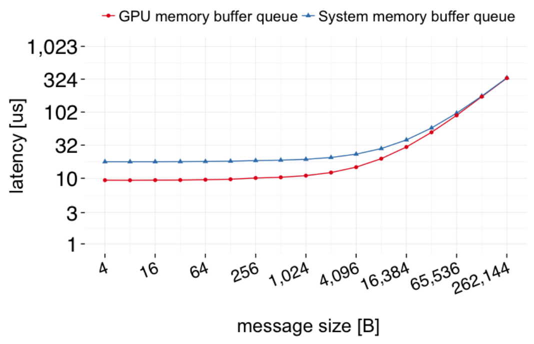
seems to be beneficial. However, EXTOLL also showed a significant performance gain by polling on device memory instead of system memory, in which the queues are allocated. Queues placed in GPU memory also show performance benefits for Infiniband, although the difference to host-buffered queues is not as significant as one might expect.

5.3.3.3 Moving EXTOLL Notifications to GPU Memory

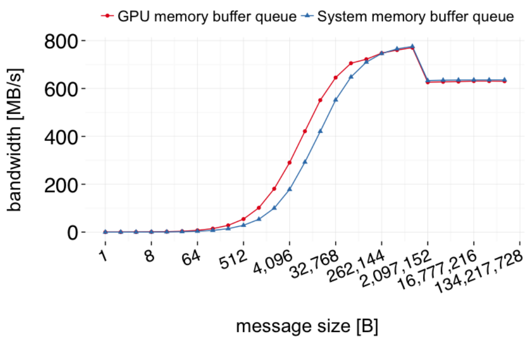
Unlike Infiniband, EXTOLL does not allow for queues to be allocated in GPU memory and significant changes to the software stack are necessary to still enable queues to reside in GPU memory. The EXTOLL NIC writes notifications to the *Buffer Queue*, which consists of a read pointer and descriptor storage. The pointers reside in the NIC's PCIe BAR space, while the descriptors are allocated in system memory. When notifications are consumed the read pointer is advanced. In order to move these to GPU memory, the following steps are required [92]:

1. When a new EXTOLL RMA port is opened, GPU memory is allocated and mapped to the GPU's PCIe BAR by the patched *nvidia.ko* driver.
2. A new set of buffer queues is created and pointers are redirected to the physical address of the GPU's PCIe BAR.
3. Read pointer and buffer queue descriptor storage are mapped to the GPU's address space through MMIO.

Polling on GPU memory notifications can be further optimized by using the *.cv* (cache volatile) specifier for load operations. Also, the read pointer is not



(a) Half-roundtrip latency



(b) bandwidth

Fig. 5.11 Latency and Bandwidth of EXTOLL's RMA for notifications residing in system and GPU memory [92].

Table 5.5 GPU performance counters for polling on EXTOLL notifications in system and GPU memory [92].

Metric	**Buffer Queue placement**	
	System memory	**GPU memory**
System memory read transactions (32B)	3842	0
System memory write transactions (32B)	2052	315
Global memory read accesses (32b)	909	1,116
Global memory read accesses (64b)	781	893
Global memory write accesses (32b)	412	412
Global memory write accesses (64b)	606	606
Instructions executed	60351	65039

incremented every time a notification is consumed, but rather in a lazy scheme after a few accesses. The latency and bandwidth is compared to the original approach with notifications residing in system memory and results are depicted in Figure 5.11.

Moving the queue to GPU memory reduces the half-roundtrip latency by about half. The bandwidth is also increased for data less than 256kB and remains the same as the bandwidth of host-buffered queues afterwards. The performance counters are show in Table 5.5.

Although the overall instruction count is even slightly higher for queues placed in GPU memory, the number of long-latency system memory accesses is significantly reduced. As a result, the remaining instructions are mostly spent on creating work requests and polling on notification no longer seems to be a limiting factor. Interestingly, the number of global memory accesses is only slightly increased, thus notifications are quickly generated by the NIC after it receives the work request.

5.3.3.4 Verdict

It has been shown how RDMA transfers can be triggered and controlled on the GPU as opposed to relying on the CPU to handle communication. Two inherently different interconnection networks showed their benefits regarding the implementation and interface of their own RDMA capability. Summarizing it can be said that it is important to make the work request generation as lean as possible with as few instructions as possible. Unlike CPUs, the GPU performance suffers from any additional instruction that is required to be executed by a single thread only. Infiniband shows that their complex work requests do not fit the GPU's execution model and thus performance is degraded significantly.

Another important aspect is the placement of notifications. Both Infiniband and EXTOLL show lower latency and higher bandwidth if notifications are kept as close to the GPU as possible. This is certainly a approach that is mandatory for GPU-controlled RDMA transfers.

5.4 Message Passing

The importance of MPI has already been pointed out in previous chapters and this section is going to introduce mechanisms that enable messaging on the GPU. Furthermore, message passing requires a management layer on top of the communication, thus it can be implemented on top of GGAS, for example.

There are two approaches how to involve the GPU in message-based communication. The first approach enhances MPI by the capability of transferring data between two GPUs. This separates control and data flow as control remains on the CPU, but the data is transferred directly between the accelerators.

The second approach moves both control and data flow to the GPU and leaves the CPU as offloading device to eventually control the NIC.

Generally, it can be distinguished by the processor that controls the communication, thus CPU- or GPU-controlled. These approaches are discussed in the following.

5.4.1 CPU-controlled

CPU-controlled message passing can be viewed as the state-of-the-art in HPC systems. Here, MPI is the most prevalent communication model and communication is managed on the CPU. Since the number of systems that employ GPUs as accelerators has been increasing in the past years, the importance of direct GPU to GPU communication has led the MPI community to adapt its communication model accordingly. Starting with MVAPICH all major MPI implementations allow to pass device pointers to MPI routines.

An important step toward GPU-aware MPI was NVIDIA's introduction of GPUDirect, allowing multiple devices to share the same pinned buffer in system memory to avoid additional copies. This was later extended by the capability of GPUs of the same PCIe root complex to transfer data directly between each other without the CPU. The next version, referred to as GPUDirectRDMA, enables other PCIe devices to access the GPU's memory directly through peer-to-peer access. This allows a NIC to read from the GPU and put the data to the remote GPU's memory directly without any host memory staging copies. However, this only works well if the NIC and the GPU share the same root complex. Otherwise PCIe's issue with peer-to-peer read operations limit the bandwidth dramatically, thus making staging copies necessary for larger data transfers [93]. Consequently, MPI implementations like MVAPICH still use staging copies for larger transfers to overcome this issue.

The latest GPUDirect, referred to as *GPUDirectAsync*, allows GPUs to trigger network operations by accessing the NIC directly. Here, the CPU puts compute and communication tasks into the GPU's command queue. When the GPU is done with a compute task it can execute the communication task by triggering the NIC. Before, control had to be returned to the CPU from where the NIC was instructed to perform communication. This feature is currently being adopted by MVAPICH, extending MPI operations by the ability to take a special CUDA streams communicator as parameter. An overview of GPUDirect techniques can be found in Table 2.3 of Chapter 2.

In addition to being CUDA-aware, an extension to *OpenMPI* even supports non-contiguous data transfers between GPUs [94]. Here, specialized (un-)pack kernels are launched from the CPU to allow for contiguous data transfers of vector data types, which eventually improves the bandwidth for vector transfers significantly. There also exists work on MPI collective operations on GPUs [95], in which callback routines are used to to maximize overlap between computation on the GPU and MPI communication on the CPU.

Another implementation of GPU-centric collective communication is NVIDIA's *NCCL* library [96]. While mainly designed for intra-node communication between GPUs it provides optimized collective operations such as allgather, broadcast, and reduce of data residing in GPU memory. The second version, *NCCL 2.0*, scales out to multiple nodes, each deploying multiple GPUs. Nonetheless, it uses asynchronous CUDA kernels that are launched from the CPU and tied to a stream. Furthermore, there are plans to extend NCCL by point-to-point and neighbor collective operations [97].

5.4.2 GPU-controlled

One of the first efforts to implement message passing on GPUs is the Distributed Computing on GPU Networks (DCGN, pronounced *decagon*) framework by Stuart and Owens [12]. Before the authors introduce their messaging approach they discuss the challenges of implementing message passing on graphics processors. The major challenge is that GPUs are unable to access the NIC, preventing the GPU to source network traffic. Consequently, the NIC is accessed by the CPU, which polls on distinct memory locations to receive communication requests from the GPU while a kernel is running. Therefore, a dedicated CPU thread is constantly waiting for messaging requests from the GPU and then uses standard MPI for message passing, thus supporting many different NIC architectures. In fact, the CPU thread handles all MPI communication from both CPU and GPU threads. The reported performance is about 10% lower for high-level applications compared to a GAS+MPI model, however, the losses in micro-benchmarks such as latency and bandwidth are significant. For example, an empty message takes about 564 times longer than with MVAPICH2. Nonetheless, the authors highlight the importance of GPU-controlled communication models, especially with respect to the programming model. Rather than aiming at high performance, the authors want to present a road map for chip makers and vendors for future systems and requirements.

Another approach is followed by Sangman Kim et al. [98] and Silberstein [14], respectively, who aim to implement a networking layer with socket abstraction on the GPU. Here, a so-called "Daemon Thread Block" is executed on the GPU, which accepts requests and spawns computation as needed. Network buffers are kept on the GPU and written by the NIC directly, using *GPUDirectRDMA*. Initialization, e.g. creation of channels, is done by the CPU and control is then handed off to the GPU. Although the authors emphasize on how important it is that communication can be performed without the CPU, their approach still relies on the CPU to control the NIC. A ring buffer is used for data and control exchange between CPU and GPU. In their first work [98] they reason that GPUs are incapable of controlling I/O devices, such as NICs. The second publication [14] corrects this statement and says this is actually supported, but left it for future work. The reported performance for a face verification server shows a speedup of up to 2.3x compared to CPU-controlled communication, and an improvement in latency by a factor of 3. Furthermore, they assessed scalability and a distributed in-GPU MapReduce framework. A small cluster with four GPUs yields a speedup of 2.9-3.5x compared to a single GPU for a word count and k-means application.

In summary it can be said that it does not exist much work to implement message passing on GPUs. The main reason lies in the different execution model and architecture of the CPU and GPU. Most applications depend on fast message transfers which requires high single thread performance and low memory access latency, especially for synchronization purposes. GPUs, however, require parallelism to perform well and the message passing model cannot take advantage of the massive amount of parallelism. Nonetheless, it is a prominent communication model that is well understood and many applications rely on it. This model is going to be discussed more when managed protocols are introduced later in this work, particularly Section 6.3. Although NICs are optimized for messaging support, many aspects are required to be processed by the processor, such as memory management or message matching. Hence, even with the support of NICs, a substantial amount of the messaging is loaded off to the processor. Additionally, a message passing model can be layered on top of any physical communication model, such as a flat virtual address space or PGAS.

5.5 Performance Comparison

The preceding part elaborated on various NIC-assisted and GPU-centric communication models and their implementation, using Infiniband and EXTOLL as interconnection networks. Here, the communication models are compared in regard to performance, energy efficiency, and compatibility to the GPU's execution model and architecture. However, the comparison takes only interactions and low-level aspects into account, while higher level abstractions and capabilities are discussed in the remainder of this work.

The following evaluation is based on the EXTOLL interconnection network. While Infiniband also provides support for message passing and RDMA, EXTOLL also allows to evaluate the global address space approach. Nonetheless, results should be viewed independently of the underlying network architecture as their strengths and weaknesses have already been discussed throughout this chapter.

5.5.1 Benchmarks

The benchmarks used to evaluate different communication models are introduced in the following. The set of application aims to cover a wide range of applications with distinct characteristics. A brief introduction is followed by a summary of distinct characteristics in Table 5.6. Note that the benchmarks are implemented with GGAS, RMA, and CPU-controlled MPI communication.

N-Body This benchmark represents astronomic scientific simulations and models the gravitational interaction between bodies of random size. The computation of the gravitational forces in a three-dimensional space is intense with a complexity of $\mathcal{O}(n^2)$ with n being the number of simulated bodies. The force F between two bodies P_0 and P_1 is determined by the following formula, in which G refers to the gravitational constant, m to the mass of the body, and r to the distance between two bodies.

$$\vec{F_g} = G \cdot \frac{m_1 \cdot m_2}{\mid r_{12} \mid^2} \cdot \vec{r_{12}}$$

According to Newton's first law of motion, the force causes an acceleration which will change the body's position and velocity. Overall, this calculation requires about 32 floating point operations per interaction and is executed $n \cdot (n-1)$ times.

The simulation aims to calculate the new positions of each body after a given time, which is divided into small steps of Δt. The distributed algorithm assigns a set of bodies to each processor, referred to as local bodies. As the calculation of the new position of each body requires to apply the force of every other body, an all-to-all communication is necessary. The communication is implemented as a ring, in which the local bodies are sent to the succeeding rank and bodies are received from the preceding rank. Whenever bodies are received, their interactions with local bodies are calculated, updating velocity and position of local bodies. Then, the received bodies are shifted along within the ring and new ones are received.

GGAS would generally allow all bodies to be placed within the global address space, however, due to technical constraints only 256MB of PCIe BAR can be opened to the network. As this is not sufficient, a different message-based approach is implemented. Here, the limited BAR memory is seen as a mailbox in which bodies can be received. A global barrier is used to signal that all interactions have been calculated and the bodies can be shifted along again. This perfectly follows a Bulk Synchronous Parallel (BSP) model [99], which is common in scientific applications.

Himeno (Stencil) As shown in Table 4.1, nearest neighbor communication is the most prevalent pattern in scientific applications, thus it is represented here by the Himeno benchmark. The basis of this benchmark is the three-dimensional Poisson equation for compressible fluid analysis, solved using the Jacobi iteration method. A structured curvilinear mesh is applied to the fluid and solved using a 19-point stencil algorithm. Unlike N-Body, which is computational-bound, this benchmark is mostly limited by memory performance.

The three-dimensional mesh is divided into planes along the z-direction (depth). Each rank is assigned a set of planes, in which it calculates interactions between fluid particles. As neighbor particles have to be taken into account for the calculation, planes have to be exchanged between adjacent ranks. Thus, top and bottom planes are calculated first, then sent to other ranks and when data from neighbor ranks is received the middle part can be calculated as well. This allows for overlap as the center can be computed while new data is awaited.

Global Sum Another aspect that has been revealed by the trace analysis is that the *(all-)reduce* collective operation is in wide use. Hence, this benchmark reduces large amounts of distributed data, for example calculating the sum of all values. First, each GPU reduces its local data and then calls a collective reduce operation on the result.

Using GGAS, each GPU sends its local data to the root GPU, where the data is reduced locally. This yields the best performance at smaller scale and a moderate number of values to be reduced. A tree-based and hierarchical reduce operation may be more efficient for large-scale installations [71].

The MPI version has the data reduced on the GPU, from which the result is copied back to the CPU and reduced by MPI's allreduce operation. An RDMA version was not implemented as the amount of transferred data is too small to make efficiently use of DMA engines. In fact, only a single value is sent by each rank and therefore the RDMA work request is already larger.

Random Access The last benchmark that is used to compare the communication models for GPUs evaluates how well random memory locations can be accessed. The *Random Access* is part of the HPC Challenge Benchmark Suite [100] and defines rules how the random accesses have to be performed. It starts with a large table that is distributed across all GPUs. Next, each GPU generates a set of random indices, which identify a location within the table. The index at which the table is accessed can either result in local table accesses or remote table accesses. Due to randomness, accesses are hardly coalesced and often memory and network transactions contain only a single valid element, increasing overhead and reducing efficiency. Hence, the rules allow to combine up to 1024 accesses that modify a particular node's table. This reduces the number of memory and network transactions.

Although the table could be placed in the globally shared memory in the GGAS approach, the resulting table size would be rather small due to the limited amount of memory that is mappable to the GPU's PCIe BAR. Another issue is the lack of remote atomic operations and it also would not be possible to combine multiple accesses together to reduce network transactions. Thus, a mailbox scheme with send/receive semantics, similar to the N-Body benchmark, is implemented. The same approach is used for the RDMA implementation.

With MPI, the table is allocated on the GPU, the indices, however, are

Table 5.6 Characteristics of various benchmarks that are used to assess various communication models [20].

Characteristic	**Application**			
	N-Body	**Himeno**	**Global Sum**	**Random Access**
Computation	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Memory	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Communication	$\mathcal{O}(n)$	$\mathcal{O}(\sqrt[3]{n^2})$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Comm. pattern	Ring	Nearest Neighbor	All-reduce	Uniform Random
Average Payload	$\frac{n}{N}$	$\sqrt[3]{n^2}$	1 (element)	0-1024 (elements)
Overlap	+	++	−	-

generated on the CPU. At first glance, this seems like an unfair comparison to the other two implementations, but generating the indices is fast and copying them to the CPU from the GPU would simply introduce too much overhead. Furthermore, the actual memory access is supposed to be benchmarked, rather than the calculation of random indices.

5.5.2 Performance

Before the performance is evaluated for every benchmark and communication model, the bandwidth are compared in Figure 5.12. It is eminently obvious that GGAS' bandwidth is significantly superior to the other models. While MPI requires to leave the kernel and return control to the CPU for communication, GPU-controlled RDMA suffers from polling on system memory. The graph also shows CPU-controlled RDMA with additional `cudaMemcpy` operations to move the data to system memory before communication. As shown, it outperforms MPI as it accesses the NIC's RDMA capabilities directly without MPI's software overhead.

While bandwidth provides first tendencies it is not sufficient to make profound statements about best suited communication models for GPU-centric communication. Thus, the performance on application level for the aforementioned benchmarks is assessed. The test system comprised 12 nodes with Intel Xeon E5 processors and one NVIDIA K20 Kepler GPU per node. The nodes are interconnected by EXTOLL, implemented in an FPGA at 157MHz.

N-Body The performance of the *N-Body* benchmark for 2 and 12 nodes (weak scaling) is shown in Figure 5.13. At the smaller scale, GGAS yields

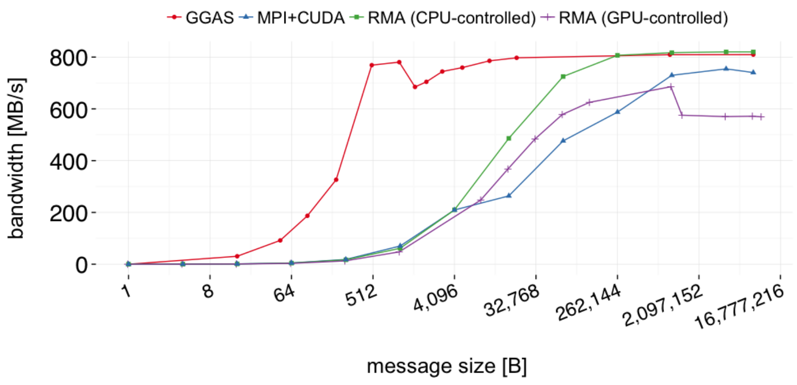
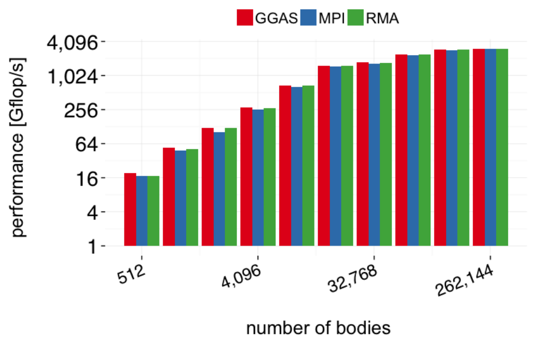


Fig. 5.12 Bandwidth of various communication models for GPU to GPU data transfers [20].

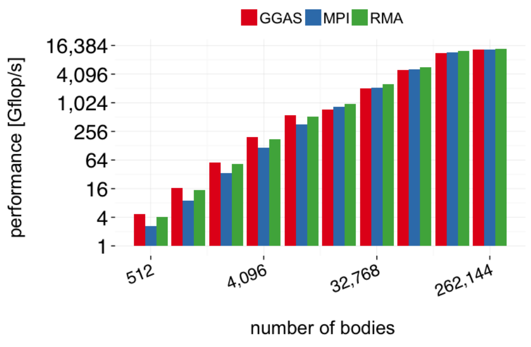
better performance for smaller problem sizes, but is outperformed by both RMA and MPI starting at about 4K bodies. The performance is similar for all three communication models at large problem sizes as computation outweighs communication by far. Increasing the node count to 12 shows similar effects, although GGAS can stay at the top until about 16K bodies.Then, GGAS is outperformed by RMA and MPI again.

While computation increases at a complexity of $\mathcal{O}(n^2)$, communication increases at $\mathcal{O}(n)$, hence time is dominated by computation at larger scale and communication fades into the background. Furthermore, the more data is needed to be transferred the more resources are occupied by GGAS, reducing resource availability for computation and limiting possible overlap. RMA, on the other hand, offloads the copy operation to the NIC and allows to fully use the GPU for computation. That is why RMA and MPI are superior to GGAS for larger problem sizes. Nonetheless, GPU-controlled communication is always favorable and outperforms MPI in any case.

Himeno (Stencil) Figure 5.14 shows the performance of the *Himeno* benchmark at weak scaling as the problem size increases linearly with the number of nodes. Unlike *N-Body*, GGAS is outperformed by MPI at any scale, but MPI's performance is itself inferior to RMA. Similar to the previous benchmark, GGAS requires GPU resources to be dedicated to copying the data while RMA and MPI allow to offload communication to the NIC and CPU, respectively. Hence, applications in which communication can be overlapped benefit from an offloading approach, therefore RMA outperforms GGAS. Nonetheless, GPU-controlled



(a) 2 nodes



(b) 12 nodes

Fig. 5.13 Performance of the N-Body benchmark for various communication models and 2 and 12 nodes [101].

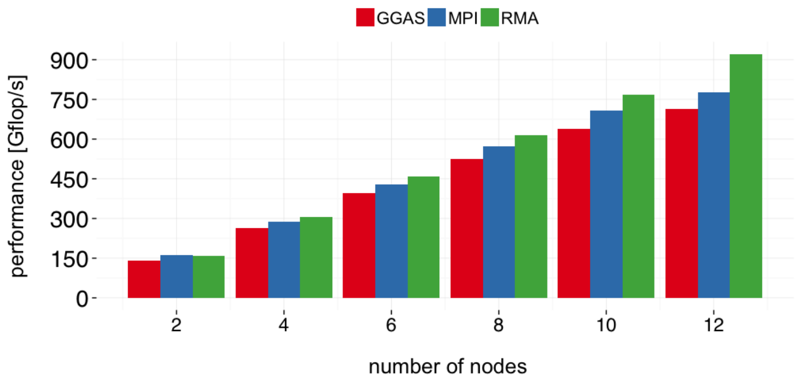
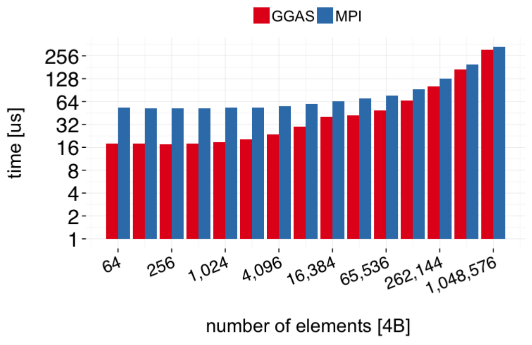


Fig. 5.14 Performance of the Himeno benchmark for various communication models at varying scale [101]. The problem size increases with the number of nodes, showing weak scaling.

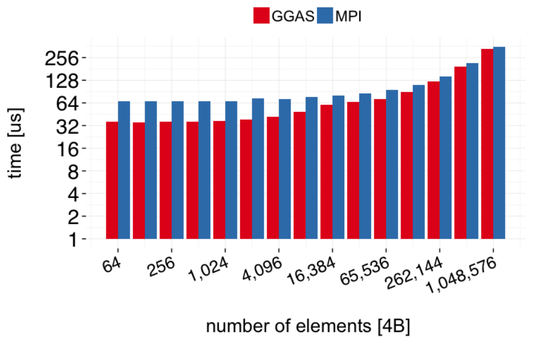
communication is yet again superior to MPI. In fact, RMA scales better than MPI as performance losses are minimal when the number of nodes is increased.

Global Sum The next benchmark that is analyzed is *Global Sum*, in which a large array is reduced to its sum. Figure 5.15 shows the execution time for various array sizes for 2 and 12 nodes, using GGAS and CPU-controlled MPI. Again, RMA is not evaluated as the benchmark's communication volume is small with only a single element per GPU.

