Improving the Accessibility of Heterogeneous System Resources for Application Developers using Programming Abstractions

Verbesserung der Zugänglichkeit heterogener Systemressourcen für Anwendungsentwickler durch Programmierabstraktionen

Max Frederik Plauth

Dissertation
zur Erlangung des Doktorgrades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Digital Engineering Fakultät
der Universität Potsdam

Improving the Accessibility of Heterogeneous System Resources for Application Developers using Programming Abstractions

Verbesserung der Zugänglichkeit heterogener Systemressourcen für Anwendungsentwickler durch Programmierabstraktionen

Max Frederik Plauth

Dissertation
zur Erlangung des Doktorgrades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Digital Engineering Fakultät
der Universität Potsdam

Betreuer: Prof. Dr. rer. nat. habil. Andreas Polze
Universität Potsdam,
Digital Engineering-Fakultät,
Operating Systems and Middleware Group

Gutachter: Prof. Dr.-Ing. Jörg Nolte
Brandenburgische Technische Universität Cottbus-Senftenberg,
Fakultät 1 / Institut für Informatik,
Fachgebiet Verteilte Systeme/Betriebssysteme

Prof. Dr.-Ing. Timo Hönig
Ruhr Universität Bochum,
Fakultät für Informatik,
Operating Systems and System Software Group

Datum der Einreichung: 03. Mai 2022
Datum der Disputation: 05. Juli 2022

Published online on the Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-55811
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-558118

Abstract

The heterogeneity of today's state-of-the-art computer architectures is confronting application developers with an immense degree of complexity which results from two major challenges. First, developers need to acquire profound knowledge about the programming models or the interaction models associated with each type of heterogeneous system resource to make efficient use thereof. Second, developers must take into account that heterogeneous system resources always need to exchange data with each other in order to work on a problem together. However, this data exchange is always associated with a certain amount of overhead, which is why the amounts of data exchanged should be kept as low as possible.

This thesis proposes three programming abstractions to lessen the burdens imposed by these major challenges with the goal of making heterogeneous system resources accessible to a wider range of application developers. The *lib842* compression library provides the first method for accessing the compression and decompression facilities of the *NX-842* on-chip compression accelerator available in IBM Power Central Processing Units (CPUs) from user space applications running on Linux. Addressing application development of scale-out Graphics Processing Unit (GPU) workloads, the *CloudCL* framework makes the resources of GPU clusters more accessible by hiding many aspects of distributed computing while enabling application developers to focus on the aspects of the data parallel programming model associated with GPUs. Furthermore, *CloudCL* is augmented with transparent data compression facilities based on the *lib842* library in order to improve the efficiency of data transfers among cluster nodes. The improved data transfer efficiency provided by the integration of transparent data compression yields performance improvements ranging between 1.11× and 2.07× across four data-intensive scale-out GPU workloads. To investigate the impact of programming abstractions for data placement in Non-Uniform Memory Access (NUMA) systems, a comprehensive evaluation of the *PGASUS* framework for NUMA-aware C++ application development is conducted. On a wide range of test systems, the evaluation demonstrates that *PGASUS* does not only improve the developer experience across all workloads, but that it is also capable of outperforming NUMA-agnostic implementations with average performance improvements of 1.56×.

Based on these programming abstractions, this thesis demonstrates that by providing a sufficient degree of abstraction, the accessibility of heterogeneous system resources can be improved for application developers without occluding performance-critical properties of the underlying hardware.

Zusammenfassung

Die Heterogenität heutiger Rechnerarchitekturen konfrontiert Anwendungsentwickler mit einem immensen Maß an Komplexität, welches sich aus zwei großen Herausforderungen ergibt. Erstens müssen Entwickler fundierte Kenntnisse über die Programmiermodelle oder Interaktionsmodelle verfügen, welche eine Voraussetzung sind um die jeweiligen heterogenen Systemressourcen effizient nutzen zu können. Zweitens müssen Entwickler berücksichtigen, dass heterogene Systemressourcen immer auch Daten untereinander austauschen müssen, um ein Problem gemeinsam zu bearbeiten. Dieser Datenaustausch ist aber auch immer mit einem gewissen Mehraufwand verbunden, weshalb die ausgetauschten Datenmengen so gering wie möglich gehalten werden sollten.

Diese Dissertation schlägt drei Programmierabstraktionen vor und ermöglicht es so, Anwendungsentwickler bei der Bewältigung dieser Herausforderungen zu entlasten, so dass heterogene Systemressourcen für eine größere Anzahl von Anwendungsentwicklern zugänglich werden. Die *lib842*-Kompressionsbibliothek bietet Anwendungen erstmals die Möglichkeit, die Kompressions- und Dekompressionsfunktionen des in IBM Power Prozessoren integrierten *NX-842* Kompressionsbeschleunigers unter Linux zu verwenden. Das *CloudCL*-Framework richtet sich an die Entwicklung von GPU-beschleunigten, verteilten Anwendungen und macht die Ressourcen von GPU-Clustern vereinfacht nutzbar, indem es viele Aspekte des verteilten Rechnens ausblendet und es so Anwendungsentwicklern ermöglicht, sich auf die Aspekte des auf GPUs üblichen, datenparallelen Programmiermodells zu konzentrieren. *CloudCL* wurde weitergehend über transparente Datenkompressionsfunktionalität auf Basis der *lib842* Programmbibliothek erweitert, um die Datenübertragungseffizienz zwischen Clusterknoten zu verbessern. Die verbesserte Datentransfereffizienz führt zu Leistungsverbesserungen zwischen 1,11-fach und 2,07-fach bei der Verwendung von vier datenintesiven, verteilten, und GPU-beschleunigten Arbeitslasten.