The GGAS implementation is significantly faster than MPI and finishes in less than half the time of the MPI implementation for array sizes up to 4K elements. Then, the difference in performance starts to diminish as computation takes over the majority of the benchmark's run time. The difference between GGAS and



(a) 2 nodes



(b) 12 nodes

Fig. 5.15 Performance of the Global Sum benchmark for GGAS and MPI and 2 and 12 nodes [101].

MPI becomes smaller as scale increases. The more GPUs participate the less data is reduced locally and the more communication is needed. As a result, both execution times become larger but GGAS shows a steeper increase. Nonetheless, GGAS is never slower than MPI and GPU-controlled communication is again favorable in any case.

Computation of this benchmark increases with the number of elements while communication only depends on the number of nodes and is independent of the problem size. Hence, overhead that is caused by communication and synchronization increases at scale. As aforementioned, larger scale also requires to implement a tree-based structure to exchange results from local reductions, however, due to the limited number of available nodes it was not possible to evaluate performance for more than 12 nodes.

Random Access The last benchmark of this analysis randomly updates a distributed table, resulting in a large amount of small messages. The performance is measured in *GUPS (Giga Updates Per Second)* and shown in Figure 5.16.

As mentioned in the previous section, the benchmark's specification allows to combine up to 1,024 updates into a single message. Nonetheless, messages are comparably small, which is clearly in GGAS' favor as it can perform twice as many updates in the same time than MPI and RMA. However, the RMA implementation slightly outperforms GGAS at a scale of 2 nodes, but communication efforts are rather low and half of the updates concern the local table. As communication efforts increase, RMA's performance decreases accordingly while

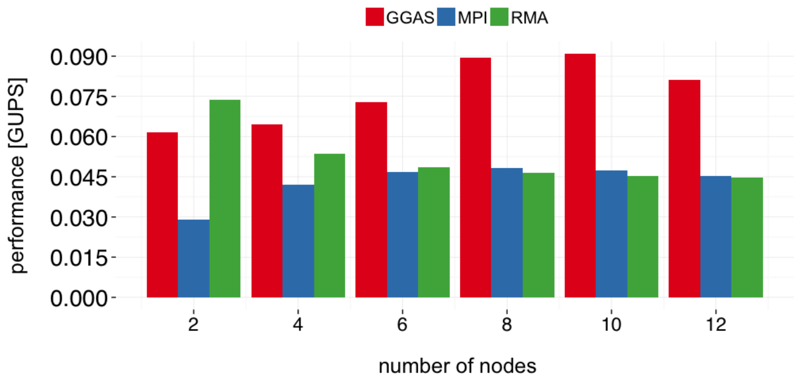


Fig. 5.16 Performance of the Random Access benchmark for various communication and a varying number of nodes [101].

GGAS' performance increases.

MPI's performance seems to slightly increase and becomes higher than RMA, for example, but this is mainly due to indices being generated on the CPU. Hence, only the updates are executed on the GPU. However, generating indices on the GPU would not make much sense as this operation is quite simple and the extra copy from device to host would dominate execution time.

5.5.3 Energy Efficiency

Although performance is often regarded as the main aspiration to improve and enlarge computing systems, they have become more and more limited by power and energy constraints. Thus, the power saving capabilities of different communication models needs to be taken into account as well. The energy E a system is spending is the integral of the power over time.

$$E = \int (P_{cpu}(t) + P_{gpu}(t) + P_{other}(t) \quad) \, dt$$

Consequently, energy can be saved by either reducing the run time of an application or reducing the power consumption of the entire system. The use of GPUs to accelerate certain compute-intensive parts of an application reduces execution time, while the GPU itself is more power efficient as the CPU as it runs at lower clock frequencies and cores are kept simpler without expensive features like out-of-order execution or branch prediction.

The large number of components makes it difficult to measure power accurately. While the overall system power can be measured with monitors, it

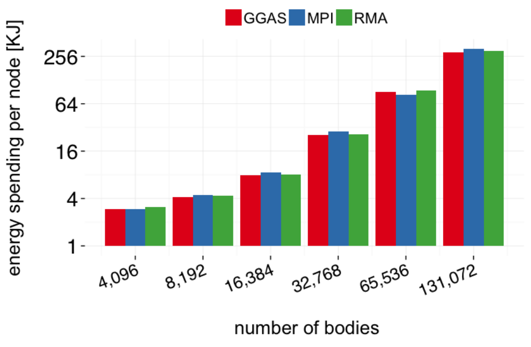
Table 5.7 Power consumption of various components and benchmarks using different communication models [20].

Comm. Model	Component	N-Body [W]	Himeno [W]	Global Sum [W]	Random Access [W]	Average [W]
GGAS	CPU	33.21	25.94	18.38	27.60	26.28
	DRAM	3.18	2.89	1.63	3.27	2.74
	GPU	81.01	105.13	63.57	68.67	79.60
	Total	147.43	133.96	83.59	99.54	116.13
RMA	CPU	32.16	23.96	n/a	28.16	28.09
	DRAM	3.27	2.60	n/a	3.22	3.03
	GPU	87.91	116.53	n/a	79.48	94.64
	Total	152.22	143.09	n/a	110.85	135.39
MPI	CPU	37.66	34.23	34.00	35.26	35.29
	DRAM	3.86	3.40	2.97	3.62	3.46
	GPU	82.93	106.99	57.48	51.79	74.80
	Total	158.25	144.62	94.44	90.66	121.99

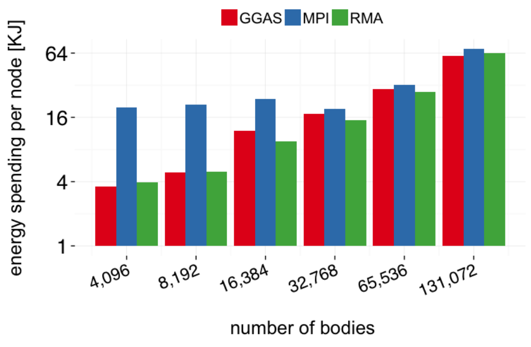
remains impossible to determine which components contribute most to the power consumption at certain moments of time. Fortunately, CPUs and GPUs provide software-centric approaches to query current power consumption. These values are based on various metrics and fed to a model which calculates how much power is spent. Intel provides the Running Average Power Limit (RAPL) interface for CPU and memory power consumption and the GPU's power can be queried with NVIDIA's System Management Interface (SMI). Other components, such as network, I/O, and peripheral devices are assumed to consume power at a constant rate.

The performance results from the previous section already show that GPU-controlled communication is always favorable. In addition, Table 5.7 shows the average power consumption by component and communication model for each of the benchmarks.

On average, the CPU consumes about 25% less power if communication is controlled on the GPU. While GGAS' system memory power consumption is also reduced by about 20% compared to MPI the GPU's power is only increased by less than 7%. However, using the RMA communication model increases the GPU's power consumption by about 25% compared to MPI. Unfortunately, it is not possible to break down the GPU's power even further to determine whether the increased power consumption is caused by compute or memory logic. Compared to MPI, while GGAS allows to save about 5% in overall power, the RMA model causes power to increase about 10%. Here, the savings in CPU



(a) 2 nodes



(b) 12 nodes

Fig. 5.17 Energy spending for the N-Body benchmark for various communication models and 2 and 12 nodes

power consumption cannot offset the increase in GPU power.

Combining power results and performance yields the energy spent on particular benchmarks. The results for *N-Body* are shown in Figure 5.17, again for 2 and 12 nodes and various problem sizes. The energy spending seems similar for GGAS, RMA, and MPI at the smaller scale with a slight advantage for GPU-controlled communication. However, increasing the scale to 12 nodes shows clear benefits for GGAS, at least for small to medium problem sizes. Again, GPU-controlled communication is beneficial in any case and allows for energy saving from 10 to 80%.

Figure 5.18 depicts the energy spending per node for the *Himeno* benchmark. Due to linear performance scaling, more nodes solve larger problem sizes in constant time and thus the energy spending per node remains constant as well. Although GGAS' performance is inferior to MPI, significant less energy is spent for 8 and 10 nodes and only a little more is spent for other configurations. RMA's energy is always less than GGAS and MPI. In fact, using RMA instead of MPI allows for energy savings from 8 to 20%. GGAS never increases the energy spending more than 4% and using 8 and 10 nodes, it even allows for savings of up to 13%.

The energy spending for the *Global Sum* benchmark follows the same course as the performance and is shown for 2 and 12 nodes in Figure 5.19. Due to additional communication and synchronization overhead the energy spent per node remains constant as the number of nodes increases. The larger the problem size becomes the less energy can be saved with GGAS as communication is

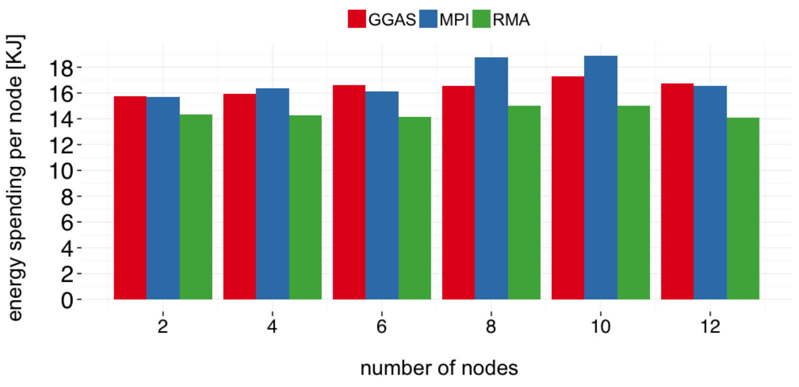


Fig. 5.18 Energy spending for the Himeno benchmark for various communication models at varying scale.

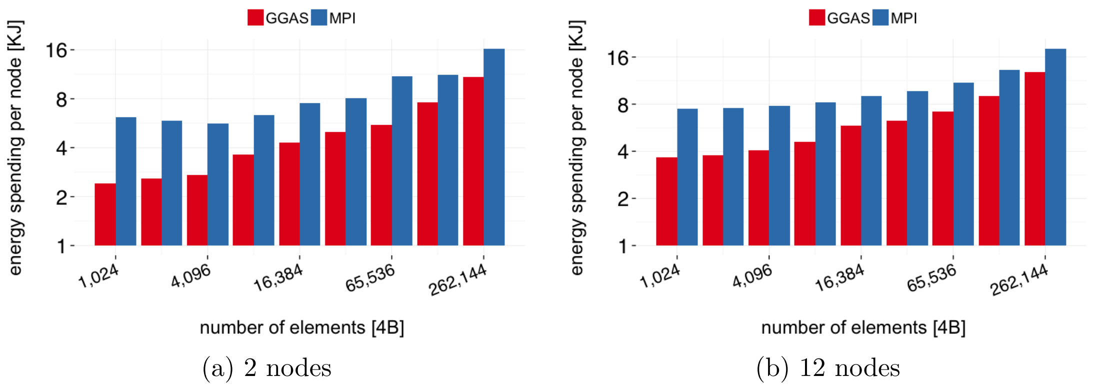


Fig. 5.19 Energy spending for the Global Sum benchmark for GGAS and MPI and 2 and 12 nodes.

outweighed by computation. Nonetheless, energy savings from 20% for large amounts of data and 50% for small amounts are enabled by switching to GPU-controlled communication.

The energy analysis of the *Random Access* benchmark completes this section and the spending is shown in Figure 5.20. GGAS' superior performance allows for tremendous energy savings at any scale, while RMA and MPI show similar spendings at larger scale. On average, GGAS' savings amount to 47%, whereas RMA still allow to reduce the energy spending by 13%, compared to MPI.

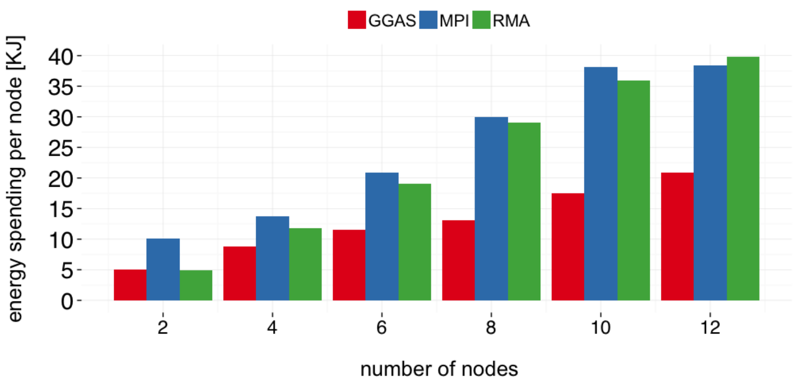


Fig. 5.20 Energy spending for the Random Access benchmark for various communication models and varying node count.

5.6 Summary

The GPU's execution model differs significantly from the CPU and communication primitives like MPI have been tailored and optimized for fast single-thread processors with low-latency memory access. Although GPUs provide tremendous compute power and significantly higher memory bandwidth their integration into computing systems requires to change the programming paradigm accordingly. Existing applications and code cannot run without being substantially rewritten and it divides the code into two branches: code that is executable on the GPU and code that runs on the CPU.

In an offloading model, the CPU owns control of the entire application, but hands off compute-intensive tasks to the GPU where data is processed and results are returned to the host processor. Existing communication models rely on an interplay with the operating system that solely runs on the CPU, thus making it impossible to implement it on the GPU.

Nonetheless, this chapter has introduced models in which the GPU can control communication by accessing networking hardware directly within CUDA kernels. Although communication is still managed on the CPU, directly orchestrating communication on the GPU has shown to be beneficial in terms of performance and energy. Here, it is shown that PGAS models can be orchestrated on the GPU as they do not require much management after memory registrations are set up. Contrary, MPI requires much more management and serves as a proxy for traditional CPU-controlled communication for the analysis of this chapter.

The PGAS model is divided into two basic operations. First, fine-grain memory access that is based on load/store forwarding, referred to as GGAS. This allows GPUs to directly access other GPU's memory at word or cache line granularity. Second, bulk transfers with put/get semantics, referred to as RMA. Here, the data transfer is offloaded to the NIC's DMA engines. These two models are compared to MPI.

The performance analysis shows that GGAS is beneficial for small messages as it occupies only a few GPU resources to copy data. The more data needs to be transferred, the better it is to use RMA as copy operations are executed by the NIC's hardware. In any case, GPU-controlled communication is favorable to traditional MPI. The improvement results from keeping control on the GPU and avoiding additional copies to host memory. While the latter can be avoided with CUDA-aware MPI as well, the consequences from context switches are still prevalent. Not only does it introduce additional latency when kernels have to be relaunched, it also aggravates programmability and is conducive to more heterogeneity.

Energy has evolved to another important metric as many systems are nowadays limited by power and energy constraints. It is shown that GGAS and RMA not only reduce time-to-solution, but also allow to save power by reducing CPU power consumption at only a small increase of GPU power. Hence, energy savings from 10 to 50% are shown for various benchmarks.

This chapter clearly shows the benefits of GPU-controlled communication and strengthens the call for stepping up efforts toward specialized communication models. The following summarizes the insights from this chapter:

Insight I: A global shared address space or PGAS model are completely align with CUDA's programming model.

Insight II: Load/Store forwarding yields the highest bandwidth and lowest latency, especially for small messages.

Insight III: Application that can overlap communication with computation benefit from offloaded models, such as put/get or even CPU-controlled message passing.

Insight IV: GPU-controlled communication is always favorable, however, only when load/store and put/get are used together. Thus, a PGAS model seems best for GPUs.

Insight V: At larger scale, managed communication might become inevitable. In unmanaged communication, the user has to administer buffers and connections as Listings 5.1, 5.2, and 5.3 have shown. With hundreds to thousands of GPUs, the complexity growths significantly when NUMA effects also have to be taken into account.

CHAPTER 6

Managing Communication on GPUs

Besides having access to the network, it is also important to be able to manage communication on the GPU. Managing communication comprises various aspects, often depending on the communication model. This chapter discusses various aspects with respect to the communication model and covers a variety of building blocks. The focus is on the matching of messages and receive requests, which has been published at the *International Parallel and Distributed Processing Symposium (IPDPS)*, 2017 [21] [1].

The question whether communication needs to be managed or not depends on the scale. An example for unmanaged communication is a large shared address space (see also Figure 5.4). Here, each processor maps every other processor's memory into its own address space, allowing for direct load/store access of any data within the system. The user has to know where data resides and care about data movement accordingly, but also administer buffers on each GPU to exploit data locality. Managed communication, such as message passing, allows to explicitly express data locality at an elevated level of abstraction.

6.1 Managing Concepts for GPUs

As it has been shown in Section 5.1.1.2, existing approaches set aside CPU cores to perform communication, however, this is still an unexplored area for GPUs.

[1]The paper received a best paper award.

CPU solutions provide two different hierarchies: first, intra-node communication to instruct a communication entity about communication tasks. This is usually implemented as shared memory queues. Second, inter-node communication between the communication entities, which currently comprises PCIe and NVLINK for GPUs, or NIC-assisted models as described in Chapter 5. While GPUs are inherently different from CPUs, the principles of communication onloading remain the same, however, there are various design decisions regarding the granularity.

One fundamental question to be answered is the granularity at which communication can be triggered on the GPU. Another open question concerns the underlying communication architecture that is presumed, on which the communication process is built.

6.1.1 Kernel-level Management

One approach is to launch two separate kernels, one for the application and another to perform communication, which is depicted Figure 6.1. It would also be possible to have multiple application kernels, served by a single communication kernel. Intra-node communication (❶) takes place between the kernels, through global memory, and inter-node communication (❷) is handled by the communication kernels only. Similar to CPU approaches, ❶ can be implemented as a queue, residing in global memory, whereas ❷ can be any inter-GPU communication substrate, possibly another queue accessed within CUDA's UVA space.

However, this approach has a significant drawback with current GPU architectures. The scheduling of kernels in the current and official CUDA release is transparent and the user has no control over the timing when kernels are brought to execution, and SMs, to which CTAs are assigned. Although CUDA implements stream priorities, it is not guaranteed that higher prioritized kernels actually run first. Even the order in which kernels are launched does not guarantee that both kernels run concurrently or that the first kernel is scheduled first. Without any additional control over the scheduling, this solution is not applicable and deadlock-free.

6.1.2 CTA-level Management

Instead of launching a separate communication kernel, each application kernel can dedicate one or a few CTAs to communication tasks. This is depicted in

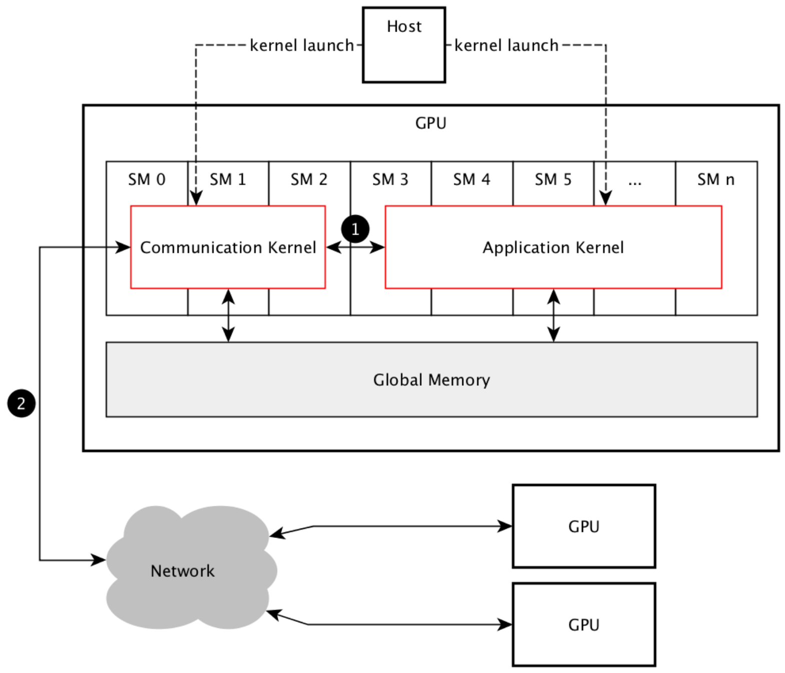


Fig. 6.1 Separate communication kernel that runs along with the application. Instructions are received through global memory.

Figure 6.2. The first CTA that is running on an SM takes over the communication role, while all others belong to the application. From an application's point of view, this approach is similar to the aforementioned approach with a separate kernel, as one or a few CTAs serve the application's communication requests.

6.1.3 Warp-level Management

A third, and most fine-granular approach, is depicted in Figure 6.3. The communication is distributed across all CTAs, each dedicating one or a few warps to communication. The main advantage of this approach is that the communication entity is ensured to be running with the CTA that is executing the application. However, the application has to spare at least one warp per CTA that is entitled to request communication. These warps are then not available to the application.

6.1.4 Software NIC Specification

In order to assess different design options, the tasks of such a communication entity have to be defined. Similar to a NIC, the communication entity has to

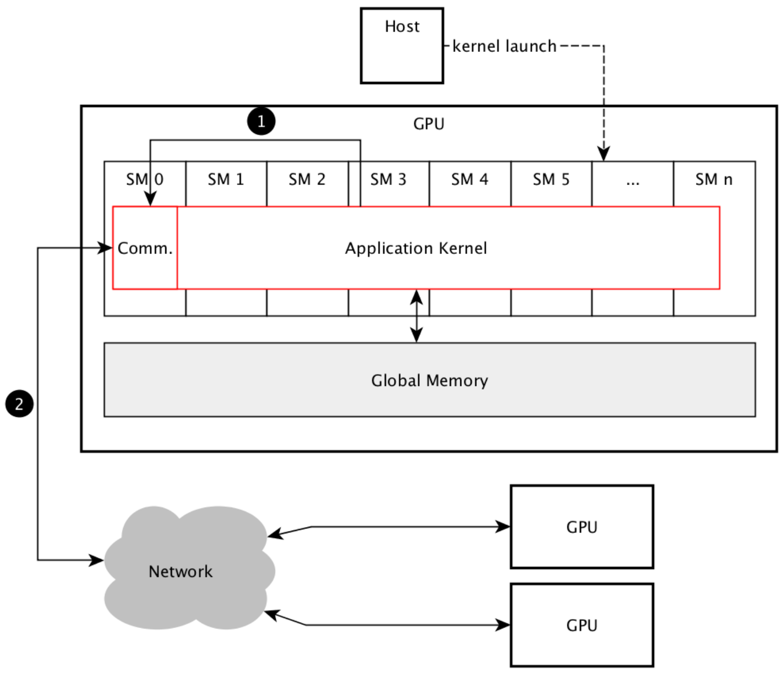


Fig. 6.2 Communication CTAs as part of the application. Instructions still need to be passed through global memory.

possess the following capabilities. The communication entity is referred to as *SoftNIC* (Software NIC) in the following.

- **Work generation:** the SoftNIC needs to receive work requests from the application. A work request comprises any communication related task, such as *send or receive a message*, *expose memory to the network*, or *query if a new message is available*. This requires a control and data path between the SoftNIC and the application, annotated with ❶ in Figure 6.1, 6.2, and 6.3.

- **Work execution:** upon receiving work requests, they need to be executed. Here it depends on the type of work the application is requesting and the communication model the SoftNIC is implementing. Work execution comprises, but is not limited to, *data transfers* (see ❷ in Figure 6.1, 6.2, and 6.3), *memory allocations and de-allocations*, *collectives*, *active messages*, *message matching*, *synchronization*, and many more. However, there are also tasks the SoftNIC is associated with that do not require explicit triggering. For example, address translation, routing, other resource allocations, such

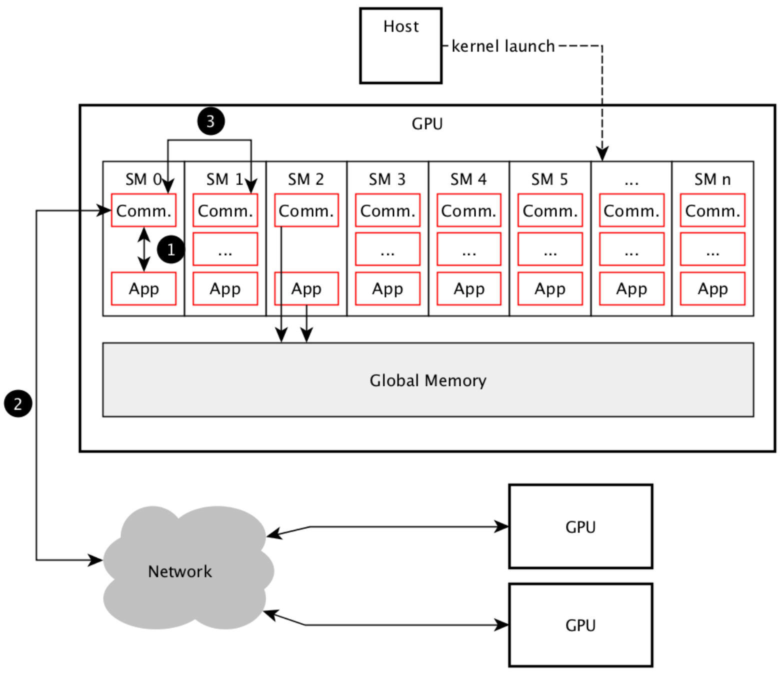


Fig. 6.3 Each CTA has its own communication warp, thus instructions can be passed through shared memory. Communication between these specialized warps still requires global memory.

as compute or I/O resources, or intra-SoftNIC communications, such as ❸ in Figure 6.3.