Um die Auswirkungen von Programmierabstraktionen auf die Datenplatzierung in NUMA-Systemen zu untersuchen, wird eine umfassende Evaluierung des *PGASUS*-Frameworks für NUMA-gewahre C++-Anwendungsentwicklung durchgeführt. Unter Verwendung einer breiten Palette von Testsystemen zeigt die Evaluierung, dass *PGASUS* nicht nur die Entwicklung von NUMA-gewahren Anwendungen erleichtert, sondern auch in der Lage ist, die Leistung von NUMA-agnostischen Implementierungen im Mittel um $1,56\times$ zu übertreffen.

Auf der Grundlage dieser Programmierabstraktionen zeigt diese Dissertation, dass heterogene Systemressourcen durch die Bereitstellung angemessener Abstraktionsmechanismen einfacher von Anwendungsentwicklern erschlossen werden können, ohne dass leistungsrelevante Eigenschaften der zugrunde liegenden Hardware verdeckt werden.

Acknowledgements

This thesis would have never been completed without the support I have received from many people whom I would like to thank. I would like to thank my advisor Andreas Polze, who has guided my work by sharing his advice for countless times. First as the advisor and later as my colleague and friend, Frank Feinbube has sparked my interest in returning to academia in order to pursue the endeavors of earning a doctoral degree. At the beginning of my undertakings, the *SSICLOPS* project, or rather the project partners involved in it, have provided me with a research direction when I was not sure which direction I wanted to pursue. Collaborating with the IBM Germany R&D Lab in Böblingen over the course of the *Hybrid DB* project was an incredible opportunity. Special thanks are due to Wolfgang Maier and his team, who did not only provide valuable feedback, but based on their support, I have been awarded with the *IBM PhD Fellowship Award* in 2017.

Over the years I have spent at the Operating Systems and Middleware Group led by Andreas Polze, I have enjoyed both the collaboration and the numerous coffee talks with the 'inhabitants' of the 'hardware corner', namely Felix Eberhardt, Andreas Grapentin, Sven Köhler, and Lukas Wenzel. I am especially grateful to Sven Köhler and Lukas Wenzel, who both have been a great source of inspiration not only professionally, but also on a personal level. Over time, Bernhard Rabe, Christian Neuhaus, Daniel Richter, Frank Feinbube, Lena Feinbube, Marcel Taeumel, Tobias Pape, and Robert Schmid have helped me advance in my endeavors. At the later stages of my undertakings, the collaboration with Timo Hönig has provided me with several new perspectives and ideas that I hope I will be able to pursue in my postdoctoral life.

Without the support of Sabine Wagner, many bureaucratic acts would have been a nightmare. Also, this work would not have been possible without the FutureSOC Lab. More important than the compute equipment provided by the lab was the relentless support of Bernhard Rabe, Tobias Pape, and Ayleen Oswald.

The biggest thanks are due to my wife Annabell and my children. Without their tremendous loving support, I could never have pulled this through.

Contents

1 Introduction

Over the last few decades, the uninterrupted growth of data quantities accumulating in the age of digitization has been driving an ever-growing demand for compute capacity. Up until the early 2000s, this demand could be easily satisfied based on the performance gains provided by frequency scaling. With frequency scaling however having reached its limits around 2006, computer architectures had to resort to different approaches in order to continuously provide improved compute capacities. Even though the first response to the end of frequency scaling was to invest the steadily increasing transistor count into multicore CPUs, the pressure to innovate created by the end of frequency scaling has promoted the entry of heterogeneous system resources into mainstream computer architectures [146]. As such, most CPU vendors have adopted cache-coherent NUMA architectures to scale multiprocessor systems to dimensions that were not feasible with Uniform Memory Access (UMA) approaches [92, 107]. Similarly, all major GPU microarchitectures have adopted unified shader architectures, paving the way for the utilization of GPUs as general purpose compute resources [39, 5, 116].