- **Work completion:** while work generation can be seen as the interface from the application to the SoftNIC, work completion needs a control and data path in the opposite direction. Both blocking and non-blocking communication requires the application to have information about work requests being completed or still being processed. Hence, the SoftNIC has to notify the application about *unconsumed messages or data*, the *completion of work requests*, or the *release of send buffers that are now available to be rewritten again.*

In the following, these items are discussed in more detail in regard to their implementation and applicability to various communication models. Various building blocks are introduced that are eventually needed to compose a SoftNIC.

6.2 Work Generation

Any time the application wants to communicate, a work request has to be generated and passed to the SoftNIC. In a CPU-centric world, the interface is usually implemented as a queue or list, mainly because these data structures preserve ordering and are rather simple to implement. However, there is no general definition of a work request and it may depend on the communication model. For example, two-sided message passing might see a work request as a send or receive operation, while one-sided communication would map put or get operations onto work requests. Nonetheless, the interface between SoftNIC and application remains the same and is discussed in the following.

6.2.1 Queues on GPUs

A queue is a fundamental abstract data type that is either implemented as a linked list or circular buffer. An important characteristic is that queues preserve ordering, meaning that elements are consumed in the order they were produced. *Insert* and *remove* are executed in constant time, thus $\mathcal{O}(1)$ complexity. On CPUs, queues have been studied extensively and are available in C++'s STL library, for example. Furthermore, there are some attempts to implement queues on GPUs.

6.2.1.1 Related Work

One of the first papers to study queues on GPUs is from Cederman et al. [102]. Their work discusses if and how well common CPU queuing algorithms, including lock-based and lock-free queues, can be implemented on graphics processors, relying on improved atomics of the Kepler GPU architecture. The authors categorize their designs into Single Producer Single Consumer (SPSC) and Multiple Producer Multiple Consumer (MPMC) queues, each implemented with different approaches. In case of the SPSC approach, one CTA dedicates a thread for the role of the producer and a second CTA provides the consumer thread. Their experiments show that using the Tesla 2050 (Fermi architecture), the *Buffered BatchQueue* [103] performs best, with 2.5M operations/s. Here the queue is divided into batches, each exclusively accessed by the producer and consumer, respectively. Additionally, the batches are buffered in shared memory to minimize access latencies. The benchmarks for the MPMC has 25% of the threads per CTA producing elements, and the remaining 75% consume elements. The number of CTAs is being varied. Regarding lock-based queues, the *dual-spin lock* approach showed the best performance, while the so-called *TZ queue* (named after Tsigas and Zhang) [104] seems to be superior to the lock-free queues, achieving 600K operations/s at high contention. Although the paper is not clear about the implementation of the dual-spin lock, it is assumed that the queue is locked by busy-waiting on write and read access. The TZ queue is an array-based queue using atomic Compare-And-Swap (CAS) operations to modify read and write pointers, respectively. It is also shown that lock-free queues scale better and are superior at high CTA counts, while lock-based queues still perform better at small scale.

Scogland et al. [105] continue this study and propose a GPU optimized design, but they also assess the performance of atomics on CPUs and GPUs. Interestingly, the performance of atomic Fetch And Add (FAA) operations is about 7 times higher than successful CAS operations on NVIDIA's K20 GPU. Unlike CPUs, the performance remains constant with an increasing number of threads. The authors propose a ticket-based queue, in which a thread increments a ticket counter before it is granted access to the queue. Only if a transaction counter equals the value on the ticket, the thread is granted permission to enqueue its element. After completing its operation, the thread increments the transaction counter to signal another thread to proceed. Performance is assessed with a benchmarks that uses an equal number of producing and consuming threads. On NVIDIA's K20 GPU, their ticket-based queue achieved a throughput of 256M elements/s. They further mention that they introduce some random work between queue operations to lower contention, otherwise performance is about 10% lower.

Although ticket-based queues seem to perform quite well, they cannot distinguish between an *empty* and *full* state. Generally, existing queue implementations on GPUs try to port CPU queues to the GPU, following a thread-oriented model. However, GPU threads are different from their counterparts on CPUs, and a warp-oriented model is more appropriate as it aims to avoid issues with divergence.

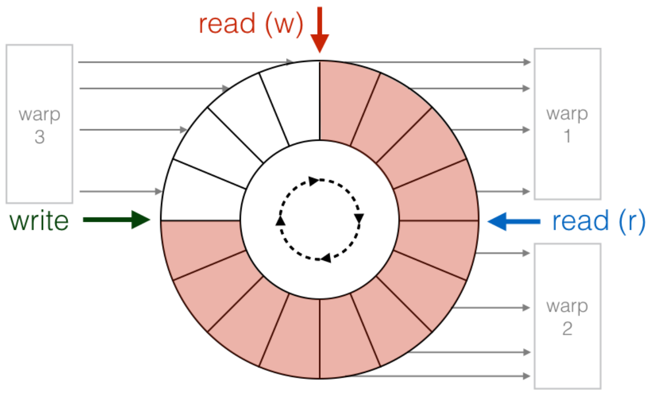


Fig. 6.4 Circular buffer to implement the queue data structure. Two read pointers are used to avoid race conditions between multiple readers. Each data element comprises a valid identifier to avoid race conditions between writers and readers.

6.2.1.2 Warp-parallel Queue

Rather than seeing threads as individual producers and consumers, warp-parallel approaches result in better efficiency by avoiding threads within a warp to diverge during queue operations. For example, assuming a warp size of 32 threads, one warp can enqueue one data element that is composed of 32 sub-elements. Alternatively, 32 threads can collaboratively enqueue one element each. This perfectly complies with the GPU's execution model and increases efficiency.

6.2.1.3 Implementation

An example of a warp-parallel queue is shown in Algorithm 1 and 2. Queue operations are divided into two main parts: first, obtaining a valid queue index and second, writing or reading to or from the queue, respectively. The first part needs to ensure that the queue can fit the amount of elements to be enqueued and that no other warp obtained the same index. Race conditions can be avoided with different techniques and need to be considered between producers and consumers themselves, but also between each other. Thus, besides a write index a *valid* field indicates if a data element is valid or stale. The write index is advanced atomically. Both mechanisms together avoid race conditions between multiple producers and producers and consumers.

Regarding the dequeue operation, it has to be ensured that data has been consumed before the read index is advanced, thus a second read index is introduced here. The second index is read and incremented atomically to avoid race

Algorithm 1 warp parallel enqueue operation

```
register: s ← global: size
do
    register: r ← global: read-index
    register: w-old ← global: write-index
    register: elements := (r - w-old) mod s
    if elements $\leq$ count and 0 $\neq$ elements then
        return: Error
    else
        elements := warp-size
    end if
    w-new := (w-old + elements) mod s
    if r == w-new then
        return: Error
    end if
    if 0 == warp-thread-id then
        update := atomicCAS( global: write-index, new-w)
    end if
while 1 $\neq$ ___ballot(update)
register: index := (w-old + warp-thread-id) mod s
global: queue[ index ] ← data
___threadfence
global: valid[ index ] ← true
```

conditions between multiple consumers. An example is depicted in Figure 6.4, in which two warps are dequeuing and one is enqueuing elements. *read (w)* is the original read pointer that indicates how far the write pointer (*write*) can advance without overwriting unconsumed data. The second read pointer, annotated with *read (r)*, is only used by consumers to set a starting position for the dequeue operation. When the data is consumed, *read (r)* can only be advanced if *read (w)* equals *read (r)* before it was incremented.

Again, the main advantage of this implementation is that it avoids divergence. Only read and write indices are incremented by a single thread, while everything else is done warp-parallel. Reading and writing to the queue also takes advantage of the memory access coalescing to minimize internal bus traffic.

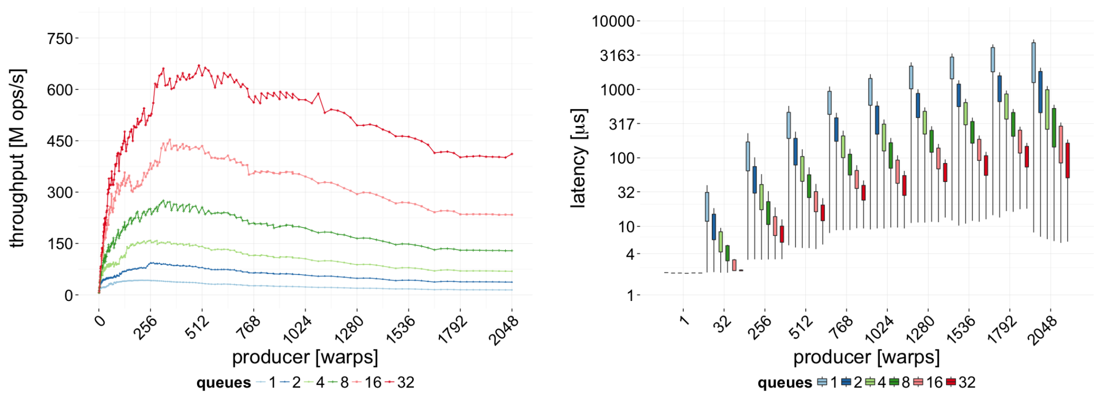
6.2.1.4 Performance

The performance of queues is assessed by two metrics: issue rate and latency. However, both metrics depend on a vast amount of parameters, such as the number of producers and consumers, queue size, or size of queue elements. A large

Algorithm 2 warp parallel dequeue operation

```
register: s ← global: size
do
    register: r-old ← global: read-index
    register: w ← global: write-index
    register: elements := (w - r-old) mod s
    if old-r == w or 0 ≠ elements then
        return: Error
    end if
    elements := MIN( elements, warp-size )
    r-new := (r-old + elements) mod s
    if 0 == warp-thread-id then
        update := atomicCAS( global: read-index, r-new)
    end if
while 1 ≠ ___ballot(update)
register: index := (r-old + warp-thread-id) mod s
if warp-thread-id ≤ elements then
    while global: valid[ index ] == 0 do
    end while
    register: data ← global: queue[ index ]
    global: valid[ index ] ← 0
end if
___threadfence
register: update := false
do
    if warp-thread-id == 0 and global: read-index == r-old then
        global: read-index ← (r-old + elements) mod s
        update := true
    end if
while 1 ≠ ___ballot(update)
```

number of producers or consumers causes significant contention as every entity issues an atomic operation to the same memory address, either write or read index. Thus, operations are serialized, or to be precise, a significant amount of CAS operations fail and associated instructions have to be executed again. This overhead can be reduced by introducing multiple queues, which distribute atomic memory accesses across multiple memory locations, thus reducing contention and serialization. In case of a static mapping between producing and consuming entities and queues, the order of elements from the same source is still preserved as one producer always enqueues elements to the same queue. This also enables load balancing techniques, for example, producers could enqueue elements to the



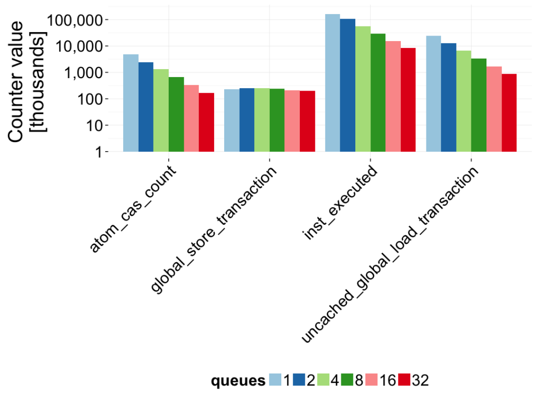
(a) Issue rate (b) Latency

Fig. 6.5 Issue rate and latency for the warp-parallel queue. Performance relates to the Pascal GTX1080 GPU.

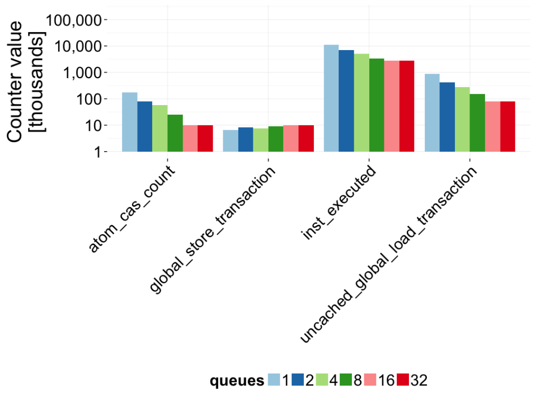
shortest queues, and consumers dequeue from the queue that contains the most elements.

Figure 6.5 shows the throughput and latency for the enqueue operation of the warp-parallel queue on the Pascal-class GTX1080 GPU. Each data point is the average of 100 iterations and the kernel launch overhead is eliminated. The issue rate as well as the latency scales well with the number of queues, a direct result from reduced contention. The peak issue rate amounts to about 600M operations/s, translating into a bandwidth of 2.4GB/s for 32-bit integer types. Experiments also showed that the bandwidth can be improved to about 25GB/s with 128B data types. Linear memory copies, for example, achieve up to 350GB/s, hence the queue is about 14 times slower in terms of bandwidth. This is due to the overhead associated with obtaining a valid queue index.

As shown, the performance of addressing multiple queues is significantly better than a single queue due to reduced contention. Figure 6.6 depicts the number of atomic operations and memory instructions for various queue sizes. Furthermore, a comparison between a thread-parallel and the warp-parallel enqueue operation is shown. For the thread-parallel approach, a mapping in which the right queue is selected based on the thread ID is implemented. For example, thread 0 enqueues its element to queue 0, thread 5 to queue 5, and if there is a total of 32 queues, thread 33 would enqueue to queue 1 again. The warp-parallel approach selects the queue based on the CTA ID. In addition, only one warp per CTA is used in both thread- and warp-parallel approaches with



(a) thread parallel



(b) warp parallel

Fig. 6.6 Various performance counters for the thread- and warp-parallel queue. Counters relate to the Kepler K80 GPU.

a total of 128K CTAs, and the queue is sufficiently large so that no dequeue operations are required.

While the number of global memory store operations remains the same regardless of the number of queues, the number of global memory load and atomic CAS operations decrease significantly in both approaches. However, the number of operations is about 10 times lower for the warp-parallel approach. These results are plausible as only the first phase, in which the right index is obtained, suffers from contention and only requires load and atomic CAS operations, while the second phase, writing the data to the queue, simply issues store operations to the previously reserved addresses. The fact that the total number of global store transactions is lower for the warp-parallel approach is another advantage over the thread-parallel approach. Every thread within a warp obtains the same index at the same time and then uses its own warp-internal thread ID to determine the address to which the thread writes the data, thus the memory controller can combine these accesses into a few transactions. In the thread-parallel approach, on the other hand, each thread obtains its own index, possibly at different times, thus the memory store operations can hardly be coalesced.

The dequeue operation also benefits from multiple queues and warp-parallelism. Instruction-level profiling shows that with a queue size of 128K elements and 32 CTAs, dequeuing elements with one warp each, reduces the number of atomic CAS operations by a factor of 20 for 32 queues, compared to a single queue. Uncached load operations are reduced by a factor of 8. On the Pascal GTX1080, the peak dequeue rate of 450M operations/s is achieved with 320 warps and 32 queues, thus 10 CTAs with 32 warps each. A single warp yields a dequeue rate of 4.7M operations/s and one CTA with 32 warps achieves 155M operations/s. The lowest latency of a warp-parallel dequeue operation amounts to $3.2\mu s$, but can rise up to thousands when many dequeuing warps contend for queue access.

6.2.2 Specialized Queuing for the SoftNIC Concept

While the previous section shows the general performance of the warp-parallel queue, the SoftNIC's use case is a bit different. Depending on who is sourcing and sinking network traffic, e.g. warps, CTAs, or kernels, the number of producers and consumers vary. If communication is handled by an independent kernel, as depicted in Figure 6.1, the SoftNIC could comprise multiple CTAs, perhaps even adaptive to the application's communication demands. However, this requires a compromise between how many resources are occupied by the SoftNIC and not available to the application, and the SoftNIC's performance. Similarly, this also applies to the approach shown in Figure 6.2, in which the application dedicates a certain number of CTAs to communication. The more CTAs are detached from the application, the more resources are available to the SoftNIC. In summary it can be said that these approaches allow communication requests to be consumed at a peak rate of 450M requests/s if the SoftNIC is assigned sufficiently enough resources, or about 150M requests/s for a single CTA, limited by the queue's dequeue rate. However, this only comprises the intra-node work generation aspect and neglects execution and completion.

In the distributed SoftNIC approach, illustrated in Figure 6.3, it would not make sense for the application warps to submit their requests through global memory, but instead shared memory may be used. A global memory queue is only required for intra-node communication between SoftNIC warps. Generally, shared memory provides an order of magnitude higher bandwidth than global memory, but also minimizes access latency. For example, while global memory accesses can easily take hundreds of cycles, a shared memory request is usually served in less than 10 cycles, thus enabling fast queuing operations.

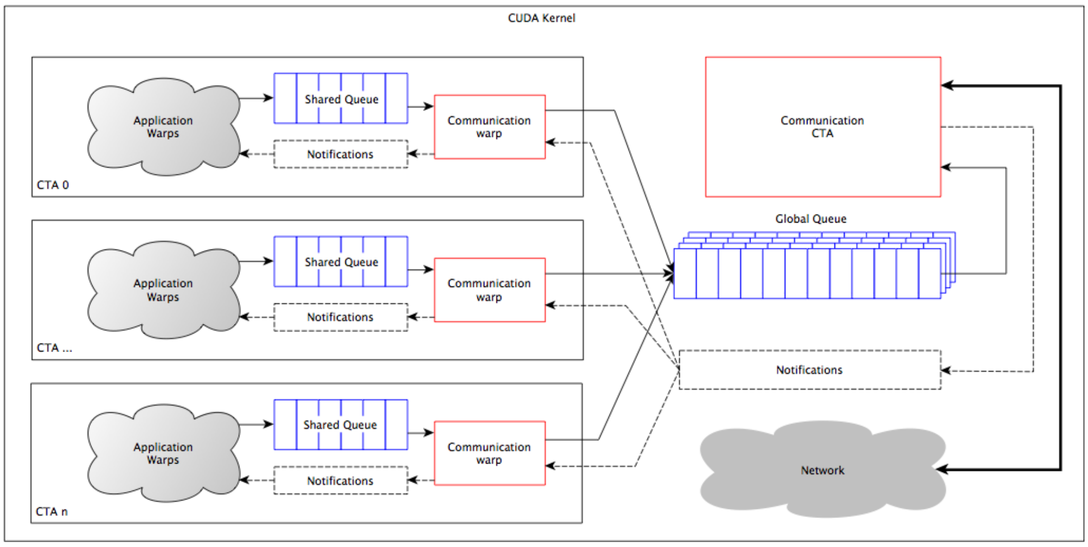
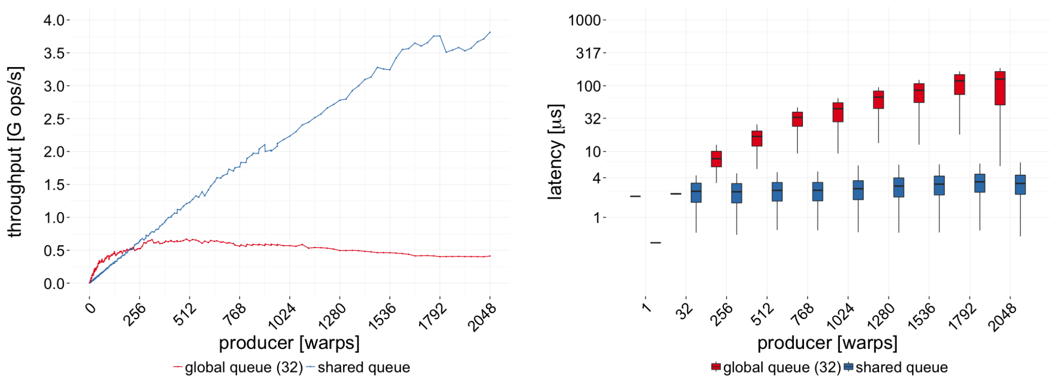


Fig. 6.7 Shared memory queue approach to allow for low-latency access. Communication warps ensure elements are passed between shared and global memory queues.

6.2.2.1 Shared Memory Queuing

An approach similar to *cudaDMA* [106] can be used to dedicate one warp per application CTA to serve requests from the application and communicate with other communication warps through the global memory queue. Additionally, a dedicated communication CTA can be used to perform the bulk of communication tasks, while scattered communication warps specialize on light-weight tasks.

Figure 6.7 illustrates this approach. The application submits work requests to a shared memory queue, from which it is forwarded to the global queue. Also, a notification mechanism is provided locally within a CTA to inform the application about completed work requests or new received data. This is going to be discussed in Section 6.4. In cases with rare communication, this approach might not be beneficial as most warps are idle. However, it might work quite well for applications that allow for overlap between computation and communication. For example, the application can submit a work request to send already computed halos in a stencil code and then continue with the computation of the inner grid. The shared queue allows to reduce the latency that is required to submit the work request, but there is enough time during computation in which the communication warp can forward the request to the central communication CTA. Also, irregular applications that dynamically spawn new work map quite well to this approach. The performance is depicted in Figure 6.8 and compared to the



(a) Issue rate (b) Latency

Fig. 6.8 Throughput and latency of the shared memory queue vs global memory queue. Results relate to the Pascal GTX1080 GPU.

global queue approach.

Accessing the shared memory queue is many times faster than the global queue. In contended situations, the shared queue provides up to 8 times higher throughput. The fact that the shared queue's throughput is lower when only a few warps access the queue is due to the atomic pointer arithmetic. While the global queue is split into 32 queues to reduce the number of same-address atomic updates, the shared queue is a single queue, meaning the atomic CAS updates the same address. Due to the limited amount of shared memory space, the queue should not be split up like the global queue. Regarding latency, the shared queue is accessed up to 30 times faster. As aforementioned, this can improve performance significantly if communication is well overlapped.

6.2.2.2 En- and Dequeue Granularity

Another interesting aspect is the applicability of warp-parallel queue operations. Unless a CTA wants to send 32 messages, most enqueue operations might contain only one or a few requests. In some cases requests might be aggregated before submission. On the other hand, as long as the queue contains enough elements, the SoftNIC always wants to dequeue as many elements as it can process concurrently. Thus, the dequeue operation is likely to benefit from warp-parallel operations. Enqueuing less than 32 elements per operation reduces the throughput accordingly. For example, enqueuing only 16 elements causes the throughput to drop by about half. The latency, however, remains the same

Table 6.1 K80, throughput in elements/s (element equals integer value)

		shared queue	**global queue (1 queue)**
enqueue	31 warps ; 1 request per warp	1,500K	700K
	1 warp ; 32 requests per warp	12,800K	7,000K
dequeue	31 warps ; 1 request per warp	670K	470
	1 warp ; 32 requests per warp	15,605K	5,221K

as it is mainly limited by obtaining a valid queue index, which is independent on the number of work requests to be submitted or fetched. Table 6.1 shows enqueue and dequeue rates for shared queue and global queue accesses. Two different scenarios are shown: first, 31 warps with 1 element per warp and second, 1 warp with 32 requests per warp. This represents the approach in Figure 6.7, in which the communication warp uses warp-parallel queue operations, but it is assumed that application warps do not want to submit more than one work request. As can be seen, the warp-parallel global queue access is fast enough to satisfy demands of the remaining 31 warps, given they only hold one element each. This makes this approach a viable and promising solution if communication is handled on warp level.

A rather special case is communication on kernel level. Here, only one CTA can generate communication requests, thus a separate SoftNIC entity might not be needed. Instead, messages can directly be sent to other kernels, running on different GPUs.