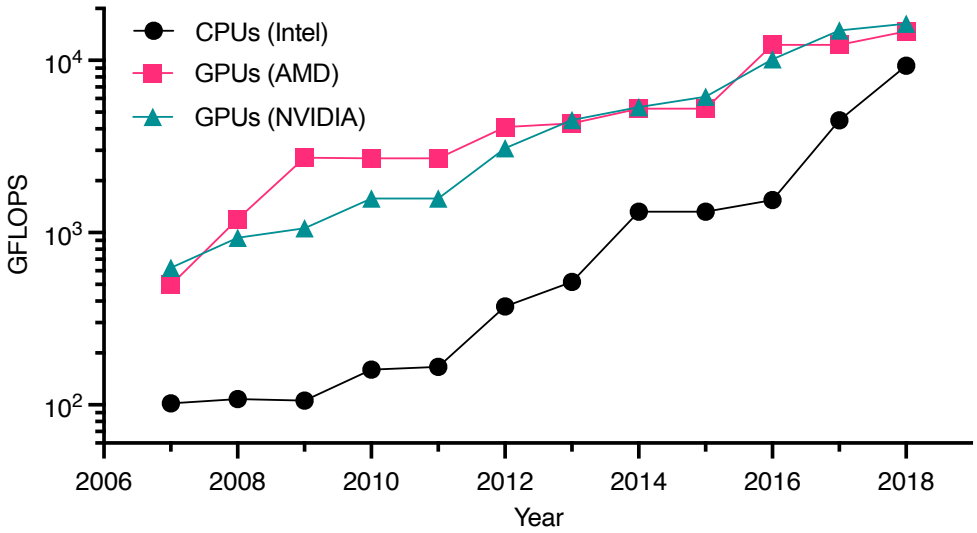

Figure 1.1: The development of single-precision peak performance of both CPUs and GPUs demonstrates that even though CPU performance has caught up based on the introduction of excessively wide Single Instruction Multiple Data (SIMD) extensions, heterogeneous compute resources such as GPUs have delivered sustained growth of compute capacity over the years. The plot has been generated based on the data provided by Karl Rupp [176].

Ever since, heterogeneous system resources have not only become indispensible in providing significant improvements in compute capacity as illustrated in Figure 1.1, but

the degree of heterogeneity in computer architectures has also been steadily increasing over the following years [214]. Upon closer examination, today's state-of-the-art systems ranging from mobile phones to high-end servers are brimmed with heterogeneous system resources such as dedicated inference engines, video compression engines, digital signal processing engines, and many more. A crucial factor driving this trend is the specialization of heterogeneous compute resources offering superior performance per watt ratings compared to the general purpose compute resources of CPUs.

1.1 Heterogeneous System Resources

In the context of computer architectures, heterogeneity is a largely overloaded term. For example, heterogeneous computing may refer to the use of different classes of compute resources (e.g., CPUs and GPUs), the use of CPUs with varying Instruction Set Architectures (ISAs), or the use of CPUs with the same ISA but differences in their microarchitectural properties such as the mixed use of performance-optimized cores and efficiency-optimized cores. Therefore, the goal of this section is to establish a mutual understanding about what types of heterogeneous system resource are considered in the context of this thesis.

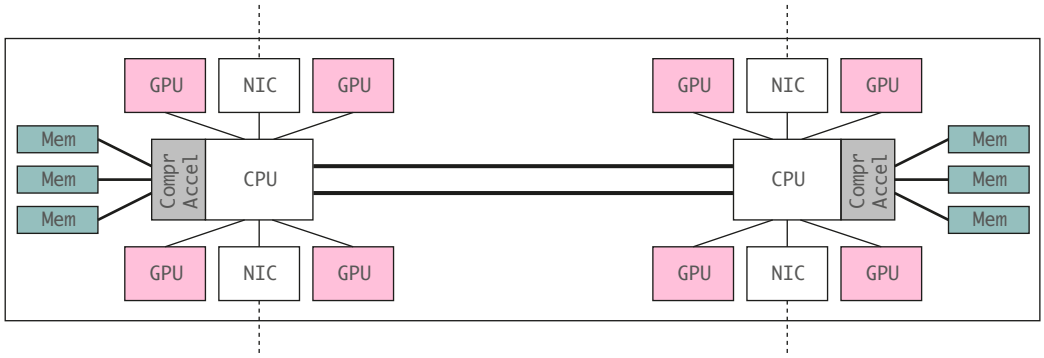

Figure 1.2: In this thesis, programming abstractions for three different types of heterogeneous system resources are presented: *NX842 compression accelerators* as an exemplary instance of *on-chip accelerators* (gray), dedicated GPUs as manifestations of *off-chip accelerators* (magenta), as well as *non-uniform memory resources* (turquoise).

This thesis considers the three types of heterogeneous system resources highlighted in Figure 1.2:

- **On-Chip Accelerators** Many modern CPUs and System on a Chip (SoC) designs are equipped with on-chip accelerators that unlike SIMD ISA extensions are not easily accessible from user-space. While these types of accelerators are tightly integrated into the on-chip communication fabric, interacting with these types of accelerators can be particularly challenging as the methods for invoking their resources may vary even among individual CPU models produced by the same vendor.
- **Off-Chip Accelerators** As the probably most common approach for heterogeneous computing, compute resources of the CPU are augmented with a different type of compute resource such as a GPU which is connected to the CPU using an off-chip

interconnect (e.g., Peripheral Component Interconnect Express (PCIe)). To interact with these kinds of heterogeneous compute resources, their vendors typically provides the necessary infrastructure. In the case of GPUs, both vendor-specific and vendor-agnostic programming frameworks are available.

- **Non-Uniform Memory Resources** Even though the memory resources available in a cache-coherent NUMA system are presented to application developers using a flat address space, both latency and bandwidth available for load and store operations may vary depending on which CPU they are issued and where the data resides. As upcoming technology-agnostic memory interfaces support the mixed use of different memory technologies in a single system (e.g., volatile Synchronous Dynamic Random-Access Memory (SDRAM) for large data quantities, High-Bandwidth Memory (HBM) resources for memory-bound algorithms, and non-volatile memory for persistent data) the heterogeneity of memory resources is only going to intensify from here on.

1.2 Problem Statement

While the ever-growing degree of heterogeneity facilitates continuously increasing compute capacities, it is also the source of two major challenges in dealing with heterogeneous system resources from the perspective of application developers:

1. Many types of heterogeneous system resources require application developers to adapt specific programming models or interaction models to make use of its capabilities. With the large variety of heterogeneous system resources available in state-of-the art computer architectures, application developers are therefore confronted with an immense degree of complexity.

2. Heterogeneous system resources have to exchange data in order to process a workload collectively. Since moving data across heterogeneous system resources can be a performance bottleneck, application developers have to be wary about balancing the use of heterogeneous system resources against the overhead associated with data transfers. However, there are certain scenarios in which prevalent system abstractions may make it hard for application developers to gauge whether their actions may trigger unnecessary data transfers or not.

With the imminent advent of disruptive memory technologies the effects of these challenges are very likely to intensify [136]. It is therefore necessary to provide application developers with programming abstractions that improve the accessibility of heterogeneous system resources without obscuring performance-critical system properties and that help developers to reduce the amount of data that has to be exchanged among heterogeneous system resources.

The goal of this thesis is to address these challenges by proposing programming abstraction approaches for each type of heterogeneous system resource considered in this work (cf. Section 1.1). In the context of this thesis, libraries and frameworks that hide certain complexities of the underlying hardware or the programming models therewith are considered as programming abstractions. For this endeavor, each approach has to factor in the peculiarities of the corresponding resource type:

On-Chip Compression Accelerators Originally intended for the use-case of transparent main-memory compression [17], one goal of this thesis is to enable the use of the *NX-842* on-chip compression accelerator available in IBM POWER CPUs to improve the efficiency of data transfers across heterogeneous system resources. On POWER7+ and POWER8 CPUs, the *NX-842* on-chip compression accelerator is only accessible from kernel-space using the privileged Initiate Coprocessor Store Word Indexed (`icswx`) instruction and is therefore inaccessible from user-space applications. Even though the Virtual Accelerator Switchboard (VAS) facilities have been introduced in the POWER9 microarchitecture with the goal of providing user-space applications access to on-chip accelerator resources such as the *NX-GZIP* compression accelerator, access to the *NX-842* units via the VAS facilities is still restricted to kernel-space unless the *skiboot* firmware is modified correspondingly. Therefore, a programming abstraction is necessary that exposes kernel-space resources to the user-space in order to make the compression facilities of the *NX-842* accessible for user-space applications. The two *NX-842* units available per POWER CPU are tightly integrated and can process up to 36.8 GB/s, exceeding even the throughput of dedicated compression accelerators attached via PCIe 4.0.

Scale-Out GPU Clusters The demand for GPU compute resources has been steadily increasing over the last few years to the point where workloads such as deep learning applications require entire GPU clusters [84] to satisfy their demand for compute resources. Application development for scale-out GPU workloads is very challenging, as developers have to be adept using both data parallel programming models (e.g., Open Computing Language (OpenCL)) and distributed memory parallel programming models (e.g., Message Passing Interface (MPI)). To produce relief, a programming abstraction mechanism is required that enables developers to focus on a single parallel programming model. However, data transfers between CPUs and GPUs can already be a bottleneck in single node scenarios. The limited bandwidth available 10 Gbit/s, 25 Gbit/s, and even 40 Gbit/s Ethernet networks which are still the norm in the vast majority of data centers [19] necessitates that the volume of data transfers is kept minimal in order not to impede the scalability of scale-out GPU workloads.

Non-Uniform Memory Access Systems Even though GPUs have become popular in many data-intensive application domains, many workloads still rely on the flexibility and versatility of multicore CPUs [208]. While several of these CPU-based workloads can be adapted to scale-out across multiple systems to provide sufficient compute resources, certain workloads such as In-Memory Databases (IMDBs) [25] or *de novo genome assembly* [133] are inherently hard to scale out and therefore require as many resources as possible in a single scale-up system. For such workloads, state-of-the-art NUMA systems support

up to 32 multicore CPUs [76] and 64 TiB of main memory [191] while maintaining a single cache-coherent address space. On one hand, making all memory resources in such a scale-up system accessible through the flat address space of the virtual memory abstraction enables application developers to hold on to the shared memory parallel programming model they are familiar with. On the other hand however, the drawback of the virtual memory abstraction is that application developers can only consider performance-critical aspects of NUMA systems using operating system specific Application Programming Interfaces (APIs) such as *libnuma*, with their usage involving several pitfalls on their own. As such, there is a strong need for programming abstraction facilities that enable application developers to easily specify memory placement policies with the goal of exploiting data locality and to avoid unnecessary data transfers across NUMA domains.

1.3 Research Question

The research question of this thesis is concerned with improving the accessibility of heterogeneous system resources for applications developers. A central assumption of the research question is that suitable programming abstractions can help to address two major challenges inherent to heterogeneous system resources: Foremost, this thesis hypothesizes that a certain degree of the complexity conditioned by the large variety of heterogeneous system resources considered in the context of this work can be encapsulated using programming abstractions without obscuring performance-critical system properties. Furthermore, this thesis hypothesizes that these abstractions can help to mitigate the performance penalty associated with data transfers across heterogeneous system resources by either improving data transfer efficiency or by avoiding unnecessary data transfers altogether. For these endeavors, it is of utmost importance to find the right balance between providing a sufficient degree of abstraction on one without burying the heterogeneous system resources under too many layers of abstraction on the other hand.