6.2.3 Hardware Optimized Queuing

As shown, the queue data structure is essential to communication management, especially with an underlying globally shared address space. However, it seems the GPU's architecture and execution model does not allow for efficient queues, mainly due to high latency memory accesses and little amount of parallelism that can be exploited for en- and dequeue operations. Thus, adapting and extending the GPU's hardware capabilities seems inevitable.

The main bottleneck with GPU queues is to obtain read and write indices and atomically advancing them to reserve entries. Since all following operations have to wait until the indices are loaded to registers, the warp is stalled for hundreds of cycles. Additionally, contention causes the atomic CAS to fail and indices have to be obtained again, stalling the warp for another hundreds of cycles.

6.2.3.1 Design Space Evaluation

There are various options to tackle the queuing issue and the most intuitive approach is to extend the instruction set. The enqueue operation, for example, requires to advance the write index only if the queue is not full, hence read and write index differ. This could be solved by a Double Compare-And-Swap (DCAS) operation [107], which compares two addresses and only updates the write index if both conditions evaluate to true. Another example is IBM PowerPC's Load-Link/Sore-Conditional (LL/SC) instruction [108], which reserves a register and stores only succeed if the reservation is still valid. However, both approaches still stall en- and dequeuing warps if operations do not meet conditions and thus have to be executed again.

It seems insufficient to add just another general instruction in order to support queuing operations more efficiently on GPUs. Instead, the memory controller needs to be extended by explicit queuing operations. One approach is to have the memory controller to administer the queue. Producer can simply write their data to the memory controller, while consumer read data from it. Indices are atomically advanced. However, it needs to be defined what happens when the queue is full and elements are written to the queue, and when the queue is empty and elements are requested.

Although this approach optimizes queuing operations, polling on the queue still results in memory bus traffic and stalling warps. Hence, the polling needs to be moved closer to the SM. Consequently, the SM is also extended by specialized hardware that communicates with the queue entity of the memory controller.

The entire approach [109] is described in the following. The evaluation is based on simulation with *GPGPUSim* [110].

6.2.3.2 Server/Client Queue Controller Approach

As already indicated, the GPU hardware needs to be extended at two places. First, the memory controller is extended by a so-called *Queue Controller Server*. The server manages queue indices and serves enqueue and dequeue operations coming from the SMs. Second, the SM itself is extended by a *Queue Controller Client*, which forwards operations to the server and aims to reduce polling on far resources.

Similar to MMIO, the queue controller provides a range of addresses that can be accessed by plain memory operations such as loads and stores. When such an address is accessed, the appropriate queue operation is triggered. The client serves as a filter to avoid polling threads to issue too many memory bus transactions and eventually reducing the available memory bandwidth to the application, which is running along with the SoftNIC.

The concept is depicted in Figure 6.9 and comprises three main parts. First, the queue itself is allocated in GPU memory. This allows for flexibility as the user can allocate as much space for the queue as needed by the application. Second, the server is placed next to the L2 cache controller. However, the L2 cache is divided into multiple parts, each covering a certain address range of the global memory. If only one controller is extended by the queue server the queue also has to reside in the part of the global memory it covers, thus limiting the size of the queue. If significantly more space would be needed the server has to move back to the memory controller as opposed to extend a single cache controller. Last, each SM needs to be extended by a queue client with access to shared memory, in which the *index table* is placed. Indices are used by threads to directly enqueue elements without having to fetch indices from global memory and to calculate the correct offset for the absolute address.

A CTA can register as a producer, consumer, or both. During registration, the CTA conveys the number of required indices to the client, which are then fetched from the server and entered into the SM's index table. When indices are consumed the client ensures to fetch more if the queue has enough space left until the CTA deregisters as client. At the same time, CTAs need to invalidate already consumed indices.

The client and server communicate through an index status table, which tracks indices that are handed out to the clients. This is necessary to avoid the same indices to be sent to multiple clients. If the CTA produces values, the queue controller provides the CTA with indices to which the data can be written to. On the other hand, consuming CTAs are provided with an index and the number of valid elements in the queue.

6.2.3.3 Simulation Results

Using *GPGPUSim* [110], the queue controller approach can be evaluated and compared to software approaches, which use an atomic CAS, for example. The enqueue and dequeue rates are depicted in Figure 6.10.

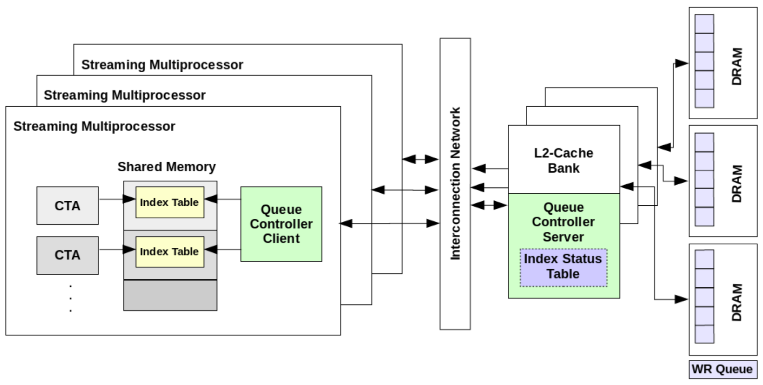
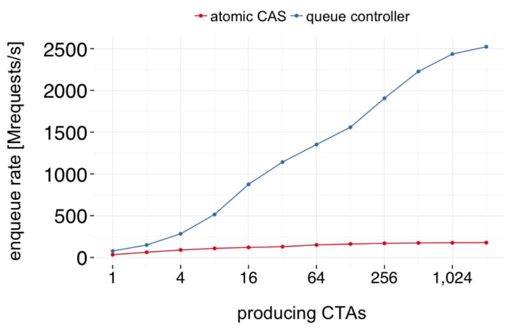
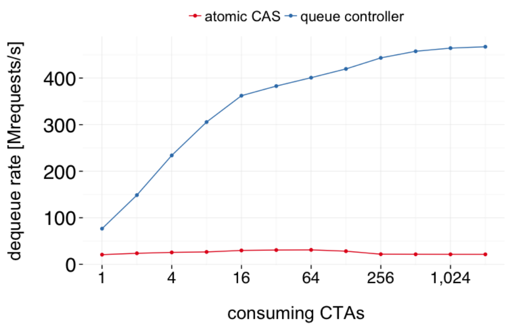


Fig. 6.9 Hardware extensions to support queue operations in GPUs [109]. The queue controller is divided into a server, extending the L2 cache controller, and a client, extending the SM.



(a) enqueue rate



(b) dequeue rate

Fig. 6.10 En- and dequeue rate for the hardware-accelerated queue approach [109].

The results show a tremendous improvement in terms of enqueue and dequeue rate. The queue controller allows CTAs to enqueue requests at up to 2.5G request/s, which is a speedup of 14x over the CAS approach. Furthermore, the queue controller scales well with an increasing number of CTAs, while the CAS implementation stagnates at about 16 CTAs.

Although this particular implementation aims to improve intra-GPU queuing, the same principle could be applied to remote queuing that is required to exchange messages, for example. Summarizing it can be said that hardware supported queuing is important for the SoftNIC to achieve high performance, but also to be able to scale to a large number of nodes.

6.3 Work Execution

When work requests are received by the SoftNIC, various actions are triggered, depending on the type of work that is requested. The types the SoftNIC has to support depend on the programming model. An universal implementation may be possible, but also causes overhead. Thus, specialized SoftNIC features need to be compiled as needed, depending on the model that is layered on top. The following presents various features and assesses their application in various programming and communication models.

6.3.1 Data Transfer

The most fundamental communication work is to transfer data between two memory locations, e.g. two physically separated GPUs. In offloaded communication, in which NICs perform data transfers to take load off the processor, the NIC's DMA engines are instructed to copy the data over the network and into the target's memory. In the absence of a NIC, the data transfer needs to be handled by the GPU as well and it ultimately depends on the network, with which the GPUs are interconnected. For example, NVLINK provides a non-coherent global shared address space, thus data transfers are essentially put/get operations as target addresses are part of the source's address space. However, this becomes problematic as the entire network's memory can hardly be mapped into a GPU's address space, rendering address translation an important problem to solve. Nonetheless, this is going to be discussed in the next chapter and for now a PGAS system is assumed, in which the target address is known at the source.

The data transfer between GPUs within a common address space can be performed either by using copy engines or the SM's load/store units. While the copy engines serve as an offload unit, the SM's resources would be occupied for communication, thus being unavailable to the application. Furthermore, the SoftNIC's allocated resources may be insufficient to perform a large number of data transfers. However, small data volumes or certain patterns may still benefit from SM-based copy operations.

The bandwidths achieved with an SM-based and copy engine data transfer are depicted in Figure 6.11, determined for communication between the two

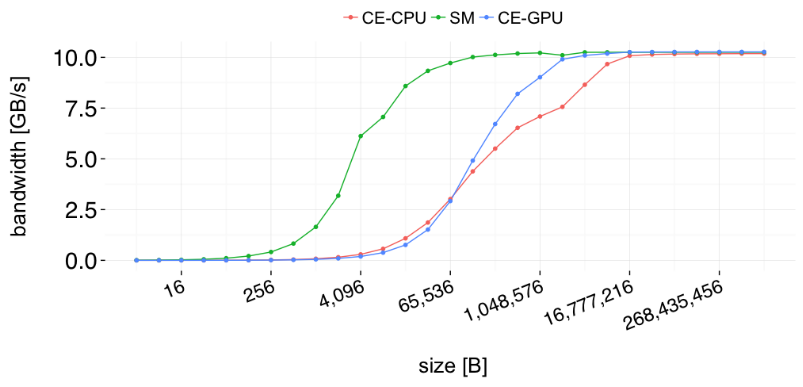
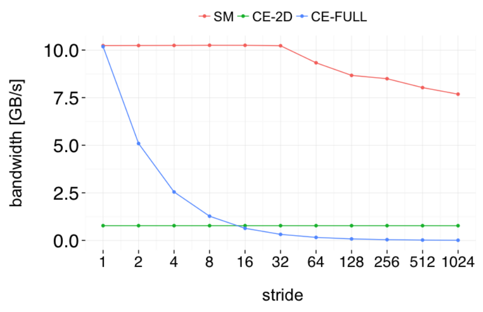


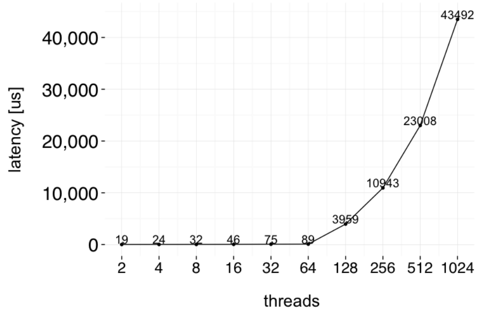
Fig. 6.11 Bandwidth for various copy approaches, including the SM and copy engine. The data was copied between the two internal GPUs of the Kepler K80.

on-board GPUs of the Kepler K80 card. Clearly, the SM-based copy outperforms DMA approaches by far in terms of bandwidth. Transfers larger than 16MB do not show any preference, while GPU-controlled DMA transfers perform slightly better than their CPU-controlled counterpart. It is worth mentioning that the SM-based approach utilizes the whole GPU and assigns each data element to one thread. However, a much more realistic approach is to restrict resources to a single SM. Here, the results in Figure 6.12 show a peak bandwidth of 10GB/s, thus one SM is sufficient to satisfy PCIe 2.0. Systems that support faster GPU communication, for example through NVLINK, might show different results as it is expected that a single SM is not sufficient anymore. Furthermore, the data type used by each thread influences the bandwidth as Figure 6.12 suggests. The `int` type scales linear, but using CUDA's vector types, such as `int2`, allows to increase the bandwidth by up to 40%.

Another important type of data transfers is non-linear copies as often found in multi-dimensional data transfers. Here, not all elements need to be copied and some elements are bypassed. Figure 6.13 shows the bandwidth for various strides. For example, a stride of two means every other element is skipped. Once again, the SM-based copy approach can handle stride transfers remarkably efficiently with almost no loss in bandwidth except for strides larger than 32. Here, cache misses reduce performance while even higher strides also result in TLB misses. The copy engine, however, cannot keep up and is outperformed by far. As the second graph shows, many concurrent accesses significantly increase the access latency of the copy engines, thus the data transfer becomes inefficient. Using the two-dimensional interface cannot solve this issue as the bandwidth remains



(a) copy engine and SM data transfer for 4GB data volume and varying stride



(b) copy engine latency when called within a kernel

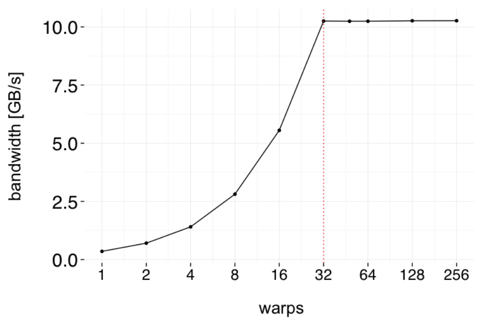
Fig. 6.12 Bandwidth of stride patterns for the copy engine compared to the SM copy approach. The right graph also shows the latency of copy engine accesses within the CUDA kernel on the GPU. Measurements are performed on the Kepler K80.

low, although constant and independent of the stride. However, this is mostly due to the data layout and row-major ordering. Column-major ordering might increase performance significantly, especially for small stride values. Summarizing it can be said that non-linear transfers are best executed by the SM itself. An alternative would be to pack the data to a continuous buffer which is then copied by the copy engine and unpacked at the destination. However, this still requires the SM to execute kernels for the pack- and unpack operation. For example, this approach is used by Wu et al. [94] to enable support for non-contiguous data types in OpenMPI.

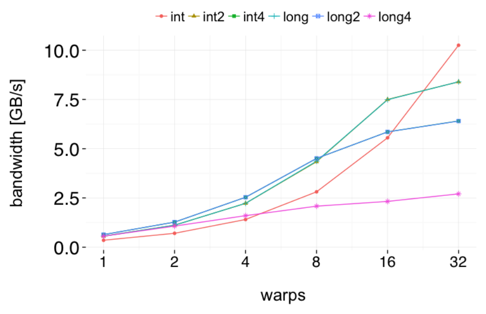
Handling data transfers on the GPU is eminently important and as experiments show still lacks enough support from the hardware and software architecture. Although the SM is capable of copying data at high bandwidth and efficiency the occupied resources become unavailable to the application and it remains doubtful if it can satisfy the application's communication demands. The copy engines, however, cannot be accessed efficiently and cause warps to stall for a long time, especially if copy engines are accessed concurrently by a large number of threads.

6.3.2 Messaging

The ability to send and receive distinct messages is required for two-sided message passing, similar to MPI on CPUs. However, many of MPI's features are sequential



(a) Bandwidth depending on the number of warps. The dotted line marks a single SM, whereas more warps are scheduled to multiple SMs.



(b) SM copy bandwidth with various data types

Fig. 6.13 Bandwidth for varying the number of warps in the SM copy approach. The right graph depicts the bandwidth for various data types. Measurements are performed on the Kepler K80.

or unpractical on GPUs, due the GPU's different execution model. For example, messages have to be matched with receive requests and memory needs to be dynamically allocatable. While the matching is a sequential task, dynamic memory allocation is poorly supported on GPUs.

6.3.2.1 Sending and Receiving Messages

Sending a message can be a viewed as submitting a work request to the SoftNIC, which instructs to transfer data and send a notification to the target processor. It has to be ensured that the data transfer is completed before the notification is received by the destination. Receiving a message is another work request that is submitted to the SoftNIC, which eventually triggers a notification when the message is matched. Notifications are going to be discussed later in this chapter.

As opposed to a single global address space, only a small part of the target's address space needs to be mapped locally and can accessed by shared memory semantics, hence significantly reducing the size of a node's address space. For example, a queue can be placed in the shared part of the GPU's memory, allowing others to enqueue messages. However, the aforementioned queue implementation would not be performant on remote memory. Hence, a different approach is necessary that avoids fetching queue elements from remote memory at high latencies.

6.3.2.2 MPI Compliant Matching

The message matching is an important aspect of two-sided communication and inherently sequential in MPI's protocol, for example. This is mainly due to wildcards and the preservation of ordering. However, this renders it difficult for GPUs to perform well and efficient at this task. For example, a queue-based matching is not feasible since elements cannot be arbitrarily removed without reordering subsequent messages accordingly. Lists, on the other hand, require too much synchronization when elements are removed. Thus, more GPU-friendly algorithms need to be found that exploit more parallelism.

Algorithm As opposed to CPU matching algorithms, both PRQ and UMQ are not separated but kept at the beginning of the receive request and message queue, respectively. The queues reside in global memory. The algorithm to match receive requests and messages is then separated into two phases. First, a matrix is generated, wherein each row represents one message, and each column one receive request. This step is referred to as *scan* phase in the following. Note that rows and columns are interchangeable since the matching is symmetric. The second phase reduces the matrix to a vector, assigning a matching message to each receive request, thus referred to as *reduce* phase in the following. Technically, a third phase is necessary to compact the queues and eliminate already matched elements, however, it is assumed that all messages and receive request match for the description of the algorithm and the sake of brevity. The algorithm is illustrated in Figure 6.14.

Scan phase During the scan, each thread of a CTA is given a message, to which it holds on for the duration of the matching, and linearly walks through the receive request queue. In order to avoid synchronization between warps, a matrix is built up in shared memory. Threads within the same warp test whether their message matches with the receive request, at which all threads of the warp are looking at the same time. CUDA's *ballot* function (see Table **tab:gpu:warpvote**) is used to obtain a bit vector, wherein the number of the bit represents the particular thread within the warp and indicates whether the thread's message matches with the receive request or not. This bit vector is written to the matrix. The hierarchical approach, in which warp vote functions are used to determine the bit vector, reduces the amount of shared memory,

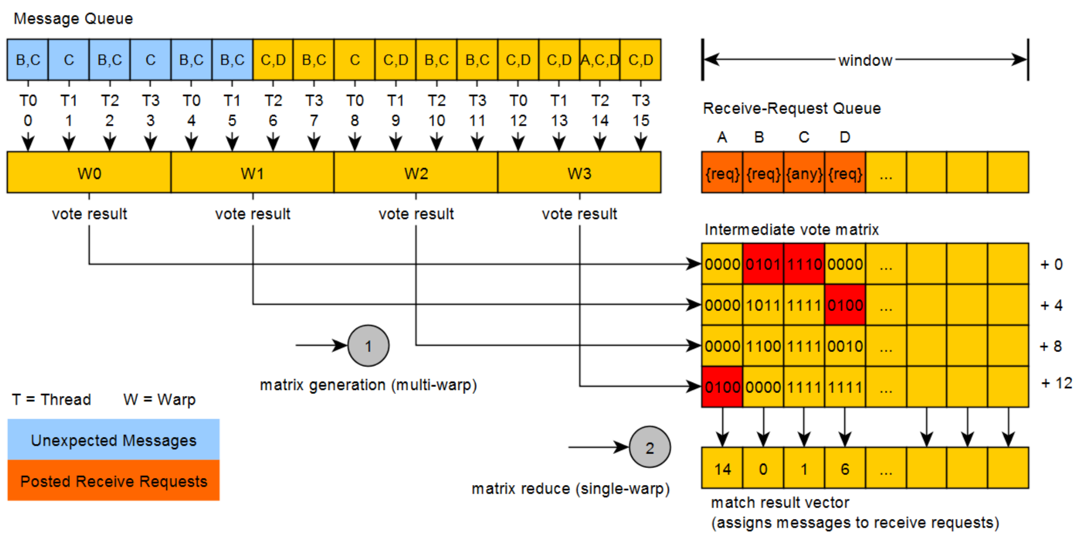


Fig. 6.14 Warp-parallel MPI compliant message matching algorithm [21]. The graph shows both phases, the scan (matrix generation) and reduce phase.

Algorithm 3 multi-warp scan [21]

```
1: SendObj = sendBuffer[ thread:id ]→getObj()
2: for i from 0 to window - 1 do
3:     RecvObj = recvBuffer[ i ]→getObj()
4:     int32 vote = __ballot( SendObj == RecvObj )
5:     voteMatrix [ warp:id * window + i ] = vote
6: end for
```

compared to each thread writing its match result to a exclusive position within the matrix. Consequently, the number of rows is determined by the number of warps performing the scan. Furthermore, the scan phase can be performed by all warps of the CTA in parallel and does not require synchronization. Algorithm 3 provides more detail on the implementation.

Reduce phase After the matrix is generated, it needs to be reduced to a single vector, assigning matching messages to receive requests. However, due to wildcards and ordering, dependencies exist between rows, but also between columns. Consequently, this phase does not allow for any parallelism and has to be executed sequentially. In particular, one warp is sufficient as the maximum matrix height amounts to 32 as this is also the maximum number of warps per CTA.

Algorithm 4 shows the reduce phase, executed by a single warp. First, each thread starts with a bit mask, in which all bits are set to '1', and loads one bit

Algorithm 4 Algorithm to reduce a column-vector [21], which contains the vote results, to a single match.

```
int32 mask = 0xFFFFFFFF
if thread:id < warps then
    for i from 0 to window - 1 do
        int32 vote = voteMatrix[thread:id * window + i]
        int32 bidders = ___ballot( vote & mask )
        if thread:id == ___ffs(bidders) -1 then
            int32 match = ___ffs( vote & mask ) - 1
            mask = mask & ~ (1<<match)
            result[ i ] = thread:id * warp:size + match
        end if
    end for
end if
```

vector from the same column. The mask is used to prevent already matched messages to be considered again for upcoming matches. Thus, each thread bitwise conjoins (binary AND) the mask and the bit vector. The result is used in the *ballot* intrinsic again to determine which threads see a matching message. In order to preserve ordering, the thread that holds the bit vector of the warp with the lowest ID wins and writes its match result to the match vector. Additionally, the position of the match is erased from the mask before the next column of the matrix is reduced.

Optimization The reduce phase is the main bottleneck of this algorithm, because of its sequential nature and execution by a single warp. However, it can be overlapped with the scan phase using a pipelined approach. Therefore, after a few columns are written by the scan phase, one warp can already start to reduce these columns.

As aforementioned, the algorithm technically comprises a third phase, in which matches are removed from both receive request and message queue. This step requires a prefix scan and all unmatched elements to be moved toward the head of the queue. If only a few matches could be found the compaction is not necessary and bubbles in the queue can be tolerated. However, the performance is reduced accordingly.

Performance The performance of the matching for three generations of GPUs, particularly Kepler (K80, single GPU), Maxwell (M40), and Pascal (GTX1080),

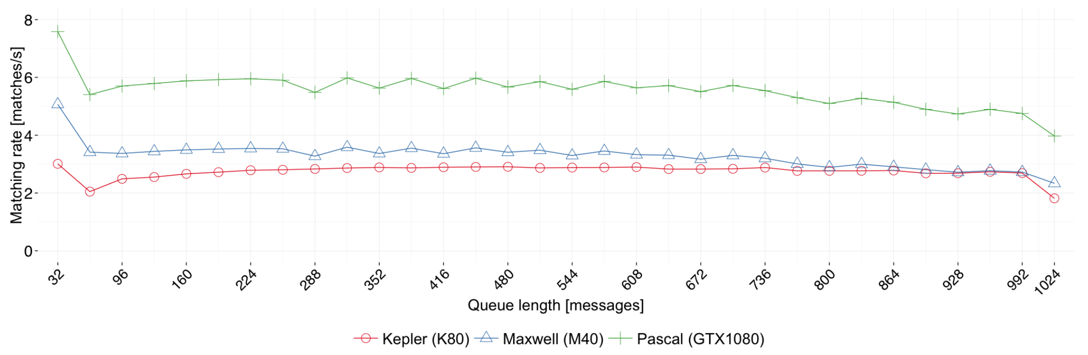


Fig. 6.15 Performance of the warp-parallel MPI compliant matching algorithm on a Kepler, Maxwell, and Pascal GPU [21]. The performance is shown for a varying queue size.

is depicted in Figure 6.15. For queue lengths up to 64 elements the matrix does not need to be generated, thus a single warp is sufficient to match both queues directly. As can be seen, the pipelined approach yields constant performance, independent on the queue length and provided all elements match. The drop for 1,024 elements per queue is due to all warps being required for the scan and hence the reduce cannot be overlapped anymore. The performance on the Pascal GPU is twice as high as Kepler. Since the algorithm has linear time complexity, the Pascal's two times higher clock rate accounts for the large difference in the matching rate.