1.4 Contributions

The author of this thesis, provides several contributions to the field of software engineering. First, the author proposes a programming abstraction mechanism in the form of the *lib842* compression library that facilitates user-space access to the resources of *NX-842* on-chip compression accelerators. To enable interoperability of data compressed using the proprietary *842* compression algorithm with arbitrary CPUs or GPUs, the library also provides the first publicly available user-space facilities for software-based *842* compression and decompression on CPUs, as well as OpenCL-based decompression on GPUs. Both the hardware as well as the software-based compression facilities provide sufficient throughput to saturate 10 Gbit/s, 25 Gbit/s, and even 40 Gbit/s Ethernet networks which are still the norm in the vast majority of data centers [19]. Second, the author introduces the *CloudCL* framework which provides a single-paradigm programming experience for scale-out GPU workloads. By abstracting away many aspects of the distributed memory parallel programming model commonly used in scale-out scenarios, the framework allows application developers to focus on the data parallel programming model associated with

GPUs. To warrant efficient data exchange across compute nodes of a GPU-cluster, the *CloudCL* framework implements transparent on-the-fly data compression based on the *lib842* compression library. Third, the author provides a comprehensive evaluation of the impact of the programming abstractions for NUMA-aware C++ application development provided by the *PGASUS* framework on both developer experience and application performance. To facilitate reproducibility, all software-artifacts presented or used in this thesis in curated in a freely accessible repository.*

1.5 Overview

This thesis is structured as follows. After the introduction, Chapter 2 analyses the state of the art of heterogeneous system resources and presents related work from the field of software engineering. Chapter 3 presents the concept, implementation, as well as the evaluation of the *lib842* compression library which provides user-space access to the hardware-based compression facilities of the *NX-842* on-chip compression accelerator. Focusing on scale-out GPU workloads, Chapter 4 highlights the programming abstraction facilities provided by the *CloudCL* framework that enable application developers and domain experts to focus their efforts on the data parallel programming model associated with GPUs. The improved developer experience offered by *CloudCL* is demonstrated in Chapter 4 by showcasing *CloudCL*-based implementations of two exemplary workloads. To improve the efficiency of data-exchange across compute nodes of a GPU-cluster, Chapter 4 also presents the concept, implementation and evaluation of transparent data compression based on the *lib842* compression library. Chapter 5 provides an overview of the programming abstractions provided by the *PGASUS* framework for NUMA-aware C++ application development, which has been proposed in the master's theses of Wieland Hagen [68] and Karsten Tausche [198]. The improved developer experience facilitated by the *PGASUS* framework is demonstrated in Chapter 5 by comparing *PGASUS*-based implementations of three exemplary workloads to NUMA-aware implementations based on the Open Multi-Processing (OpenMP) interface or the *pthreads* library. As another central aspect of Chapter 5, the performance impact of the *PGASUS* framework is evaluated. Chapter 6 discusses the contributions of the thesis and reflects on potential links for future research before the thesis concludes in Chapter 7.

1.6 Publications

I have already published partial results about many central aspects of the major contributions I am going to present in the following chapters of this thesis. With this section I would like to provide a list of the papers that have directly shaped the contributions I am presenting in this thesis and which have been published in peer-review workshops, conferences, and journals. In my research efforts leading to these publications, I have been supported by the work of many colleagues and master's students, with their support being acknowledged in each publication's respective list of authors.

*https://github.com/plauth/thesis-artifacts

[69] Wieland Hagen, Max Plauth, Felix Eberhardt, Frank Feinbube, and Andreas Polze. "PGASUS: A Framework for C++ Application Development on NUMA architectures". In: *Proceedings of the Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE. Nov. 2016, pages 368–374. DOI: 10.1109/CANDAR.2016.0071

[160] Max Plauth, Florian Rösler, and Andreas Polze. "CloudCL: Distributed Heterogeneous Computing on Cloud Scale". In: *Proceedings of the Fifth International Symposium on Computing and Networking (CANDAR)*. IEEE. Nov. 2017, pages 344–350. DOI: 10.1109/CANDAR.2017.49

[161] Max Plauth, Florian Rösler, and Andreas Polze. "CloudCL: Single-Paradigm Distributed Heterogeneous Computing for Cloud Infrastructures". In: *International Journal of Networking and Computing* 8.2 (July 2018), pages 282–301. ISSN: 2185-2847. DOI: 10.15803/ijnc.8.2_282

[159] Max Plauth and Andreas Polze. "Towards Improving Data Transfer Efficiency for Accelerators Using Hardware Compression". In: *Proceedings of the Sixth International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. Nov. 2018, pages 125–131. DOI: 10.1109/CANDARW.2018.00031

[158] Max Plauth and Andreas Polze. "GPU-Based Decompression for the 842 Algorithm". In: *Proceedings of the Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. Nov. 2019, pages 97–102. DOI: 10.1109/CANDARW.2019.00025

[150] Max Plauth, Joan Bruguera Micó, and Andreas Polze. "Improved Data Transfer Efficiency for Scale-Out Heterogeneous Workloads Using On-the-Fly I/O Link Compression". In: *Concurrency and Computation: Practice and Experience* (Dec. 2020), e6101. DOI: 10.1002/cpe.6101

[152] Max Plauth, Felix Eberhardt, Andreas Grapentin, and Andreas Polze. "Improving the Accessibility of NUMA-Aware C++ Application Development Based on the PGASUS Framework". In: *Concurrency and Computation: Practice and Experience* (Feb. 2022), e6887. DOI: 10.1002/cpe.6887

1.7 Context

Over the years that have led to this work, I had the honor and pleasure to work with many wonderful colleagues in several research projects, collaborations, or teaching activities. Therefore, the goal of this section is to provide a brief overview of the activities that have both accompanied and influenced my work.

1.7.1 Scalable and Secure Infrastructures for Cloud Operations (SSICLOPS)

Funded from the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866, the *SSICLOPS* project lasted from February 2015 to January 2018. As outlined in Figure 1.3, the major goal of the project was to investigate resource management strategies in federated private cloud infrastructures. Operating under the guiding principle "resource management from core to cloud", the work package dealing with workload and data placement strategies in private cloud infrastructures

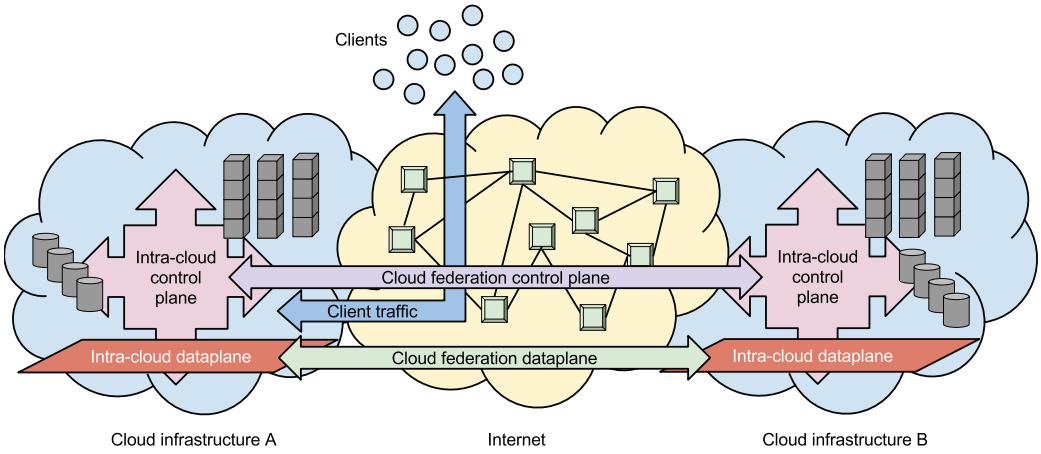

Figure 1.3: The *SSICLOPS* project has investigated federated private cloud infrastructures [180].

turned out to be a productive environment for my colleague Felix Eberhardt and me to investigate workload and data placement strategies on the scope of NUMA systems. In our joint research efforts centered around NUMA systems, we have co-supervised the master's thesis by Wieland Hagen [68], which has yielded the initial version of the *PGASUS* framework. *PGASUS* provides the foundation for my contributions I am going to present in Chapter 5. Beyond the NUMA topic, I have supervised the seminar project by Karsten Tausche [199], in which he investigated the *dOpenCL* API forwarding library to evaluate whether GPU resources can be virtualized on the abstraction level of the OpenCL interface [199]. The successful evaluation yielded by this seminar has served as the basis for the *Dynamic OpenCL* framework presented in the master's thesis by Florian Rösler [175], which again has laid the groundwork for my work on the *CloudCL* framework, which I am going to present in Chapter 4. Together with my colleagues I have undertaken further research efforts in the work package on policy and security as well as the work package on scenario integration. Even though these research efforts do not align with the topics covered in my thesis, they have yielded several joint publications.

1.7.1.1 Project-Related Publications

The joint research efforts I have conducted together both with my colleagues Felix Eberhardt and Stefan Halfpap (Klauck) from HPI as well as our collaboration partners from the *SSICLOPS* project have yielded several peer-reviewed publications. Even though these publications are not necessarily related to my thesis or have only influenced my research efforts indirectly, they are listed hereinafter to highlight the research activities I have conducted alongside my work on this thesis:

[49] Frank Feinbube, Max Plauth, Christian Kieschnick, and Andreas Polze. "Evolving Scheduling Strategies for Multi-Processor Real-Time Systems". In: *Proceedings of the 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. July 2015, pages 57–62. URL: https://www.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-p57.pdf

[12] Jossekin Beilharz, Frank Feinbube, Felix Eberhardt Eberhardt, Max Plauth, and Andreas Polze. "Claud: Coordination, Locality And Universal Distribution". In: *Proceedings of the Parallel Computing Conference 2015 (PARCO)*. Sept. 2015, pages 605–614. DOI: 10.3233/978-1-61499-621-7-605

[151] Max Plauth, Felix Eberhardt, Frank Feinbube, and Andreas Polze. "A Survey of Security-Aware Approaches for Cloud-Based Storage and Processing Technologies". In: *Proceedings of the Third HPI Cloud Symposium "Operating the Cloud"*. Nov. 2015, page 33. DOI: 10.13140/RG.2.2.26664.57604

[154] Max Plauth, Frank Feinbube, Frank Schlegel, and Andreas Polze. "Using Dynamic Parallelism for Fine-Grained, Irregular Workloads: A Case Study of the N-Queens Problem". In: *Proceedings of the Third International Symposium on Computing and Networking (CANDAR)*. IEEE. Dec. 2015, pages 404–407. DOI: 10.1109/CANDAR.2015.26