While the matching rate in Chapter 4 is shown for best, average, and worst case, depending on the order of the elements in the receive request and message queue, the order does not matter with the GPU algorithm. However, this only applies if the queue is smaller or equal than 1,024 elements. Longer queues cannot be matched in one iteration by a single CTA and thus require more iterations or threads have to match more than one message at a time. However, the trace analysis in Chapter 4.3.3 also shows that queues range below 1,024 elements most of the time, rendering one CTA sufficient to perform the matching in one iteration.

6.3.2.3 Matching in Relaxed Message Passing Protocols

While the presented algorithm complies with MPI's protocol in that it supports tag/source wildcards and respects ordering, the performance on GPUs is limited by the lack of parallelism. Compared to the CPU's matching performance, shown in Chapter 4.3.2, the 'best case' matching rate is about four times higher than the GPU algorithm. However, the difference becomes smaller for the average case as the CPU's performance depends on the order of receive requests and messages.

Prohibiting wild cards While MPI has been designed and optimized for latency-optimized processors, a GPU message passing system has different requirements. For example, wildcards significantly limit parallelism, but can easily be given up as many of the exascale proxy applications do not even apply them. In fact, it would be sufficient to prohibit the source wildcard, allowing to partition the rank space and implement multiple queues. Each queue is matched individually, allowing for more parallelism and enabling higher matching rates.

The performance for such partitioned approach is depicted in Figure 6.16. The matching scales linearly for up to four queues, but then ranges below linear speedup due to additional overhead. Although both phases are pipelined, the scan and reduce still needs synchronization, which applies to all warps of a CTA, thus affecting the matching of all queues. Nonetheless, the performance can be increased up to about 60M matches/s on the Pascal GTX1080 for queues with 1,024 elements. Longer queues require additional CTAs, annotated with numbers in the graph, but overall performance drops. As the matching requires shared memory and register space and all CTAs are scheduled to the same SM, only two CTAs can be executed concurrently, thus the execution of more CTAs is serialized. Nonetheless, multiple CTAs still allow for much longer queues to be matched. If matching of long queues is important and the shown performance is insufficient, more SMs can be used, which will allow for linear speedup.

Although the matching is significantly improved by prohibiting source wildcards, the application's communication behavior poses limits on that approach. For example, the number of queues is limited by the peers a rank is communication with. If an application exchanges data with only eight other processes, no more than eight queues could be used. Also, if each queue does not contain at least 32 elements, the matching becomes inefficient as not all threads of the warp are utilized.

No unexpected messages Another performance-limiting aspect of the matching is the number of unexpected messages. A message is unexpected if no matching

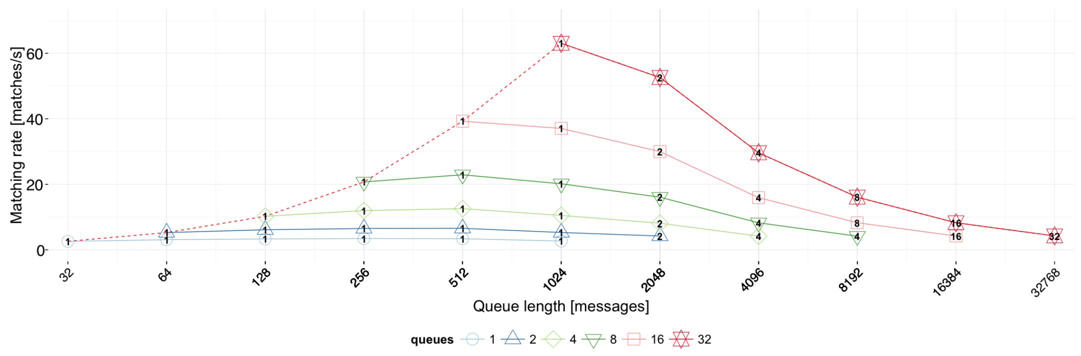


Fig. 6.16 Matching performance for multiple queues, enabled by the prohibition of wild cards [21]. The numbers inside the data points refer to the number of CTAs. Performance is shown for varying the number of queues over the total queue size. Measurements are performed on the Pascal GTX1080.

receive request is available at the time of the matching. Thus, the matching rate is reduced linearly. For example, if only half of the messages can be matched with receive requests the matching performance is also halved. Furthermore, compaction becomes necessary to remove matched messages and make space for other messages to be matched in the next round. If there were no unexpected messages, the compaction would not be required and it is sufficient to move head and tail pointers within the queues.

While both relaxations, namely prohibiting the source wildcard and unexpected messages, improve the matching rate significantly, the main limitation is still the order of messages that has to be preserved. Without ordering, the choice of the data structure that is used to perform the matching is no longer limited to queues and lists.

6.3.2.4 Hash-Table-based Matching

When ordering can be given up, hash tables seem promising as they allow for constant insert and search time. Similar to regular tables, the values can be stored and retrieved at arbitrary positions. Inserting a value v requires to compute the key k by using a hash function h. Given an universe of keys U and a hash table T which has space for m elements, one has [111]:

$$h : U \rightarrow \{0, ..., m-1\}$$

Instead of using a regular table with size $|U|$, the size of the hash table m is much smaller. However, this implies that not all values of the universe U can be stored in the hash table and multiple values map to the same key in that $h(v_i) \to k_i$ and $h(v_j) \to k_i$ with $i, j \in \mathbb{N}$. This is referred to as *collision*. There exist various strategies to resolve collisions, which are briefly described in the following [112].

Open Addressing If two values map to the same key the hash table is searched for an unoccupied bucket, in which the colliding value can be stored. This is commonly referred to as *probing*. Various probe sequences exist to find an empty bucket. *Linear probing*, for example, uses the hash function $h'(k,j) = (h(k)+j) \mod m$ with $j \in \mathbb{N}$ denoting the probing sequence number. Although computing the sequence is simple, it quickly leads to long clusters which have to be traversed before an empty bucket is found. Another probing approach uses the hash function $h'(k,j) = (h(k)+c_0 j + c_1 j^2) \mod m$ with constants $c_0 \leq 0, c_1 > 0$, which is known as *quadratic probing*. A third approach is to use a second hash function to try another bucket, with $h'(k,j) = h_j(k)$. Here, clustering is not an issue but a set of hash functions need to be found that resolve collisions efficiently.

Eviction Instead of taking the colliding value and finding an empty bucket, the value stored in the table is replaced by the colliding value and an empty bucket has to be found for the original value. A prominent example is *Robin Hood Hashing* [113], in which the probe sequence for each value is tracked and values are only evicted if their number of sequences is smaller, thus the value is younger. Overall this reduces the number of probes compared to open addressing. Another eviction-based strategy is *Cuckoo Hashing* [114], in which a set of hash functions is required to compute new keys for every iteration. However, there is no guarantee the eviction process terminates and it depends on the choice of hash functions that are used.

Closed Addressing Here, each bucket can hold multiple values. This is usually implemented as a dynamically allocated list. However, as GPUs do not provide sufficient support for dynamic memory management this strategy is not applicable to the message matching on GPUs.

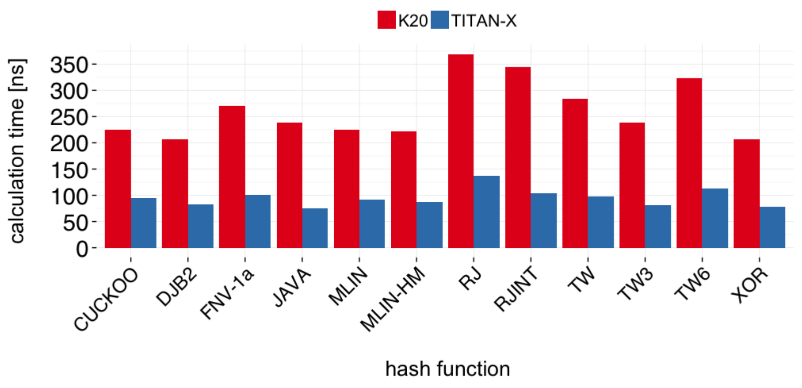


Fig. 6.17 Execution time for various hash functions on the Kepler K20 and Pascal TITAN X [112].

In the following, various approaches are implemented and evaluated regarding their applicability to the message matching on GPUs. Generally, hash functions are assessed based on their avalanche behavior, key distribution, and time to calculate the hash function. The avalanche behavior is the probability of the key's j^{th} bit flipping when the value's i^{th} bit is flipped. A good hash function has a probability of 0.5 for every i and j.

The execution time for various hash functions is depicted in Figure 6.17. The hash function was computed on the Kepler K20 and the Pascal Titan X GPU. The best performance can be observed for the *JAVA*, *XOR*, and *TW3* (Thomas Wang) [115] hash function. However, except for *RJ* (Robert Jenkins) [116], the performance difference is not significant. The same applies to avalanche behavior and collision rate. Thus, the *XOR* hash function is chosen for the following implementations.

The performance of hash tables does not only depend on the hash function and collision strategy, but strongly depends on the input data and its distribution. For example, if the input data comprises unique values only a suitable hash function can avoid collisions. On the other hand, if many redundant elements are present in the input data set the performance suffers from many collisions, independent of the hash function. Consequently, an understanding of the input data is inevitable for the design of performant hashing approaches. In context of message matching, wildcards or receive request that match multiple messages increase collisions and limit the hash table's performance.

Figure 6.2 shows the matching rate for various approaches, which are briefly introduced in Table 6.2. The x-axis shows the multiplicity, thus the percentage of

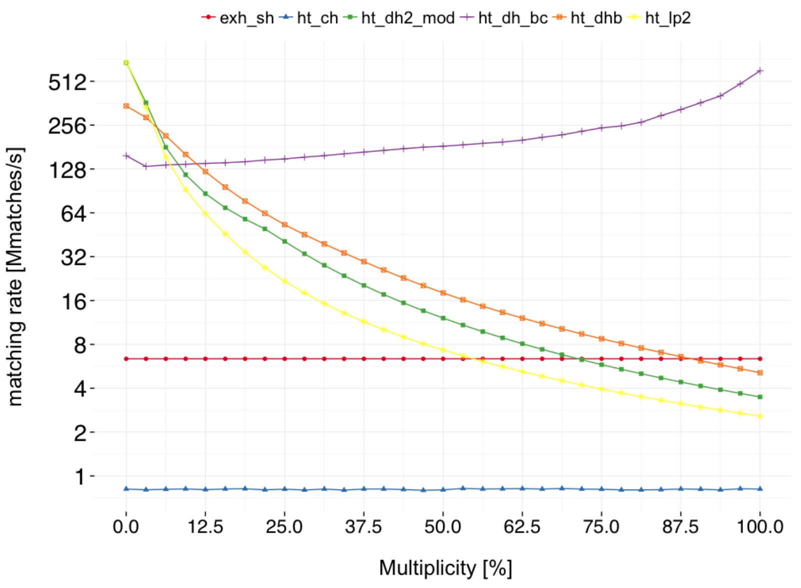


Fig. 6.18 Matching rate for various hashing approaches over the multiplicity of the input data set [112]. Results relate to the Pascal TITAN X.

identical elements in the input data set. For example, 100% means all elements are identical, while with 50% half of the values are unique and the other half is identical.

As shown, there is no approach that consistently performs best for all degrees of multiplicity. Instead, the performance can be divided into three regimes, of which the first two are rather small. The first regime ranges up to 6% and the highest matching rate is achieved by linear probing and double hashing, thus open addressing approaches. Next, between 6 and 12%, the chained batch approach yields the highest performance. After that, the chained batch approach with preprocessing is superior to all other approaches. In fact, the performance slightly increases with higher multiplicity while other approaches' performances decrease. This is mostly due to the preprocessing which eventually reduces the number of elements that are inserted into the hash tables for growing multiplicity. Also, the preprocessing requires less iterations for less unique values, thus overall performance benefits from redundant input data.

Using hash tables and suitable approaches, a matching rate of 128M to 512M can be achieved, which is significantly higher than the queue-based approaches in previous sections. However, the performance also depends on the length of the queue. The best performance is achieved for queues up to 1,024 elements. Larger queues as well as shorter queues yield lower performance.

Table 6.2 Brief description of various hashing approaches. More details can be found in Kühlwein's work [112].

Abbreviation	Description
exh_sh	Reference approach in which the message queue is sequentially searched for each receive request.
ht_ch	Cuckoo hashing as provided in the CUDPP library [117].
ht_lp2	Hashing with linear probing to resolve collisions.
ht_dh2_mod	Instead of linear probing, double hashing is used to resolve collisions.
ht_dhb	Chained approach in which identical are stacked, similar to closed addressing.
ht_dh_bc	Chained approach in which a preprocessing step links identical values before they are inserted into the hash table.

The trace analysis of various exascale applications shows that queues barely exceed 512 elements, thus hash tables yield between 200M and 500M matches/s, depending on the approach that is used. Regarding multiplicity, it can be said that multiplicity varies among applications. Most applications show a UMQ multiplicity of 10 to 30% [112] and it seems suitable to implement hash tables for the message matching as hashing still performs better than queue-based approaches under these constraints.

6.3.2.5 Related Work

Although there is no existing work for the message matching on the GPU, various optimizations have been explored for the CPU. Zounmevo et al. [118] propose a new matching algorithm that aims to reduce the memory footprint while it also improves scalability. Multiple queues and sequence numbers are used to partition among the rank space and support wildcards. The reported performance indicates significant performance improvements, but no absolute numbers are provided.

A hash table approach is proposed by Flajslik et al. [119]. Marker entries are inserted to preserve ordering information, which is necessary to support wildcards. A fire dynamics simulation is studied with the new matching approach and the performance is about 3.5 times higher than with the standard MPI matching. Note, the matching alone accounts for these gains, emphasizing the importance of this particular problem.

Bayatpour et al. [120] suggest a dynamic bin-based approach with multiple lists and rank partitioning, which performs two times better than the default matching.

Message matching without wildcards has been analyzed by Dang et al. [121]. Their proposal comprises a hash table based design and aims to support a large number of threads on many core processors, such as Intel's Xeon Phi.

6.3.3 Active Messages

Along with transferring data and serving synchronization purposes, messages can also carry work to other processors. *Active Messages*, as this form of messaging is called, is a fundamental building block for task-based programming models. They generally allow to exploit data locality by moving computation to the data, as opposed to bringing data closer to computation.

6.3.3.1 Related Work

An active message framework tailored for heterogeneous systems has been proposed by LeBeane et al. [122], focusing on AMD's Heterogeneous System Architecture (HSA). Their approach is based on RDMA transfers and a shared command queue between CPU and GPU. When messages are received a kernel lookup table is searched for a match. The matching entry points to a kernel, which is launched with arguments carried by the received message. The authors evaluate their approach with simulating an Accelerated Processing Unit (APU) that implements CPU, GPU, and NIC. A performance improvement of 10-15% is reported for an MPI reduce operation on two nodes. A speedup of about 1.25x is reported for the allreduce operation during the training phase of a neural net using Microsoft Cognitive Toolkit (CNTK). The main improvement comes from bypassing the CPU when work is launched remotely on GPUs.

6.3.3.2 Active Messages Using SoftNIC Queuing

A similar approach can be implemented with the queues presented earlier in this chapter, which are used to exchange messages between GPUs. The SoftNIC process, either as separate kernel or integrated into the application's CTAs, has to execute preregistered kernels upon reception of messages. Similar to GASNet, code can not only be provided for the reception of a message, but also for the response the SoftNIC generates.

In the first step, kernels are registered and given an identifier. This has to be executed on all GPUs accordingly. When a message is received the lookup table is probed to obtain the correct handler, which is invoked with the message as argument. A crucial aspect is the entity that executes the handler. When the message is received by a single warp, the handler can be executed by this particular warp. However, the number of threads is limited to the warp size which currently amounts to 32 threads. Another possibility is to launch a separate kernel with as many threads as needed, however, the current GPU's scheduling cannot guarantee that the kernel can be executed along with the application that is currently running. A third option is to use all SoftNIC resources, for example an entire CTA, but that requires coordination and synchronization and stalls communication for the duration of the handler execution. Nonetheless, these are architectural limitations and the following presents results from a warp-based active messaging while one has to be aware of the consequences and opportunities that are provided by an improved architecture.

API The active messaging API consists of a base handler class providing virtual functions. A new handler derives from that class and implements the functions according to the operations that need to be executed when the message is received or the response is generated. The following code excerpt shows how an active message handler can be implemented for the *Random Access* benchmark, which Chapter 5 describes in more detail. The handler `AMUpdate` derives from class `AMBase` and overrides the `run` function. Here, the message is passed as an argument and contains the index for the table update. The update is then performed by an atomic *xor* operation.

Listing 6.1 Example code for an active message handler [123].

```
class AMUpdate : public Mantaro::AMBase
{
  /* ... */
   __device__ Mantaro::Error_t run(AMMsgData& data)
   {
    /* ... */
    uint64_t addr = data.op ;
    uint64_t idx = get_local_idx(addr);
    /* ... */
    atomicXor( mem_array[idx], (unsigned long long) addr);
    /* ... */
   }
   /* ... */
}
```

The handler is registered on the CPU and is passed along with the control flow to the GPU. Here, messages can be sent and received using queues. When a valid message is received by a warp it is executed as shown in the following example. The handler's `warp_exec` function is a wrapper for the handler's `run` function. As can be seen, the simplicity of active messages allows to easily move work to the location of the data on which the instructions are to be performed.

Listing 6.2 Example code for receiving and executing an active message [123].

```
/* receive message */
err = handler -> deq_recv (msg , gpu_id_1);
if ( err != Mantaro :: Error_t :: SUCCESS )
  msg = MSG_EMPTY ;

/* ... */

/* execute message */
err = handler -> warp_exec ( msg );
```

Performance The results of the benchmark are shown in Figure 6.19. As opposed to the experiments in Chapter 5, GPUs (Kepler K80) are deployed in a single node and connected by PCIe, hence exchanging data at a bandwidth that

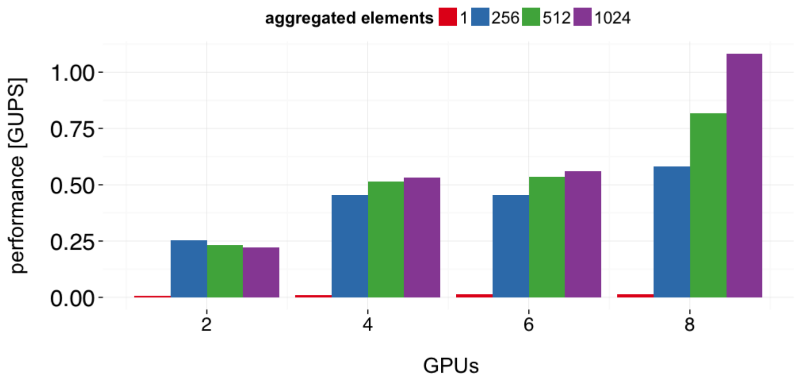


Fig. 6.19 Performance for the *Random Access* benchmark for various bucket sizes [123]. The results relate to the Kepler K20.

is effectively about 10x higher than provided by EXTOLL's FPGA-based NIC. Nonetheless, an update rate of more than 1 GUPS is achieved with 8 GPUs and 1,024 updates aggregated into one message, which is 10x more than the similar benchmark achieved with EXTOLL.

The complexity of implementing more benchmarks prohibits broader analyses, however, it shows that such an abstract interface can yield comparable performance. Nonetheless, there are several limitations that need to be overcome to support active messages more efficiently, first and foremost better queuing and more control over scheduling. The concept is still valuable, especially in regard to task-based programming models.

6.3.4 Memory Management

The ability to dynamically allocate and deallocate memory is another important building block for managing communication. As for now, memory management is handled by the CPU and allocated memory resources are passed to the kernel. This is static and does not allow for changes unless control is returned to the CPU. In order to implement a communication management layer memory allocations need to be adaptive in case for large messages waiting to be received or if the management requires more memory for administrative purposes. Many publications exist in this area and are briefly introduced in the following, but it is referred to the publications themselves for more details.

One of the first to implement a dynamic memory allocator for GPUs were Huang et al. [124]. Their *xMalloc* allocator uses lock-free FIFO data structures residing in global memory with headers being small enough to be modified by a single atomic operation. Their experiments show that the allocation latency is reduced by a factor of 211 compared to *cudaMalloc* (0.166ms versus 0.05ms measured with xMalloc) while it also scales well with the number of threads.

ScatterAlloc [125] is another allocator, proposed by Steinberger et al. It uses fixed size memory regions and concurrent requests are scattered across these regions as opposed to linearly searching through a list, but their approach eventually leads to fragmentation. The largest entity is a "super block", which itself is divided into pages. Once a memory request is received, the page is split into equally sized parts and the whole page can only be freed when all parts are free. The authors state that their approach is 10x faster than the aforementioned *xMalloc* allocator by Huang et al. [124] at full GPU utilization.

Widmer et al. [126] optimize *xMalloc* and *ScatterAlloc* by sharing super blocks among threads of a CTA. Their approach is called *FDGMalloc*. A single thread is assigned to handle all allocations as opposed to having all threads allocating memory concurrently. The authors report speedups in the range of 10 to 100 over *ScatterAlloc*, depending on the number of threads that are issuing requests.

Another approach, referred to as *Halloc*, implements a concurrent slab allocator [127], in which hashing is used to find free blocks. The evaluation is again compared to *ScatterAlloc* and shows significant improvement of 4x for up to 256K threads and 1,000x for more than 576K threads. The benchmark consists of a kernel invoking a large number of allocations and deallocations.

Vinkler et al. [128] not only propose two additional allocators, namely *Atomic Wrap Malloc (AWMalloc)* and *Circular Malloc (CMalloc)*, but also compare to the previously mentioned allocators and provide recommendations of when certain allocators should be used. Both introduced allocators use a circular memory pool, but *CMalloc* uses a list for organization. With *AWMalloc*, memory cannot be freed and it only works for a large pool and allocations should be used only for a small period of time. Consequently, *CMalloc* adds a header to allocated chunks to support deallocations. Similar to *ScatterAlloc*, chunks are split, but not uniformly and more adaptive to the application. The authors recommend to use *Halloc* if there are many threads allocating memory, but to use *FDGMalloc* if each thread issues a large number of allocations. On the other hand, their own *CMalloc* is meant to be used if requirements are unknown.

With respect to the SoftNIC, it seems that *Halloc* provides the best performance, although *CMalloc* yields similar results. Both approaches are made publicly available.

6.4 Work Completion

Work completion is the third and last important aspect of the SoftNIC. It comprises techniques to notify the application about status of requests and arrival of new data or messages. Nonetheless, it is also challenging as current GPUs do not provide any opportunity to control scheduling or even allow CTAs to be preempted from execution.

6.4.1 Scheduling

One of the main differences between a CPU and GPU is the scheduling. While threads can be put to sleep and preempted on the CPU, the GPU does not provide such mechanisms as scheduled CTAs complete before the SM can execute another CTA. Consequently, interactions between kernels or CTAs are prone to deadlock. However, in order to work efficiently the SoftNIC requires the following guarantees to be given by the system:

- It has to be ensured that the SoftNIC is executed, for example by dedicating one or a few SMs entirely to execute communication tasks. The current scheduling cannot guarantee that kernels are scheduled together and allows for kernels to be executed sequentially.
- Communication demands can vary during the application's run time. If demands are increasing the SoftNIC should be able to allocate more resources, for example occupy more SMs as originally anticipated. Hence, application CTAs should be evicted and scheduled at a later time in order to make room for the SoftNIC. This is especially important for active messages as incoming messages may trigger a kernel that needs to be executed in a timely manner.
- CTAs that are waiting for messages should be preempted and brought back to execution when the appropriate message arrives. The same applies to CTAs waiting for synchronizing operations to complete.

Although CTA preemption would solve most issues it would also increase latency as the CTA's context need to be saved and restored. A CTA could use up to 256KB of register space and 64KB shared memory, thus a maximum of 320KB context needs to be saved and restored. This amounts to almost $1\mu s$ at 732GB/s memory bandwidth. Although the context only needs to be saved when the execution is preempted the additional latency hurts performance significantly. Consequently, more efficient ways are required to ensure high performance and scalability.

Related work The scheduling of real time applications is discussed by Tanasic et al. [129]. In order to support multiple contexts executed by the SM, the execution unit is extended by a context table. Each context, or process, is given its own virtual address space, similar to virtual addressing in CPUs. The context table contains the base page table register that is probed in case of a TLB miss, as opposed to the baseline GPU architecture that uses the same page table for all SMs and kernels. When multiple kernels are scheduled for execution and none of the SMs is idle an SM is preempted. The authors present two preemption strategies, namely context switching and SM draining. The first follows the traditional context save/restore model, while the latter finishes the execution of a CTA before other CTAs from a different kernel are brought to execution. Regarding the turnaround time, the context switch approach is significantly better than SM draining for a large variety of concurrently scheduled applications. Furthermore, SM draining would not help in the SoftNIC application as CTAs might be waiting on messages and would not complete until the message arrives. Nonetheless, the work shows that context switching is feasible at reasonable overhead.