[156] Max Plauth, Wieland Hagen, Frank Feinbube, Felix Eberhardt, Lena Feinbube, and Andreas Polze. "Parallel Implementation Strategies for Hierarchical Non-uniform Memory Access Systems by Example of the Scale-invariant Feature Transform Algorithm". In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. May 2016, pages 1351–1359. DOI: 10.1109/IPDPSW.2016.47

[153] Max Plauth, Frank Feinbube, Frank Schlegel, and Andreas Polze. "A Performance Evaluation of Dynamic Parallelism for Fine-grained, Irregular Workloads". In: *International Journal of Networking and Computing* 6.2 (July 2016), pages 212–229. ISSN: 2185-2847. DOI: 10.15803/ijnc.6.2_212

[157] Max Plauth and Andreas Polze. "Are Low-Power SoCs Feasible for Heterogenous HPC Workloads?" In: *Proceedings of the European Conference on Parallel Processing*. Springer. Aug. 2016, pages 763–774. DOI: 10.1007/978-3-319-58943-5_61

[199] Karsten Tausche, Max Plauth, and Andreas Polze. "dOpenCL–Evaluation of an API-Forwarding Implementation". In: *Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud"*. Nov. 2016. DOI: 10.13140/RG.2.2.16598.24641

[162] Max Plauth, Christoph Sterz, Felix Eberhardt, Frank Feinbube, and Andreas Polze. "Assessing NUMA Performance Based on Hardware Event Counters". In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. May 2017, pages 904–913. DOI: 10.1109/IPDPSW.2017.51

[96] Maël Kimmerlin, Peer Hasselmeyer, Seppo Heikkilä, Max Plauth, Paweł Parol, and Pasi Sarolahti. "Network Expansion in OpenStack Cloud Federations". In: *2017 European Conference on Networks and Communications (EuCNC)*. June 2017, pages 1–5. DOI: 10.1109/EuCNC.2017.7980655

[155] Max Plauth, Lena Feinbube, and Andreas Polze. "A Performance Survey of Lightweight Virtualization Techniques". In: *Proceedings of the European Conference on Service-Oriented and Cloud Computing*. Springer. Sept. 2017, pages 34–48. DOI: 10.1007/978-3-319-67262-5_3

[97] Maël Kimmerlin, Max Plauth, Seppo Heikkilä, and Tapio Niemi. "A Practical Evaluation of a Network Expansion Mechanism in an OpenStack Cloud Federation". In: *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. 2017, pages 1–6. DOI: 10.1109/CloudNet.2017.8071540

[149] Max Plauth, Matthias Bastian, and Andreas Polze. "Facilitating Policy Adherence in Federated OpenStack Clouds with Minimally Invasive Changes". In: *Proceedings of the Fifth HPI Cloud Symposium "Operating the Cloud"*. Nov. 2017. DOI: 10.13140/RG.2.2.34267.28969

[60] Andreas Grapentin, Max Plauth, and Andreas Polze. "MemSpaces: Evaluating the Tuple Space Paradigm in the Context of Memory-Centric Architectures". In: *Proceedings of the Fifth International Symposium on Computing and Networking (CANDAR)*. IEEE. Nov. 2017, pages 284–290. DOI: 10.1109/CANDAR.2017.55

[77] Jens Hiller, Maël Kimmerlin, Max Plauth, Heikkilä Seppo, Stefan Klauck, Ville Lindfors, Felix Eberhardt, Dariusz Bursztynowski, Jesus Llorente Santos, Oliver Hohlfeld, and Klaus Wehrle. "Giving Customers Control Over Their Data: Integrating a Policy Language into the Cloud". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018, pages 241–249. DOI: 10.1109/IC2E.2018.00050

[163] Max Plauth, Fredrik Teschke, Daniel Richter, and Andreas Polze. "Hardening Application Security using Intel SGX". in: *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. Aug. 2018, pages 375–380. DOI: 10.1109/QRS.2018.00050

[99] Stefan Klauck, Max Plauth, Sven Knebel, Marius Strobl, Douglas Santry, and Lars Eggert. "Eliminating the Bandwidth Bottleneck of Central Query Dispatching Through TCP Connection Hand-Over". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019)*. Edited by Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer. Gesellschaft für Informatik, Bonn, 2019, pages 97–106. DOI: 10.18420/btw2019-07

1.7.1.2 Project-Related Master's Theses

In the context of the *SSICLOPS* project, I have (co-)supervised the following master's theses:

[184] Patrick Schmidt. "Optimization Guidelines for NUMA Architectures". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Jan. 2016. URL: https://osm.hpi.de/bookshelf/Details/533

[68] Wieland Hagen. "A Programming Model for C++ Application Development on Non-Uniform Memory Access Architectures". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2016

[186] Vincent Schwarzer. "Evaluierung von Unikernel-Betriebssystemen für Cloud-Computing". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, June 2016

[192] Christoph Sterz. "Analyzing NUMA Performance Based on Hardware Event Counters". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2016. URL: https://osm.hpi.de/bookshelf/Details/530

[11] Jossekin Beilharz. "Koordinierungssprachen — von NUMA-Knoten bis zu Cloud-Verbünden". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Oct. 2016

[8] Matthias Bastian. "Entwurf und Integration eines Frameworks zur Einhaltung nutzerdefinierter Policies in OpenStack". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Jan. 2017. URL: https://osm.hpi.de/bookshelf/Details/457

[175] Florian Rösler. "Dynamic OpenCL - Distributed Computing on Cloud Scale". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2017. URL: https://osm.hpi.de/bookshelf/Details/460