Similar approaches are analyzed by Jason J. K. Park [130], in which three preemption strategies are presented: switching, draining, and flushing. While switching and draining are the same approaches that are analyzed by Tanasic et al., the flushing strategy simply discards the currently running CTA and starts over at a later time. Theoretically, this approach also works with the SoftNIC with the assumption that already exchanged messages are ignored in the repetitive execution. However, the authors also state that the right preemption strategy depends on the application and it cannot generally be said that one approach is superior than the others. Their proposed framework uses all three strategies and applies the one that suits the application best.

Zhen Lin et al. [131] focus on the context switching. Using live variable analysis, which allows to determine variables that are potentially read before their next write at any point within the program, the register states can be reduced significantly. Since keeping track of the variable liveness would be expensive for every point of time, preemption is restricted to certain points within the program. Furthermore, register states are compressed by detecting patterns and only saving meta data that is necessary to restore the original pattern. For example, if threads access an array with their thread ID the switching unit only needs to save the start address and the number of threads. They report that using this approach the context can be reduced to about 10% of its original size. Overall, the presented results are promising and show that context switching might indeed be a reasonable strategy for preemption. The fact that CTAs could only be preempted at certain points is inline with the SoftNIC concept as preemption is only needed when the CTA waits for messages or synchronization.

In addition, Jin Wang et al. [132] propose *LaPerm*, a scheduler designed for dynamic parallelism. It assumes that parent and child kernels share temporal and spatial memory locality, of which the scheduler is aware. The results show an improvement of about 27% over round-robin scheduling.

Chen et al. [133] present *EffiSha*, which comprises a framework for software preemption. Through source-to-source translation, kernels are transformed to a persistent thread approach, in which eviction of CTAs is only allowed at the end. This is similar to the draining approach, but does not require any additional hardware. Nonetheless, it is inapplicable to the SoftNIC concept.

Outlook With more and more applications demanding more control over scheduling future GPUs are expected to add additional features accordingly. The latest Pascal architecture already added support for instruction-level preemption, in which it prioritizes graphics applications and can preempt running computational kernels [10]. Although only little information is available, it seems the context is saved to off-chip memory when a kernel is preempted. However, it has to be stated that without having control over the scheduling the SoftNIC concept is hard to realize as messages cannot be exchanged without being prone to deadlocks. Especially the approach presented by Lin et al. [131] seems promising and describes a viable and promising solution.

6.4.2 Events and Notifications

Independent of scheduling, events and notifications are important as communication path from the SoftNIC back to the application. The notation here regards notifications as notifiers that are simply discarded when they are consumed, while events can be used in more complex tasks, for example to trigger other operations or bringing back preempted CTAs. A good comparison for notifications can be found in Infiniband (e.g. notification in the completion queue) and EXTOLL, while events rather follow the concept presented in the active message framework *Realm* [40].

Software approach The simplest approach would have each CTA to maintain its own queue in which events are placed. However, it not only requires a significant amount of memory but also pollutes the memory system with requests when CTAs are polling on their queues. Hence, a shared data structure is required and queues seem unsuitable as they enforce elements to be consumed in order. Hash tables, on the other hand, seem to be the natural choice as they allow for constant insert and retrieval time.

Using hash tables to store events requires both producer and consumer to use the same key to enter the table, for example the destination CTA and GPU ID. Another approach is to use a stack to hold available keys and when CTAs submit a work request they first obtain a new key and include it in the work request. This way the SoftNIC knows where the CTA expects the event to appear and after the request is matched with the appropriate message the event is written to the event table. This also allows to aggregate events as multiple requests can use the same key and the SoftNIC increments a counter. For example, if two requests are submitted with the same key the requesting CTA waits until the table entry counter equals two.

The performance is mainly limited by concurrently accessing the event key stack, which needs to be atomic and can lead to serialization. Furthermore, CTAs still need to poll on the table to consume events, resulting in significant traffic within the memory subsystem. Furthermore, the SoftNIC would need to keep state information, which requires additional memory overhead.

Hardware approach Similar to the queuing problem, the polling on the event table needs to be kept local in the SM. In fact, the queue controller server/client approach can be extended by the support of events [109], depicted in Figure 6.9. A CTA can register events locally at the client, which extends the SM. Once registered, the client notifies the server and waits for a response. When the event is triggered by another CTA the server broadcasts the event and all clients waiting for the event to be triggered can update the local table.

Simulation shows that the number of global memory accesses can be reduced significantly [109]. The software approach performs similar for less than 4,000 CTAs as events can be kept in the L2 cache. However, this is measured with an isolated benchmark and the L2 cache should rather be used for the application's working set.

Outlook Generally it can be said that events become powerful if the GPU supports the preemption of CTAs. In that case, a CTA can be preempted if it waits for an event and as soon as an event is triggered the CTA becomes eligible for execution again. The hardware approach simply extends the queue controller approach and reduces the memory traffic significantly. Although the SoftNIC benefits significantly from extended support for events and notifications, it remains to be shown whether other applications benefit as well and thus rendering a stronger case for vendors to adapt the hardware accordingly. With signs of preemption becoming available in future GPU generations, the software approach for events may be sufficient for the SoftNIC concept.

6.5 SoftNIC Architecture Discussion

This chapter has been introducing and discussing various building blocks individually. Here, the overall SoftNIC architecture is discussed, composed of previously introduced building blocks.

6.5.1 Architecture

An important aspect of the SoftNIC is modularity and flexibility, which are the main advantages software has over hardware. Depending on the communication model, various SoftNIC incarnations can be compiled, only implementing features that are needed. For example, a PGAS model might only rely on the SoftNIC for memory registrations and put/get operations, but mainly uses load/store

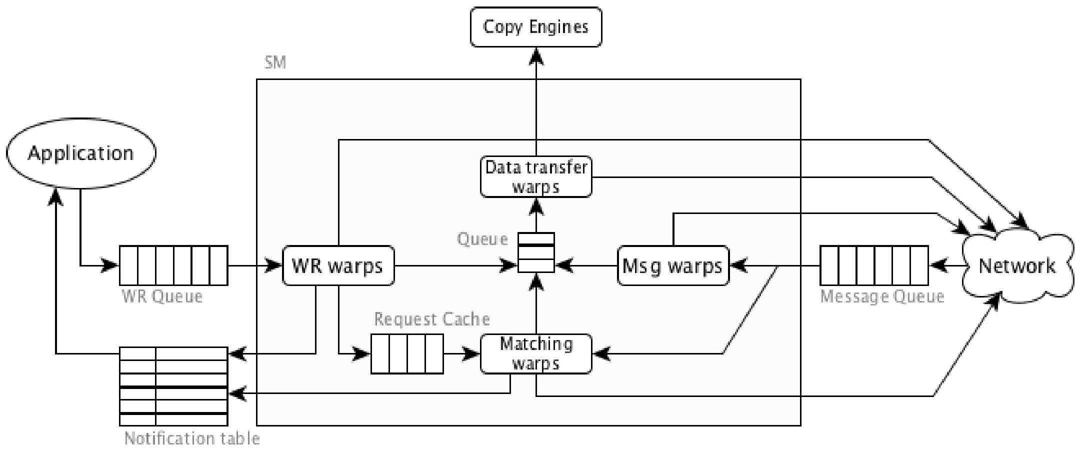


Fig. 6.20 A possible SoftNIC architecture with static warp-specialization.

operations for communication. On the other hand, a message passing model requires much more management, hence strongly relying on SoftNIC features.

Figure 6.20 shows an example of a possible SoftNIC architecture. The basis of the SoftNIC is warp specialization, in which warps are dedicated to certain tasks and independent of each other. For example, some warps take elements from the work request queue and place them in a cache structure, which is necessary for the matching with messages, performed by other warps. When a warp fetches one or multiple work requests it processes it completely before a new one is fetched again.

A different approach is depicted in Figure 6.21. Although warp specialization is still applied, warps' tasks are much more flexible and dynamic. A supervisor warp decides which tasks currently demand the most resources and thus assigns worker warps to particular tasks. While some warps are constantly engaged with a certain task, free workers are pooled into a worker pool. Workers poll on an internal work distribution structure to fetch small tasks, which they execute entirely or partially until they generate new tasks that are shared through the distribution structure.

Although the supervised approach seems superior due its flexibility, the internal distribution structure quickly becomes a bottleneck. Warps are constantly and concurrently accessing the structure, requiring atomic operations which, due to contention, are likely being serialized. Furthermore, shared memory is scarce, significantly limiting the number of pending tasks. Shared memory is also required to support the matching and cache receive requests.

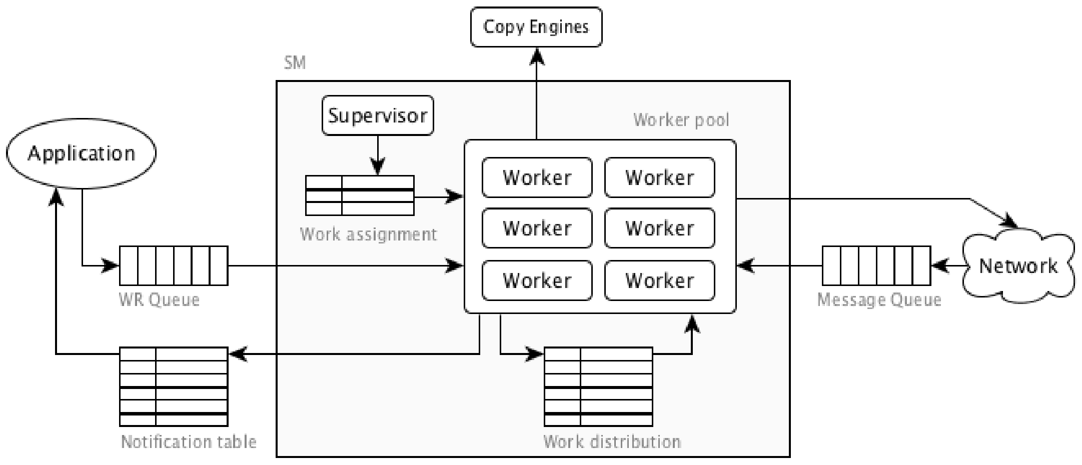


Fig. 6.21 Another SoftNIC architecture that allows for dynamic task assignment.

6.5.2 Differences to Software-centric Networking on CPUs

The main difference is the execution model. CPU threads have their own context, including registers and program counter. On GPUs, threads are tied together to warps and share a register file in which data resides even if threads are currently stalled. A single program counter is also shared among threads of the same warp. Consequently, tasks need to be executed by 32 threads in order to yield high efficiency and performance. Since many tasks are hardly parallelizable, each thread is more likely to execute a different request. However, this is only efficient if requests are processed by the same instructions without divergence. This could be the case if threads process the same kinds of work requests, but may be impossible if threads work on different requests. Nonetheless, the parallel SoftNIC approach certainly optimizes throughput rather than latency, which is also the GPU's foundation.

Another difference is that the CPU has more control over the system through the OS. For example, more memory can easily be allocated through system calls and allocations can be pinned to prohibit pages to be swapped to disk. GPUs cannot rely on the OS and depend on the CPU to prepare everything before a kernel is executed, including memory allocations and I/O mappings. This certainly reduces flexibility and limits the GPU's ability to perform networking tasks.

6.5.3 Verdict

The chapter has shown that it is possible to perform networking tasks on the GPU on top of a large shared memory system, including inherently sequential tasks like message and receive request matching. Various building blocks have been presented and evaluated in regard to their compatibility with the GPU's execution model and architecture.

One of the most important building blocks is the queue as it implements the interface between application and SoftNIC. While queues allow for a reasonable throughput to be yielded on GPUs, the benefit of extra queuing hardware is significant and would improve throughput much more. In software, the latency to enqueue elements is quite high and requires many memory accesses and arithmetic instructions, distracting threads from working on the application.

The second interface comprises notifications, providing a path from the SoftNIC to the application. Although this can be implemented reasonably well in software, the mechanism is significantly limited by the GPU's scheduling. The only way for the application to receive notifications from the SoftNIC is polling, which not only generates many memory transactions but also blocks resources. Because this can easily result in a deadlock situation, a persistent thread programming model is inevitable with the current scheduling approach.

Summarizing it can be said that the SoftNIC can provide abstractions to hide complexity of communication in larger scale systems, aiming to improve programmability rather than performance. Furthermore, a persistent thread model is the only feasible approach at the moment. Nonetheless, it can be observed that GPUs are becoming richer in their ability to perform general purpose tasks and more and more features are added to make the GPU a first class processor. Along that road, the SoftNIC becomes an attractive approach to allow applications to scale out to GPU-centric systems.

CHAPTER 7

Discussion

In this chapter, insights from previous chapters are reviewed and further discussed. Furthermore, current trends are taken into account to evaluate results and the impact of this work.

7.1 Related Work

Several related publications have been presented throughout this work so far and this section reviews directly related work and how they distinguish from this work.

One of the first to propose GPU-sided sourcing of communication was Stuart et al. with their DCGN framework [12]. The proposed message passing scheme dedicates a CPU thread that receives work from the GPU to execute communication. Although this marks a first step, the GPU still relies on the CPU which is different from what this work is proposing. Furthermore, only message passing was considered as communication model, while this work also looks at flat and partitioned shared address spaces and one-sided communication.

Similar CPU-dependent approaches are dCUDA [13] and GPUnet [14]. The first approach implements one-sided communication with help from MPI running on the CPU. GPUnet, on the other hand, implements a networking layer on the GPU based on TCP/IP sockets. The CPU is required to assist with notifications. Again, as distinguished from this work these approaches still rely on the CPU and focus on a single communication model.

The closest related work has been published by Lena Oden [15], [85], [90]. Oden proposed GGAS and Infiniband VERBS for GPUs and studied how the GPU can directly interact with the NIC. GGAS is used in this work and compared to other GPU-centric communication models. Oden's work also focuses on unmanaged communication.

This work is the first to investigate communication management on GPUs, especially regarding the matching of messages and receive requests. Various applications have been studied in regard to their MPI characteristics and queue lengths [51]–[55], [57], [134], but this work is the first to consider exascale proxy applications.

7.2 Application Analysis

Chapter 4 analyzed various exascale applications in regard to their communication characteristics, focusing on message passing and MPI. This analysis is important to understand communication in HPC environments and allows to optimize and design the communication architectures of HPC systems.

7.2.1 Communication and Synchronization

It is shown that an average of 35% of the application's execution time is spent within the MPI library on communication and synchronization tasks, clearly emphasizing the importance of communication. The MPI time can be further broken down into particular MPI operation, which shows that synchronization is the main bottleneck, especially at larger scale. While most data is transferred through point-to-point operations, the most time is spent in collective operations due to their implicit synchronization. Non-blocking collectives as proposed by Torsten Hoefler [58] have been been observed in the traces.

Looking at exascale applications and systems, it is advisable to use non-blocking operations wherever it is possible, even for collectives. Waiting on a large number of processes to reach a certain point within the application poses a tremendous bottleneck. In fact, it might be better to rethink the traditional message passing model and rather think in tasks that can be executed independently. In case some processes are still waiting for other processes to finish a collaborative task, smaller tasks can be executed in the meantime. At larger scale, things like reliability and fault tolerance also have to be addressed, which seem to be more intuitive in task-based models.

One of the most rapidly emerging applications is the training of deep neural nets. Here, synchronization is most important for scalability. While today's applications mostly rely on asynchronous data parallelism, model parallelism and synchronous data parallelism might be dominating in the near future. This is mainly due to two aspects. First, asynchronous data parallelism can hurt the convergence of the model significantly as the batch size indirectly increases and weight updates might get lost. This is know as the stale gradient problem and described in more detail by Suyog Gupta et al.[65]. Second, with models becoming larger and larger the need for model parallelism increases. However, model parallelism requires synchronous collective communication, which needs to be supported and optimized by the the system. Here, GPU-sided communication becomes increasingly important, which renders this work valuable for the design of tailored communication architectures.

Nonetheless, it was shown that model parallelism shows great potential for scalability if GPUs are equipped with a fast network access.

7.2.2 Messaging at Large Scale

The way an application sends messages can be described by how many messages are exchanged, the message rate, and the message size. Here, it must be distinguished between weak and strong scaling. Generally, strong scaling applications has shown that messages tend to become smaller as problems are decomposed into smaller tasks. Thus, the communication architecture should focus on the exchange of small messages with an emphasize on the matching of messages and receive requests.

The impact on weak scaling depends on the application and its computational complexity. The studied applications show that messages become larger with an increased problem size and a constant number of processes. The number of messages, however, seems to be more or less unchanged.

Furthermore, point-to-point communication is rather local as ranks only communicate with a small subset of other ranks, rendering the mapping and topology important. Another important aspect are collective operations, which are frequently used in scientific applications and exclusively used in the parallel training of deep neural nets. It is crucial for future systems to focus on collective operations in regard to system design and optimization.

MPI's messaging also needs to become more GPU-aware to further accommodate for a steady increase in heterogeneity. While GPUDirectAsync is a step in the right direction, it still requires the CPU to prepare a work plan and to submit all tasks to streams. In the future, GPU's should be capable of triggering communication within kernels, for example by declaring certain stream tasks as ready. These are then executed by the GPU.

Important issues to be addressed at larger scale also include reliability, quality of service, and process mapping.

7.2.3 Message Matching

With messages becoming smaller at larger scale, the matching becomes key to low latency. However, the current MPI matching protocol is too complex and need to be relaxed in order to allow for the best performance. For example, almost none of the applications use wildcards. It would be a simple change to MPI to allow communicators without wildcards to enable parallel matching algorithms.

More drastically, MPI could introduce something like fences that reestablish ordering. The following example shows how a region could be defined in which messages are matched out-of-order. Here, the user uses source and tag to identify each message uniquely. The start of the region instructs MPI to replace its lists with hash tables for the matching of messages. At the end, the region is closed and lists are reestablished. Alternatively, the system could allow users to create communicators in which ordering is not guaranteed.

Listing 7.1 Example for unorderered transfer regions in MPI.

```
MPI_Unordered_region_open( MPI_COMM_NO_WC );

for( src = 0 ; src < number_ranks ; ++src )
{
  if( src == my_rank )
    continue ;

  for( tag = 0 ; tag < number_messages ; ++tag )
    MPI_Irecv( <buffer>, <count>, <type>, \
      src, tag, MPI_COMM_NO_WC, <request> ) ;

}

MPI_Unordered_region_close( MPI_COMM_NO_WC );
```

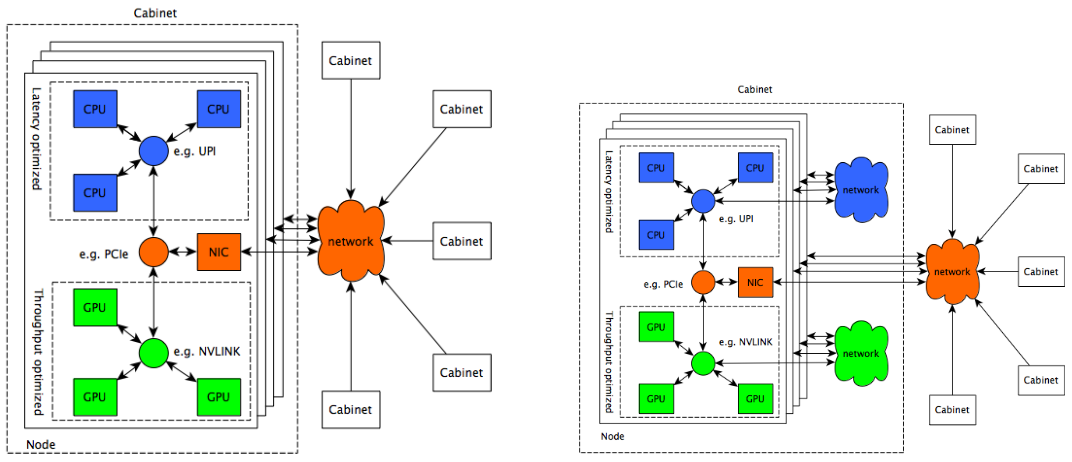
7.3 GPUs in Control of the Network

The importance of GPUs in scientific computing has been increasing in the past years, but due to their recent success in deep learning, GPUs have become more important in other domains as well. Undoubtedly, many distributed systems are going to be enhanced by GPUs and it can also be expected that the number of GPUs per CPU is increasing. Examples are NVIDIA's *Saturn V*[1] and the *Summit* system at the Oakridge National Laboratory [135]. That being said, the need for GPUs to orchestrate communication has never been higher.

7.3.1 System Architecture

Scientific computing and artificial intelligence suggest that the number of systems with specialized accelerators is increasing, mainly driven by power constraints. The current model, in which the CPU orchestrates everything and certain compute-intensive parts are offloaded to another device, seems unpromising as it requires additional context switches and data copies. Instead, accelerators

[1]https://blogs.nvidia.com/blog/2016/11/14/dgx-saturnv/, last visited on June 9, 2017.



(a) single inter-node network (b) hierarchical network architecture

Fig. 7.1 Two possible high-performance and heterogeneous system architectures.

are becoming first class processors that are part of the network, receiving and executing tasks and communicating data with others. A good example is Intel's Xeon Phi processor which can be used as a processor within a SMP system, but also NVIDIA's efforts to make GPUs more general purpose. An example system, which is similar to NVIDIA's Saturn V supercomputer, is sketched in Figure 7.1a. The system consists of two domains, the CPU SMP optimized for latency, and a group of GPUs optimized for throughput. Both CPU and GPU processors are interconnected by their own networks and share a NIC for inter-node communication. There are also systems in which CPU and GPU share the same network, namely NVLINK, such as Oakridge's Summit with IBM PowerPCs and NVIDIA's Volta GPUs. However, this still requires PCIe to access the NIC.

Figure 7.1b shows a different system, in which another hierarchy is introduced. GPUs within a node are also connected to other GPUs of other nodes within the same cabinet by a fast and throughput oriented network, for example NVLINK. The same applies to the SMP processor. Each cabinet is also connected to a global network comprising all of the system's cabinets using a network that is shared between CPUs and GPUs, for example Infiniband.

What is crucial about this system is its specialization, not only in computation but also in communication. That means that CPUs and GPUs can implement different communication models, tailored to their execution model. However, there might also be interactions between the two types of processors, for example in a task-based programming model in which tasks can be scheduled to the processor that suits best the task's requirements. Here, dependencies and events might require communication between a CPU and GPU. Nonetheless, programming these kind of systems at larger scale is going to be challenging.

An open research question is also how much of an application can actually be performed on the GPU. If the critical path of an application can be entirely run on the GPU, a simpler and more power efficient CPU might be sufficient. In an extreme case, the CPU could also be seen as a co-processor for the GPU which offloads sequential tasks.

NIC architecture Regarding the NIC architecture, three aspects have to be considered. First, GPUs within a node are going to remain connected by NVLINK and a shared address space. Nonetheless, the internal NVLINK NIC should be extended by RDMA capabilities, for example by significantly improving access to copy engines and also allowing them to generate distinct notifications. In fact, it can be assumed that NVLINK's scale might be extended to a few nodes within a cabinet, for example. Second, EXTOLL's work request generation has been shown to be superior to Infiniband. The NIC that connects to other nodes should be accessible from the GPU and EXTOLL's interface seems promising as work requests are lean and simple to generate. Third, notifications have to be placed in GPU memory, which is possible with both EXTOLL and Infiniband. Overall, the NIC has to support both CPU and GPU communication models as it is shared. Dedicated NICs for each processor are not cost efficient and only increase the system's complexity.

GPU architecture Changing the GPU architecture requires to prove significant value by adding new features. The SM architecture has not added any new features until Volta and its tensor cores, driven by deep learning and its multi-billion market. The SoftNIC, as shown in Chapter 6, benefits significantly from hardware support for queuing, data management, scheduling, and events, but unless there is much greater value these features are not going to be implemented. Nonetheless, Volta's new threading model can help to implement much faster queues and matching algorithms and thread groups provide synchronization across CTAs. However, it is unclear how much the latter is supported by hardware or implemented in software. An overview of architectural improvements are shown in Table 7.1 with respect to the Volta architecture and how much it will improve certain aspects. It remains to be seen how deep learning algorithms evolve and how much they require GPUs to communicate directly with each other. If model parallelism is going to replace data parallelism at some point, GPU-sided communication and synchronization quickly become key to high training performance.

Another approach is followed by AMD, who combines the CPU and GPU to an APU. NVIDIA offers a similar architecture for their embedded vertical market, but has not shown any plans to expand this to the Tesla market, which gears to HPC and data centers. Nonetheless, if more and more features keep being added to the GPU to make it more general purpose, a simple Advanced RISC Machines (ARM) core on the GPU die might be beneficial. The core would be programmable and could implement SoftNIC functionality, for example, but could also be used for other applications that require faster single-thread performance.

7.3.2 Communication Model

As for now, the most prevalent communication model in HPC is MPI and this is not expected to change in the near future. However, this mostly concerns the CPU and with GPUs becoming interconnected another or adapted message passing model is required. NVLINK introduces a shared memory model, but is rather limited to smaller scale in current DGX-1 systems.

Another question is what it needs to further scale the shared memory model, as it also showed the best performance on GPUs in Chapter 5. The following elaborates on some of the challenges.

Address translation Scaling a shared memory model also means that the size of the shared address space growth linearly, but it also increases the number of pages that need to be addressable. One problem is the page table of each GPU, which needs to contain addresses of all pages in the network. Also, the TLB has to become larger to work efficiently. Hence, it is mandatory to find more efficient and better ways to translate addresses and support page migration between remote GPUs in order to scale out the shared memory model to a larger network

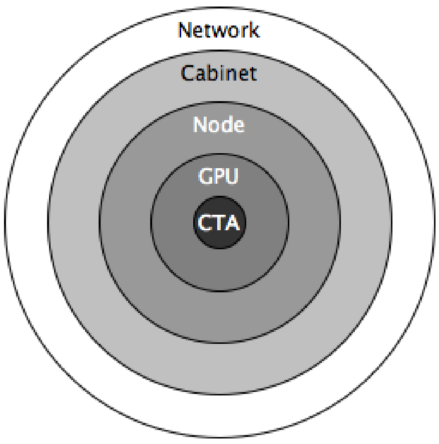


Fig. 7.2 Concentric consistency model as it is supported by current GPUs.

Consistency Consistency is a major concern in all shared memory approaches. Currently, GPUs see consistency in a scoped and concentric way as depicted in Figure 7.2. Fences are used to ensure previously posted memory operations are visible at the particular scope. However, extending the scope to a cabinet or even the whole system is too expensive in terms of latency as it concerns too many processors.

A better, more performant way is to be able to enforce consistency between objects or variables. Here, it only needs to be guaranteed that one or a group of variables are visible before another group or particular variable. The model enforces the order of write operations and their visibility to all processors. This is certainly required at larger scale and allows to scale to larger GPU networks.

Managing communication Although shared memory models allow for fine-grain accesses on word or page granularity, the user is required to know the address from which data is loaded or to which data is written. At larger scale, a large number of buffers might be used, which make it hard for the programmer to be aware of data locality, which is crucial to performance. Also, notifications have to be managed explicitly, which means notifications have to be found for particular requests and the programmer has to be aware of pending and outstanding operations. Even with systems like NVIDIA's DGX-1, not all GPUs can access other GPU's memory through NVLINK and thus the user has to handle the routing explicitly. This problem is amplified at larger scale.

In order to ensure large systems are still programmable and yield high performance, the system needs to provide suitable abstractions. While put/get semantics allow to express communication, as opposed to implicit communication by assignment, addresses and registrations still need to be known and administrated. Consequently, managed communication is important to provide high-level abstractions to hide much of the system's and network's complexity.

The right communication model The communication model is likely to become hierarchical. Looking at the systems in Figure 7.1, it can easily be imagined that GPUs within a node or cabinet communicate through a shared memory model, while inter-node communication still relies on message passing, for example MPI. However, due to complexity, reliability, and scalability reasons, the right model might be using tasks instead of a static rank-to-processor mapping. This also improves performance as resources can be better utilized and work is moved instead of data. A promising task-based model is Legion [33], for example. Nonetheless, the granularity of tasks is still to be determined as tasks can be as small as a kernel that runs on a single GPU, or as large as a group of processes running on an entire node. The latter requires the task itself to be decomposed to run on all available processors.

7.4 Managing Communication on GPUs

Chapter 6 and the previous section have shown that it will not be enough to simply allow the GPU to control the network by providing it access to the NIC. On the contrary, the GPU needs to manage communication to some extend. The level of abstraction, however, depends on the communication model, but generally the management is mostly limited by the following items.

7.4.1 Queuing

Asynchronous communication requires data structures in which messages can be stored until they are eventually consumed. Queues are an efficient data structure to exchange messages, but as shown, GPUs struggle with the queue's sequential nature. In order to en- or dequeue elements, various pointers have to be fetched, modified, and written back atomically. The proposed hardware extensions implement these operations inside the memory controller and allow for high-throughput queuing, which is necessary for both inter-CTA, but also inter-GPU communication.

CUDA 9.0 introduces the notion of *Cooperative Thread Groups* which allow for fast inter-thread synchronization and communication across warps. It is now possible to expand the warp-parallel de- and enqueue to more than 32 threads. The Volta GPU architecture goes even a step further by allowing threads to diverge, enabling more efficient queuing implementations. Nonetheless, hardware supported queuing is still desirable to cope with communication demands at larger scale and a massive amount of end-points.

7.4.2 Scheduling

When CTAs are scheduled to run on an SM they cannot be preempted, thus if messages are used for synchronization the GPU might end up in a deadlock situation. With preemption, CTAs could be set aside while they are awaiting messages and brought back when the message has arrived, perhaps triggered by an event.

Starting with the Pascal architecture, GPUs support instruction-level preemption for compute kernels. However, switching the context still takes up to 100μs [10] and should only be used infrequently. It seems unlikely that GPUs are going to implement and expose full CTA preemption in their API as the current scheduling and oversubscription are key to the GPU's high performance in many applications.

CUDA's aforementioned *Cooperative Thread Groups* allow for synchronization across multiple CTAs, given it is guaranteed all CTAs are resident on the GPU. The SoftNIC in its current form supports the persistent thread model, overcoming current scheduling limitations, but is at the same time limited in its applications.

7.4.3 Memory Management

Another significant difference between CPUs and GPUs is the memory management, which is much more dynamic on the CPU. There are two main issues resulting from the lack of dynamic memory management on GPUs. First, the SoftNIC and communication runtime may have to allocate memory to buffer incoming messages or to stage data for the transfer and application. Second, one-sided communication requires registrations, which the application should be able to allocate within the CUDA kernel as opposed to registering everything prior to the kernel launch.

Although there is existing work implementing dynamic memory management it is still an active research area and CUDA still does not provide any support. It is certainly desirable to have this feature in future GPU generations, perhaps even with specialized hardware acceleration. Nonetheless, Volta's new threading model improves the performance of rather sequential data structures, such as lists, but also hash tables as threads within a warp are able to diverge. These data structures are needed to implement dynamic memory allocators.

7.4.4 Data Copy Engines

Although it has been shown throughout this work that fine-grain communication is superior on GPUs, large and frequent data transfers can occupy a significant amount of resources. Hence, offloading data transfers allows for better overlap. The current Pascal P100 GPU architecture implements two copy engines to which data transfers can be offloaded, however, accessing them within CUDA kernels entails high latencies. Consequently, a faster direct access is required, but also a virtualized interface in order to be able to serve many concurrent requests. For example, each copy engine could implement multiple queues in which threads can enter requests in parallel.

7.5 Outlook

The major driver in GPU computing is certainly artificial intelligence, especially deep learning. Consequently, innovations in GPU architecture or in the programming model are mostly focused on benefiting these applications, which also share requirements with traditional scientific computing.

The latest Volta architecture implements tensor cores to accelerate tensor operations, mainly aimed at deep learning but also benefiting other linear algebra applications and thus many HPC workloads as well. Furthermore, the revised threading model in which threads can diverge allows for more flexibility and improved efficiency at tasks that require fine-grain synchronization, for example double-linked lists. This certainly improves the matching and queuing performance required by a SoftNIC implementation.

The most interesting innovation in regard to the SoftNIC is the *Cooperative Thread Group*. Due to the scheduling, the SoftNIC is mostly limited to the

Table 7.1 Architectural improvements to allow for better communication management on GPUs. The table shows what features are improved by Volta and values the improvement.

Item	**How Volta helps**	**Valuation on Volta**	**What is desired**
Queuing	Revised threading model (TM)	Medium	Queue instructions, supported by hardware
Scheduling	Cooperative Thread Groups (CTG)	Medium	Preemption, but CTG is sufficient for persistent thread model
Msg. Matching	Revised TM, CTG	High	Faster single-thread performance
Mem. Mngmnt.	Revised TM	Medium	Faster and built-in dynamic memory allocations within kernels
Data transfer	NVLINK 2	Low	Highly concurrent and virtualized copy engines
Synchronization	NVLINK 2, CTG	Medium	Better scheduling
Messaging	NVLINK 2	Medium	Better queuing, locally and remotely

persistent thread model, which is now significantly improved with the latest CUDA features on Pascal and Volta GPUs. Besides synchronization across CTAs, communication and collective primitives are going to be supported, even across multiple GPUs. This further allows to keep state information in registers instead of reloading everything after the kernel re-launch. A task model, in which thread groups receive tasks from a queue and create new tasks is now a feasible and attractive approach. Here, the SoftNIC can be used for group collectives and inter-node communication between groups running on different GPUs in different nodes.

As aforementioned, the major driver is deep learning. The size of neural nets is growing at a tremendous speed and the parallel training has gained a focus in research. While asynchronous data parallelism is still the most prevalent choice, model parallelism is becoming more important. Thus, communication and synchronization between GPUs is soon becoming the bottleneck in order to scale out deep learning. This will make a strong case for further enhancements in architecture and the programming model, rendering this work a good reference. With Volta's enhanced support for the persistent thread model the need for in-kernel communication becomes evident.

Chapter 8

Conclusion

Heterogeneous computing has become the standard in HPC to keep up with performance demands under power constraints that have become increasingly harder to meet. While computation is an interplay between CPUs and GPUs, communication is still homogeneous and entirely handled by the CPU.

With computing shifting toward GPU-centric systems, in which both processors are peers, communication needs to be specialized and heterogeneous, similar to computation. This work has presented and analyzed the GPU's capability to orchestrate communication and source and sink networking traffic, from the GPU's interaction with networking hardware to how communication can be managed in software to provide suitable communication abstractions that hide complexity from the user. Furthermore, several exascale applications were studied in regard to their communication behavior.

The application study showed that point-to-point communication is rather local and limited to a few peers. The most data is transferred through point-to-point operations like send/recv, but the most time is spent in collective communication, such as barrier, (all-)reduce, or all-to-all. This results from collective's implicit synchronization and is especially critical at large scale as imbalances are more likely and significant. Large scale also leads to a higher number of small messages, rendering the matching of messages with receive requests more important. Consequently, the matching and its queue structures were analyzed and it was found that queues barely exceed 512 messages. The analysis is important to identify boundary conditions and define design goals.

This investigation was followed by a comprehensive analysis of the interaction between GPU and NIC. Three fundamental communication models were presented: global shared address space, one-sided put/get, and traditional CPU-controlled MPI. Various benchmarks with common scientific communication patterns were used to assess these models in regard to execution time and energy spending dependent on the underlying communication model. Results show that GPU controlled communication is always favorable, not only in terms of time-to-solution but also regarding the energy spent to execute an application. Energy savings range between 10 and 50%. Although fine-grain communication using loads and stores shows the highest bandwidth and lowest latency, overlap with computation is limited. Thus, applications that allow for overlap benefit from one-sided put/get communication as more resources are available to the application and the actual copy operations are offloaded to the NIC hardware. Nonetheless, the proposed communication models are unmanaged with low-level abstractions, making it difficult to develop applications for large-scale systems.

The final part of this work considered communication management on GPUs, a widely unexplored research area. A concept was presented in which communication is handled by a kernel that runs along with the application. The kernel receives requests from the application through queue structures and performs communication tasks, occupying less resources as a single SM. This kernel is referred to as SoftNIC (Software NIC). First, queuing approaches were discussed and evaluated and it was concluded that additional hardware structures could improve the throughput significantly. Another focus has been on the message and receive request matching. Although an MPI protocol compliant algorithm was developed and assessed, a more relaxed protocol was presented that yields much higher message rates on GPUs. The most severe limitation with a software NIC is the scheduling, which is non-preemptive and only allows to apply a persistent thread model.

Although the current technology limits the SoftNIC to a persistent thread model and mainly tackles programmability rather than performance, trends and advances in the GPU architecture make this concept more and more attractive and feasible. On the one hand, GPUs are increasingly adding new features toward general purpose computing which eventually will promote them to first-class processors. On the other hand, the need for GPU controlled communication is becoming more important than ever, not least because of the rapidly expanding field of deep learning.

This work contributes a comprehensive study of GPU communication through the entire computing stack, from low-level interactions between GPU and NIC to the applications' communication behavior. This makes this work valuable to system architects but also to researchers that aim to take their applications to large scale and distributed GPU systems. In addition, this work also encourages more research in GPU architecture to focus on communication, but also to explore more scalable and GPU-centric programming models with suitable high-level abstractions. The SoftNIC can play an important part to hide complexity of large-scale GPU accelerated installations.

Acknowledgements

First and foremost I want to thank my parents, Dieter and Elfriede Klenk, for their unconditional and endless support. They have enabled me to pursue my goals and I will always be grateful for being blessed with such amazing parents. This dissertation is also for them.

I want to thank Prof. Dr. Holger Fröning for countless advice during the course of this work. He taught me research and many other things, always supported me, and gave me the opportunity to obtain a doctoral degree.

I would also like to express my gratitude to all members of the *Computer Architecture Group* of Prof. Dr. Ulrich Brüning and the employees of the *EXTOLL GmbH*. They provided help and advice at the early stage of this work and never hesitated to answer any questions. I am also grateful to the *Ruprecht-Karls University Heidelberg* and the *Faculty for Mathematics and Computer Science* for the opportunity to pursue the doctorate. I also want to thank Dr. Lena Oden for her support and contributions. She laid the foundation to this work with her research. I appreciate Günther Schindler's and Arthur Kühlwein's master thesis and project report, which contributed also to this work.

Special thanks to Dr. Hans Eberle for mentoring me during my two internships at *NVIDIA Corp.* He taught me well in scientific methodologies and how to address research problems. I also thank Dr. Larry Dennison who has been very inspiring and provided me with the opportunity to serve these internships. I would like to think his networking research team and all the people I interacted with at NVIDIA for their support and candor. The results of these internships contributed greatly to this work.

Last but not least, I want to express my sincere appreciation to the people that have always been around and supported me in so many ways: my friends. Thanks to Daniel Schlegel, who I have known since elementary school and who also contributed to this dissertation with his master thesis and project report.

Many thanks to my roommate Juri Schmidt and my "office-mate" Alexander Matz, who I wish all the best for their own doctoral dissertations. Thanks to Christine Harvey, who has supported me since the very beginning of my doctoral studies.

I also want to sincerely thank Felix Zahn. He has been an amazing and very supportive friend. I also wish him all the best for his doctoral dissertation and following career.

Many of this would not have turned out the way it did without Melissa Lam. She has always supported me and stood by my side and I can’t even express how grateful I am to have her in my life. I can’t wait for all the memories we will create together.

Since it’s impossible to thank everyone that has accompanied me on this journey I want to thank all the amazing people I met at conferences and all over the world during the past years. Many acquaintanceships turned into friendships and I’m grateful for all of you.

List of figures

List of tables

Acronyms

APGAS Asynchronous Partitioned Global Address Space. 30

API Application Programmable Interface. 14, 26, 85, 149, 171

APU Accelerated Processing Unit. 148, 168

ARM Advanced RISC Machines. 168

ASIC Application Specific Integrated Circuit. 72

ATU Address Translation Unit. 72, 73, 87

BAR Base Address Register. 74, 78, 79, 81, 83, 85, 87, 90, 91, 94, 101, 102

BSP Bulk Synchronous Parallel. 101

CAS Compare-And-Swap. 121, 124, 126, 129, 130, 132, 133

CNN Convolutional Neural Net. 66

CNTK Microsoft Cognitive Toolkit. 148

CPU Central Processing Unit. vii–ix, 2–7, 9, 12–17, 25, 31, 50, 56, 57, 59, 63–65, 67, 69, 70, 75–78, 82, 83, 85–88, 91, 92, 96–100, 102–105, 107, 108, 111, 112, 115, 116, 120, 121, 135, 136, 138, 141, 142, 147, 148, 150, 151, 153, 154, 159, 161, 164–168, 171, 175, 176, 187

CTA Collaborative Thread Array. 11, 15, 82, 84, 116, 117, 120, 121, 125–130, 132, 133, 138, 139, 141, 142, 148, 149, 152–157, 167, 170, 171, 174

CUDA Compute Unified Device Architecture. 3, 9–11, 13–16, 78, 82, 84, 97, 98, 111, 112, 116, 135, 138, 171, 172, 174

CUDPP CUDA Data Parallel Primitives. 147

DCA Direct Cache Access. 65

DCAS Double Compare-And-Swap. 131

DDR Double Data Rate. 12

DMA Direct Memory Access. 64, 69–71, 74, 85, 88, 102, 112, 134, 135

DOE U.S. Department of Energy. 31, 38, 187

DSM Distributed Shared Memory. 26, 27, 29

FAA Fetch And Add. 121

FFT Fast Fourier Transformation. 66

FIFO First In First Out. 73, 151

FLOPS Floating Point Operations Per Second. 5, 52

FMA Fused Multiply Add. 4

FPGA Field Programmable Gate Array. 91, 103, 151

GAS Global Address Space. 77, 98

GASNet Global Address Space Network. 26, 28, 30, 148

GGAS GPU Global Address Space. 77–88, 96, 100–110, 112, 162, 182

GPU Graphics Processing Unit. vii–x, 2–7, 9–17, 19, 30, 50–53, 55–60, 62, 63, 65, 67, 70, 75–88, 90–100, 102–113, 115, 116, 120–122, 125, 130–138, 141, 142, 144, 145, 147–157, 159–172, 174–177, 182, 185, 187

HBM High Bandwidth Memory. 67

HCA Host Channel Adapter. 67, 70, 85, 86, 93

HPC High Performance Computing. vii, 2–4, 7, 9, 22, 26, 29–31, 50, 66, 97, 102, 162, 168, 172, 175

HSA Heterogeneous System Architecture. 148

HSHUB High Speed Hub. 75

HT Hyper Transport. 65

I/O Input/Output. 65, 99, 108, 119, 159

LL/SC Load-Link/Sore-Conditional. 131

MMIO Memory Mapped I/O. 65, 74, 78, 83, 84, 86, 87, 94, 131

MPI Message Passing Interface. 4, 7, 15–17, 22–27, 29–37, 40, 44–46, 48, 50, 60, 61, 63, 65, 66, 83, 85, 87, 96–98, 100, 102–112, 136, 138, 141, 142, 147, 148, 161, 162, 164, 168, 170, 176, 181, 187

MPMC Multiple Producer Multiple Consumer. 120, 121

MPMD Multiple Program, Multiple Data. 20

MPP Massively Parallel Processing. 20

MPSD Multiple Program, Single Data. 20

NIC Network Interface Controller. 6, 16, 17, 19, 44, 64–73, 75, 77, 79, 83, 85–88, 90, 91, 93, 94, 96–100, 103, 104, 112, 116–118, 134, 148, 151, 162, 166, 167, 170, 176, 177, 182, 187

NLA Network Logical Address. 72, 73, 87

NUMA Non-Uniform Memory Access. 19, 75, 113

OS Operating System. 1–3, 159

PCI Peripheral Component Interconnect. 70

PCIe Peripheral Component Interconnect Express. 16, 17, 44, 65–67, 70, 71, 74–76, 78, 79, 81, 83, 85, 88, 90–92, 94, 97, 101, 102, 116, 135, 150, 166

PCM Phase Change Memory. 67

PGAS Partitioned Global Address Space. 7, 26, 28, 30, 68, 83, 85, 99, 111, 112, 134, 157

PRQ Posted Receive Queue. 24, 33, 44–46, 48, 138

PTX Parallel Thread Execution. 14

QCD Quantum Chromodynamics. 66

QP Queue Pair. 70, 71, 85, 86, 93

RAPL Running Average Power Limit. 108

RDMA Remote Direct Memory Access. 23, 26, 77, 85, 87, 90, 96, 100, 102, 103, 148, 167

RMA Remote Memory Access. 72–74, 85, 87, 91, 94, 100, 104–110, 112, 182

RPC Remote Procedure Call. 26, 30

SIMD Single Instruction Multiple Data. 2

SIMT Single Instruction Multiple Threads. 2, 10

SM Streaming Multiprocessor. 9–12, 84, 116, 117, 131, 132, 134–136, 142, 153, 154, 156, 157, 167, 171, 176

SMFU Shared Memory Functional Unit. 70, 72, 74, 75, 77–79, 81

SMI System Management Interface. 108

SMP Symmetric Multiprocessor. 27, 75, 166

SOOP Software Out-Of-Order Processor. 29

SPMD Single Program, Multiple Data. 20, 22

SPSC Single Producer Single Consumer. 120

SPSD Single Program, Single Data. 20

SSD Solid State Disk. 77

SST Structural Simulation Toolkit. 32

STL Standard Template Library. 45, 46, 120

TLB Translation Lookaside Buffer. 28, 135, 154, 168

UMA Uniform Memory Access. 19

UMQ Unexpected Message Queue. 24, 33, 44, 45, 48, 138, 147

UPC Unified Parallel C. 26, 28

UVA Unified Virtual Addressing. 14, 16, 75, 116

VAS Virtual Address Space. 73–75, 77, 85

VELO Virtualized Engine for Low Overhead. 72

VPID Virtual Process ID. 73, 87

References

[1] M. Y. Vardi, “Science has only two legs,” *Communications of the ACM*, vol. 53, no. 9, 2010 (cit. on p. 1).

[2] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ISBN: 012383872X, 9780123838728 (cit. on pp. 2, 10, 14, 20, 21).

[3] E. Strohmaier, J. Dongorra, H. Simon, and M. Meuer. (Nov. 2016). Top500 - The List. Last visited on May 17, 2017, [Online]. Available: https://www.top500.org (cit. on pp. 2, 20).

[4] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted MOSFET’s with very small physical dimensions,” *Journal of Solid-State Circuits (JSSC)*, vol. 9, no. 5, pp. 256–268, 1974 (cit. on p. 3).

[5] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *SIGARCH Computer Architecture News*, ACM, vol. 39, 2011, pp. 365–376 (cit. on p. 3).

[6] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A parallel programming standard for heterogeneous computing systems,” *Computing in science and engineering*, vol. 12, no. 3, pp. 66–73, 2010 (cit. on p. 3).

[7] R. Farber, *Parallel programming with OpenACC*. Newnes, 2016, ISBN: 9780124103979 (cit. on p. 3).

[8] S. Lee, S.-J. Min, and R. Eigenmann, “OpenMP to GPGPU: A compiler framework for automatic translation and optimization,” *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009 (cit. on p. 3).

[9] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *AFIPS Conference Proceedings of the Apr. 18-20 spring joint computer conference*, ACM, 1967, pp. 483–485 (cit. on p. 3).

[10] NVIDIA Corp., “NVIDIA Tesla P100,” Whitepaper, 2016, Last visited on July 14, 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (cit. on pp. 4, 11, 65, 75, 76, 155, 171).

[11] ASCR Advisory Committee, “The opportunities and challenges of exascale computing,” Advanced Scientific Computing Advisory Committee (ASCAC), Exascale Advisory Committee Report, 2010. [Online]. Available: http://science.energy.gov/ascr/ascac/ (cit. on pp. 4, 31).

[12] J. A. Stuart and J. D. Owens, “Message passing on data-parallel architectures,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, Rome, Italy, May 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5161065 (cit. on pp. 5, 98, 161).

[13] T. Gysi, J. Baer, and T. Hoefler, “dCUDA: Hardware supported overlap of computation and communication,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, IEEE, Salt Lake City, UT, Nov. 2016 (cit. on pp. 5, 87, 161).

[14] M. Silberstein, S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, and E. Witchel, “GPUnet: Networking abstractions for GPU programs,” *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 3, p. 9, 2016 (cit. on pp. 5, 98, 99, 161).

[15] L. Oden, “Direct communication between distributed GPUs,” Doctoral Dissertation, Ruprecht-Karls University Heidelberg, Mannheim, Germany, 2014 (cit. on pp. 5, 77, 82, 86, 162).

[16] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, *et al.*, “Scaling the power wall: A path to exascale,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, IEEE, 2014, pp. 830–841 (cit. on p. 5).

[17] B. Klenk and H. Fröning, “An overview of MPI characteristics of exascale proxy applications,” in *International Supercomputing Conference (ISC)*, Frankfurt, Germany, Jun. 2017 (cit. on pp. 6, 32, 39–41, 43, 47, 49, 61).

[18] B. Klenk, L. Oden, and H. Fröning, “GPU-centric communication for improved efficiency,” in *Workshop on Green Programming, Computing, and Data Processing (GPCDP)*, Invited paper, Dallas, TX, Nov. 2014 (cit. on pp. 6, 63).

[19] ——, “Analyzing put/get APIs for thread-collaborative processors,” in *Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications (HUCAA)*, Minneapolis, MN, Sep. 2014, pp. 411–418. DOI: 10.1109/ICPPW.2014.61 (cit. on pp. 6, 63, 92–94).

[20] ——, “Analyzing communication models for distributed thread-collaborative processors in terms of energy and time,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Philadelphia, PA, Mar. 2015, pp. 318–327 (cit. on pp. 6, 63, 103, 104, 108).

[21] B. Klenk, H. Fröning, H. Eberle, and L. Dennison, “Relaxations for high-performance message passing on massively parallel SIMT processors,” in *International Parallel Distributed Processing Symposium (IPDPS)*, IEEE, Orlando, FL, May 2017 (cit. on pp. 7, 115, 139–141, 143).

[22] M. Macedonia, “The GPU enters computing’s mainstream,” *Computer*, vol. 36, no. 10, pp. 106–108, 2003 (cit. on p. 9).

[23] NVIDIA Corp., *CUDA programming guide*, version 8.0.61, 2017 (cit. on pp. 9, 84).

[24] S. Cook, *CUDA programming: A developer’s guide to parallel computing with GPUs*. Newnes, 2012, ISBN: 9780124159334 (cit. on p. 9).

[25] N. Wilt, *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013, ISBN: 9780133261509 (cit. on p. 9).

[26] NVIDIA Corp., “Kepler GK110,” Whitepaper, 2015, Last visited on Mai 19, 2017. [Online]. Available: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (cit. on p. 10).

[27] J. Wang and S. Yalamanchili, “Characterization and analysis of dynamic parallelism in unstructured GPU applications,” in *International Symposium on Workload Characterization (IISWC)*, IEEE, 2014, pp. 51–60 (cit. on p. 15).

[28] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” in *European PVM/MPI Users’ Group Meeting*, Budapest, Hungary, Sep. 2004, pp. 97–104 (cit. on pp. 22, 46).

[29] Argonne National Laboratory. (2017). MPICH. Last visited on February 2, 2017, [Online]. Available: https://www.mpich.org/ (cit. on pp. 22, 46).

[30] Ohio State University. (2017). MVAPICH. Last visited on February 2, 2017, [Online]. Available: http://mvapich.cse.ohio-state.edu/ (cit. on pp. 22, 46).

[31] D. Bonachea, “GASNet specification, v1.8,” Berkeley, CA, USA, Tech. Rep., 2008 (cit. on p. 26).

[32] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, “Optimizing bandwidth limited problems using one-sided communication and overlap,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, Apr. 2006, 10 pp. DOI: 10.1109/IPDPS.2006.1639320 (cit. on p. 26).

[33] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, IEEE, Salt Lake City, Utah: IEEE Computer Society Press, 2012, 66:1–66:11, ISBN: 978-1-4673-0804-5 (cit. on pp. 26, 29, 170).

[34] R. Brightwell, B. Lawry, A. B. MacCabe, and R. Riesen, “Portals 3.0: Protocol building blocks for low overhead communication,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, Washington, DC, USA, 2002, pp. 268– (cit. on pp. 26, 44).

[35] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, “Introducing OpenSHMEM: SHMEM for the PGAS community,” in *Conference on Partitioned Global Address Space Programming Model (PGAS)*, ACM, 2010, p. 2 (cit. on p. 26).

[36] J. Armstrong, “Making reliable distributed systems in the presence of software errors,” PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003 (cit. on p. 26).

[37] N. Raychaudhuri, *Scala in action.* Manning Publications Co., 2013 (cit. on p. 27).

[38] D. Lenoski and W. Weber, *Scalable shared-memory multiprocessing.* Elsevier Science, 2014, ISBN: 9781483296012 (cit. on pp. 27, 28).

[39] B. L. Chamberlain, D. Callahan, and H. P. Zima, “Parallel programmability and the chapel language,” *The International Journal of High Performance Computing Applications (IJHPCA)*, vol. 21, no. 3, pp. 291–312, 2007 (cit. on p. 28).

[40] S. Treichler, M. Bauer, and A. Aiken, “Realm: An event-based low-level runtime for distributed memory architectures,” in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Edmonton, AB, Canada: ACM, 2014, pp. 263–276. DOI: 10.1145/2628071.2628084 (cit. on pp. 30, 156).

[41] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, “Ompss: A proposal for programming heterogeneous multicore architectures,” *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011 (cit. on p. 30).

[42] 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,” in *SIGPLAN Notices*, ACM, vol. 40, 2005, pp. 519–538 (cit. on p. 30).

[43] P. Suter, O. Tardieu, and J. Milthorpe, “Distributed programming in scala with APGAS,” in *SIGPLAN Symposium on Scala*, ACM, 2015, pp. 13–17 (cit. on p. 30).

[44] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996 (cit. on p. 30).

[45] S. Rajasekaran, L. Fiondella, M. Ahmed, and R. Ammar, *Multicore computing: Algorithms, architectures, and applications*, ser. Chapman & Hall/CRC Computer and Information Science Series. Taylor & Francis, 2013, ISBN: 9781439854341 (cit. on p. 30).

[46] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *International Parallel Distributed Processing Symposium (IPDPS)*, IEEE, May 2013, pp. 712–725. DOI: 10.1109/IPDPS.2013.78 (cit. on p. 30).

[47] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: The new adventures of old X10," in *International Conference on Principles and Practice of Programming in Java (PPPJ)*, ACM, 2011, pp. 51–61 (cit. on p. 30).

[48] J. Hsu, "When will we have an exascale supercomputer?" *IEEE Spectrum*, vol. 52, no. 1, pp. 13–16, 2015 (cit. on p. 31).

[49] U. D. of Energy. (2016). Characterization of the DOE Mini-apps. Last visited on May 17, 2017, [Online]. Available: https://portal.nersc.gov/project/CAL/doe-miniapps.htm (cit. on p. 31).

[50] D. Dechev and G. Hendry, "A macroscale simulator for exascale software/hardware co-design," *Journal of Information Technology and Software Engineering*, vol. 3, no. 3, p. 1, 2013 (cit. on p. 32).

[51] A. Faraj and X. Yuan, "Communication characteristics in the NAS parallel benchmarks.," in *International Conference on Parallel and Distributed Computing and Systems (PDCS)*, IASTED, 2002, pp. 724–729 (cit. on pp. 33, 162).

[52] R. Riesen, "Communication patterns [message-passing patterns]," in *International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2006, 8–pp (cit. on pp. 33, 162).

[53] S. Kamil, L. Oliker, A. Pinar, and J. Shalf, "Communication requirements and interconnect optimization for high-end scientific applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 2, pp. 188–202, 2010 (cit. on pp. 34, 162).

[54] J. S. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," *Journal of Parallel and Distributed Computing*, vol. 63, no. 9, pp. 853–865, 2003 (cit. on pp. 34, 162).

[55] P. G. Raponi, F. Petrini, R. Walkup, and F. Checconi, "Characterization of the communication patterns of scientific applications on Blue Gene/P," in *International Parallel and Distributed Processing Symposium, Workshops, and Phd Forum (IPDPSW)*, IEEE, 2011, pp. 1017–1024 (cit. on pp. 34, 162).

[56] M. J. Koop, W. Huang, K. Gopalakrishnan, and D. K. Panda, "Performance analysis and evaluation of PCIe 2.0 and quad-data rate infiniband," in *Symposium on High Performance Interconnects*, IEEE, Aug. 2008, pp. 85–92 (cit. on p. 44).

[57] R. Keller and R. L. Graham, "Characteristics of the unexpected message queue of MPI applications," in *European MPI Users' Group Meeting*, Springer, 2010, pp. 179–188 (cit. on pp. 45, 162).

[58] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, IEEE, 2010, pp. 1–10 (cit. on pp. 50, 83, 162).

[59] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *International Conference on Machine Learning (ICML)*, 2013, pp. 1337–1345 (cit. on p. 50).

[60] B. Catanzaro, *HPC: Powering deep learning, Smoky mountains computational sciences and engineering conference (smc)*, Presentation, Gatlinburg, Tennessee, 2015 (cit. on p. 50).

[61] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning (adaptive computation and machine learning series)*. Cambridge: The MIT Press, 2016, ISBN: 9780262035613 (cit. on p. 51).

[62] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009 (cit. on p. 52).

[63] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems (NIPS)*, 2012, pp. 1097–1105 (cit. on pp. 53, 57, 62).

[64] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server.," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 14, 2014, pp. 583–598 (cit. on p. 54).

[65] S. Gupta, W. Zhang, and J. Milthorpe, "Model accuracy and runtime tradeoff in distributed deep learning," *ArXiv preprint arXiv:1509.04210*, 2015 (cit. on pp. 55, 163).

[66] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *ArXiv preprint arXiv:1512.01274*, 2015 (cit. on p. 56).

[67] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ArXiv preprint arXiv:1409.1556*, 2014 (cit. on pp. 57, 62).

[68] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2015, pp. 1–9 (cit. on pp. 57, 62).

[69] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2016, pp. 770–778 (cit. on pp. 57, 62).

[70] B. Baumann, "A performance model for the training of DNNs on GPUs," Master Thesis, Ruprecht-Karls University Heidelberg, Mannheim, Germany, 2016 (cit. on p. 59).

[71] L. Oden, B. Klenk, and H. Fröning, "Energy-efficient collective reduce and allreduce operations on distributed GPUs," in *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, ACM, IEEE, Chicago, IL, May 2014, pp. 483–492. DOI: 10.1109/CCGrid.2014.21 (cit. on pp. 63, 82, 83, 102).

[72] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct cache access for high bandwidth network I/O," in *International Symposium on Computer Architecture (ISCA)*, Washington, DC, USA: IEEE Computer Society, 2005, pp. 50–59. DOI: 10.1109/ISCA.2005.23 (cit. on p. 65).

[73] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel omni-path architecture: Enabling scalable, high performance fabrics," in *Annual Symposium on High-Performance Interconnects*, Aug. 2015, pp. 1–9. DOI: 10.1109/HOTI.2015.22 (cit. on pp. 65, 66).

[74] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, IEEE, Austin, Texas, 2015, 30:1–30:12. DOI: 10.1145/2807591.2807602 (cit. on p. 65).

[75] P. Lai, P. Balaji, R. Thakur, and D. K. Panda, "ProOnE: A general-purpose protocol onload engine for multi- and many-core architectures," in *Computer Science - Research and Development (CSRD)*, Springer Berlin / Heidelberg, Springer Berlin / Heidelberg, May 2009 (cit. on p. 66).

[76] G. Shainer and A. Ancel, "Network-based processing versus host-based processing: Lessons learned," in *International Conference on Cluster Computing, Workshops, and Posters (CLUSTER WORKSHOPS)*, IEEE, Sep. 2010, pp. 1–4. DOI: 10.1109/CLUSTERWKSP.2010.5613096 (cit. on p. 67).

[77] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: A hardware/software approach.* Gulf Professional Publishing, 1999, ISBN: 9781558603431 (cit. on pp. 67, 68).

[78] W. Dally and B. Towles, *Principles and practices of interconnection networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, ISBN: 0122007514 (cit. on p. 69).

[79] R. Buyya, T. Cortes, and H. Jin, *An introduction to the infiniband architecture*. Wiley-IEEE Press, 2002, pp. 616–632, ISBN: 9780470544839. DOI: 10.1109/9780470544839.ch42 (cit. on pp. 70, 71).

[80] M. Nüssle, M. Scherer, and U. Brüning, "A resource optimized remote-memory-access architecture for low-latency communication," in *International Conference on Parallel Processing (ICPP)*, Vienna, Austria, Sep. 2009, pp. 220–227. DOI: 10.1109/ICPP.2009.62 (cit. on p. 73).

[81] Mondrian Nüssle, "Acceleration of the Hardware-Software Interface of a Communication Device for Parallel Systems," PhD thesis, University of Mannheim, 2008 (cit. on pp. 73, 74).

[82] H. Fröning and H. Litz, "Efficient hardware support for the partitioned global address space," in *International Parallel Distributed Processing Symposium, Workshops, and Phd Forum (IPDPSW)*, IEEE, Atlanta, GA, Apr. 2010, pp. 1–6. DOI: 10.1109/IPDPSW.2010.5470851 (cit. on p. 74).

[83] H. Montaner, F. Silla, H. Fröning, and J. Duato, "MEMSCALE: A scalable environment for databases," in *International Conference on High Performance Computing and Communications (HPCC)*, IEEE, Sep. 2011, pp. 339–346. DOI: 10.1109/HPCC.2011.51 (cit. on pp. 75, 77).

[84] Alexandre Bicas Caldeira and Volker Haug and Scott Vetter, "IBM power system S822LC for high performance computing introduction and technical overview," Whitepaper, 2016, Last visited on June 12, 2017. [Online]. Available: http://www.redbooks.ibm.com/redpapers/pdfs/redp5405.pdf (cit. on p. 76).

[85] L. Oden and H. Fröning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *International Conference on Cluster Computing (CLUSTER)*, IEEE, Indianapolis, IN, Sep. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702638 (cit. on pp. 77, 162).

[86] H. Fröning, A. Giese, H. Montaner, F. Silla, and J. Duato, "Highly scalable barriers for future high-performance computing clusters," in *International Conference on High Performance Computing (HiPC)*, IEEE, 2011, pp. 1–10 (cit. on p. 82).

[87] S. Xiao and W.-c. Feng, "Inter-block GPU communication via fast barrier synchronization," in *International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2010, pp. 1–12 (cit. on p. 82).

[88] M. G. Venkata, P. Shamis, R. Sampath, R. L. Graham, and J. S. Ladd, "Optimizing blocking and nonblocking reduction operations for multicore systems: Hierarchical design and implementation," in *International Conference on Cluster Computing (CLUSTER)*, IEEE, 2013, pp. 1–8 (cit. on p. 83).

[89] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU concurrency: Weak behaviours and programming assumptions," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 577–591, 2015 (cit. on p. 84).

[90] L. Oden, H. Fröning, and F. J. Pfreundt, "Infiniband-verbs on GPU: A case study of controlling an infiniband network device from the GPU," in *International Journal of High Performance Computing Applications, Special Issue on Applications for the Heterogeneous Computing Era*, 2015 (cit. on pp. 85, 86, 162).

[91] F. Daoud, A. Watad, and M. Silberstein, "GPUrdma: GPU-side library for high performance networking from GPU kernels," in *Nternational Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, Kyoto, Japan: ACM, 2016, 6:1–6:8. DOI: 10.1145/2931088.2931091 (cit. on p. 86).

[92] D. Schlegel, "Heterogeneous memory support for the EXTOLL BufferQueue," Project Report, Ruprecht-Karls University Heidelberg, Mannheim, Germany, 2016 (cit. on pp. 94, 95).

[93] R. Ammendola, M. Bernaschi, A. Biagioni, M. Bisson, M. Fatica, O. Frezza, F. L. Cicero, A. Lonardo, E. Mastrostefano, P. S. Paolucci, *et al.*, "GPU peer-to-peer techniques applied to a cluster interconnect," in *International Parallel and Distributed Processing Symposium, Workshops, and PhD Forum (IPDPSW)*, IEEE, 2013, pp. 806–815 (cit. on p. 97).

[94] W. Wu, G. Bosilca, R. vandeVaart, S. Jeaugey, and J. Dongarra, "GPU-aware non-contiguous data movement in Open MPI," in *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, Kyoto, Japan: ACM, 2016, pp. 231–242. DOI: 10.1145/2907294.2907317 (cit. on pp. 97, 136).

[95] A. Venkatesh, K. Hamidouche, H. Subramoni, and D. K. Panda, "Offloaded GPU collectives using CORE-direct and CUDA capabilities on infiniband clusters," in *International Conference on High Performance Computing (HiPC)*, Dec. 2015, pp. 234–243. DOI: 10.1109/HiPC.2015.50 (cit. on p. 98).

[96] NVIDIA Corp. (2017). NCCL. Last visited on May 17, 2017, [Online]. Available: https://github.com/NVIDIA/nccl. (cit. on p. 98).

[97] S. Jeaugey, *NCCL 2.0*, Technical presentation, San Jose, CA, 2017 (cit. on p. 98).

[98] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking abstractions for GPU programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 201–216 (cit. on pp. 98, 99).

[99] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990 (cit. on p. 101).

[100] P. Luszczek, 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,” Tech. Rep., 2005 (cit. on p. 102).

[101] B. Klenk, “Comparing different communication paradigms for data-parallel processors,” master thesis, Ruprecht-Karls University Heidelberg, Mannheim, Germany, 2013 (cit. on pp. 105–107).

[102] D. Cederman, B. Chatterjee, and P. Tsigas, “Understanding the performance of concurrent data structures on graphics processors,” in *European Conference on Parallel Processing (Euro-Par)*, Springer, 2012, pp. 883–894 (cit. on p. 120).

[103] T. Preud’Homme, J. Sopena, G. Thomas, and B. Folliot, “Batchqueue: Fast and memory-thrifty core to core communication,” in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, 2010, pp. 215–222 (cit. on p. 120).

[104] P. Tsigas and Y. Zhang, “A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems,” in *Symposium on Parallel Algorithms and Architectures (SPAA)*, ACM, 2001, pp. 134–143 (cit. on p. 121).

[105] T. R. W. Scogland and W.-c. Feng, “Design and evaluation of scalable concurrent queues for many-core architectures,” 2015 (cit. on p. 121).

[106] M. Bauer, H. Cook, and B. Khailany, “CudaDMA: Optimizing GPU memory bandwidth via warp specialization,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, IEEE, Seattle, Washington, 2011, 12:1–12:11. DOI: 10.1145/2063384.2063400 (cit. on p. 128).

[107] O. Agesen, D. L. Detlefs, C. H. Flood, A. T. Garthwaite, P. A. Martin, N. N. Shavit, and G. L. Steele Jr, “DCAS-based concurrent deques,” in *Symposium on Parallel Algorithms and Architectures (SPAA)*, ACM, 2000, pp. 137–146 (cit. on p. 131).

[108] T. Brown, F. Ellen, and E. Ruppert, “Pragmatic primitives for non-blocking data structures,” in *Symposium on Principles of Distributed Computing (PODC)*, ACM, 2013, pp. 13–22 (cit. on p. 131).

[109] G. Schindler, “GPU architecture extensions for advanced communication and synchronization,” Master Thesis, Ruprecht-Karls University Heidelberg, Mannheim, Germany, 2016 (cit. on pp. 131, 133, 157).

[110] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2009, pp. 163–174 (cit. on pp. 131, 132).

[111] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms, third edition*, 3rd. The MIT Press, 2009, ISBN: 0262033844, 9780262033848 (cit. on p. 143).

[112] A. Kühlwein, "Hash tables for unordered message matching on SIMT processors," Master Thesis, Ruprecht-Karls University Heidelberg, Mannheim, Germany, 2017 (cit. on pp. 144–147).

[113] P. Celis, P.-A. Larson, and J. I. Munro, "Robin hood hashing," in *Symposium on Foundations of Computer Science (FOCS)*, IEEE, 1985, pp. 281–288 (cit. on p. 144).

[114] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004 (cit. on p. 144).

[115] T. Wang. (2007). Integer hash function. Last visited on May 17, 2017, [Online]. Available: http://www.concentric.net/~ttwang/tech/inthash.htm (cit. on p. 145).

[116] R. Jenkins. (2006). 4-byte integer hashing. Last visited on May 17, 2017, [Online]. Available: http://burtleburtle.net/bob/hash/integer.html (cit. on p. 145).

[117] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Building an efficient hash table on the GPU," *GPU Computing Gems*, vol. 2, pp. 39–53, 2011 (cit. on p. 147).

[118] J. A. Zounmevo and A. Afsahi, "An efficient MPI message queue mechanism for large-scale jobs," in *International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2012, pp. 464–471 (cit. on p. 147).

[119] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI message matching misery," in *International Supercomputing Conference (ISC)*, Springer, Frankfurt, Germany, 2016, pp. 281–299 (cit. on p. 147).

[120] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda, "Adaptive and dynamic design for MPI tag matching," in *International Conference on Cluster Computing (CLUSTER)*, IEEE, 2016, pp. 1–10 (cit. on p. 148).

[121] H.-V. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *European MPI Users' Group Meeting*, ACM, 2016, pp. 1–14 (cit. on p. 148).

[122] M. LeBeane, B. Potter, A. Pan, A. Dutu, V. Agarwala, W. Lee, D. Majeti, B. Ghimire, E. Van Tassell, S. Wasmundt, B. Benton, M. Breternitz, M. L. Chu, M. Thottethodi, L. K. John, and S. K. Reinhardt, "Extended task queuing: Active messages for heterogeneous systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, IEEE, 2016, p. 80 (cit. on p. 148).

[123] D. Schlegel, "Active messaging in autonomous GPU networks," Master Thesis, Ruprecht-Karls University Heidelberg, Mannheim, Germany, 2016 (cit. on pp. 150, 151).

[124] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. mei W. Hwu, “XMalloc: A scalable lock-free dynamic memory allocator for many-core machines.,” in *International Conference on Computer and Information Technology (CIT)*, IEEE, IEEE Computer Society, 2010, pp. 1134–1139 (cit. on pp. 151, 152).

[125] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, “Scatteralloc: Massively parallel dynamic memory allocation for the GPU,” in *Innovative Parallel Computing (InPar)*, San Jose, CA, 2012 (cit. on p. 152).

[126] S. Widmer, D. Wodniok, N. Weber, and M. Goesele, “Fast dynamic memory allocator for massively parallel architectures,” in *Workshop on General Purpose Processing Using Graphics Processing Units (GPGPU)*, Houston, Texas, USA: ACM, 2013, pp. 120–126, ISBN: 978-1-4503-2017-7. DOI: 10.1145/2458523.2458535 (cit. on p. 152).

[127] D. P. Andrew V. Adinetz, “Halloc: A high-throughput dynamic memory allocator for GPGPU architectures,” in *GPU Technology Conference (GTC)*, San Jose, CA, 2014 (cit. on p. 152).

[128] M. Vinkler and V. Havran, “Register efficient dynamic memory allocator for GPUs,” in *Computer Graphics Forum*, Wiley Online Library, vol. 34, 2015, pp. 143–154 (cit. on p. 152).

[129] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, “Enabling preemptive multiprogramming on GPUs,” in *SIGARCH Computer Architecture News*, ACM, vol. 42, 2014, pp. 193–204 (cit. on p. 154).

[130] J. J. K. Park, Y. Park, and S. Mahlke, “Chimera: Collaborative preemption for multitasking on a shared GPU,” *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 593–606, 2015 (cit. on p. 154).

[131] Z. Lin, L. Nyland, and H. Zhou, “Enabling efficient preemption for SIMT architectures with lightweight context switching,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, IEEE, 2016, p. 77 (cit. on p. 155).

[132] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, “LaPerm: Locality aware scheduler for dynamic parallelism on GPUs,” in *International Symposium on Computer Architecture (ISCA)*, IEEE, 2016, pp. 583–595 (cit. on p. 155).

[133] G. Chen, Y. Zhao, X. Shen, and H. Zhou, “EffiSha: A software framework for enabling effficient preemptive scheduling of GPU,” in *SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM, 2017, pp. 3–16 (cit. on p. 155).

[134] R. Brightwell and K. D. Underwood, “An analysis of NIC resource usage for offloading MPI,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2004, p. 183 (cit. on p. 162).

[135] J. Wells, B. Bland, J. Nichols, J. Hack, F. Foertter, G. Hagen, T. Maier, M. Ashfaq, B. Messer, and S. Parete-Koon, “Announcing supercomputer summit,” Oak Ridge National Laboratory (ORNL), Oak Ridge, TN, USA, Tech. Rep., 2016 (cit. on p. 165).