[203] Fredrick Teschke. "Hardening Applications with Intel SGX". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2017

[102] Sven Knebel. "Interfaces for New Networking Challenges". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, June 2018. URL: https://osm.hpi.de/bookshelf/Details/528

[127] Jan-Henrich Mattfeld. "Design and Implementation of a Unified Middleware for Policy Enforcement in Multi-Cloud Infrastructures". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2018. URL: https://osm.hpi.de/bookshelf/Details/480

1.7.1.3 Project-Related Master's Projects

In the master's programs at HPI, the master's thesis is preceded by a larger project comparable to the scope of a final year project. Over the course of the *SSICLOPS* project, I have (co-)supervised the following master's projects:

[91] Marvin Keller, Philipp Pajak, Florian Rösler, and Robert Schäfer. "Scalable and Secure Infrastructures for Cloud Operations". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Mar. 2016

[126] Fabian Maschler, Jan-Henrich Mattfeld, and Norman Rzepka. "Scalable and Secure Infrastructures for Cloud Operations". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Sept. 2016

1.7.2 Hybrid DB

The *Hybrid DB* project has been conducted in cooperation with *IBM Germany Research and Development* from 2016 through 2021 as a follow-up to preceding research efforts centered around NUMA systems undertaken by my colleague Felix Eberhardt. Hence, a central aspect of the *Hybrid DB* project was the investigation of heterogeneous properties of main memory resources in large scale-up NUMA systems. In this continuation of NUMA-related research, I have supervised the master's thesis by Karsten Tausche [198] in which he has contributed improvements to the *PGASUS* framework. *PGASUS* serves as the

foundation for my contributions I am going to present in Chapter 5. As another central aspect of the *HybridDB* project was the investigation of new approaches for interacting with heterogeneous system resources, the project provided me with an ideal environment for focusing my research efforts on this topic. In this environment, I have co-supervised the master's theses by Lukas Wenzel [209] and Robert Schmid [181], who have explored programming abstractions for leveraging coherently integrated Field-Programmable Gate Array (FPGA) accelerators for data-intensive workloads such as IMDBs leveraging the then unique Coherent Accelerator Interface Architecture (CAIA) facilities introduced with the POWER8 microarchitecture. Their work on data-intensive workloads has sparked my research interests centered around the *NX-842* on-chip compression accelerators, which I am going to introduce in Chapter 3. Building up on top of the work of Karsten Tausche (cf. *dOpenCL* [199]) and Florian Rösler (cf. *Dynamic OpenCL* [175]) conducted within the framework of the *SSICLOPS* project (cf. Section 1.7.1), I have furthermore investigated strategies for virtualizing GPU compute resources in multi-tenant scenarios in the context of the *Hybrid DB* project. For this work, I have been awarded with the *IBM PhD Fellowship Award* in 2017. In these GPU-related research efforts, I have proposed the *CloudCL* framework which I am going to present in Chapter 4.

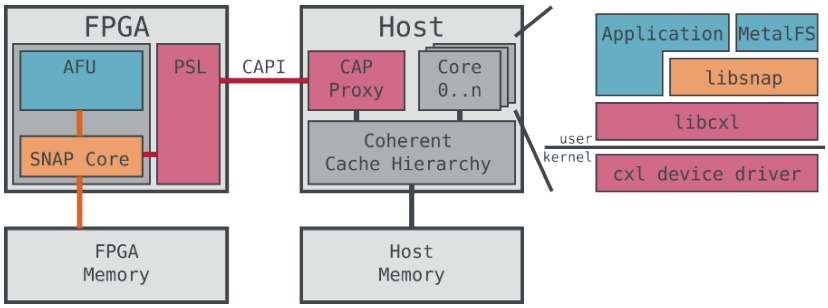

Figure 1.4: Introduced with the POWER8 microarchitecture, the CAIA facilities are comprised of the Coherent Accelerator Processor Proxy (CAPP) on the side of the host CPU and the POWER Service Layer (PSL) on the side of the accelerator, with both components communicating via the Coherent Accelerator Processor Interface (CAPI) protocol. Image source: [209].

1.7.2.1 Project-Related Publications

The research efforts I have conducted together with my colleagues Felix Eberhardt, Robert Schmid, and Lukas Wenzel have yielded several peer-reviewed publications. Even though these publications are not necessarily related to my thesis or have only influenced my research efforts indirectly, they are listed hereinafter to highlight the research activities I have conducted alongside my work on this thesis:

[210] Lukas Wenzel, Robert Schmid, Balthasar Martin, Max Plauth, Felix Eberhardt, and Andreas Polze. "Getting started with CAPI SNAP: Hardware Development for Software Engineers". In: *Euro-Par 2018: Parallel Processing Workshops*. Springer. Aug. 2018, pages 187–198. DOI: 10.1007/978-3-030-10549-5_15

[183] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. "Orchestrating Near-Data FPGA Accelerators Using Unix Pipes". In: *Proceedings of the Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. Nov. 2019, pages 125–128. DOI: 10.1109/CANDARW.2019.00030

[182] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. "Accessible Near-Storage Computing with FPGAs". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, Apr. 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387557

1.7.2.2 Project-Related Master's Theses

In the context of the *Hybrid DB* project, I have (co-)supervised the following master's theses:

[103] Sven Köhler. "On-Chip Accelerators on POWER8". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, May 2017. URL: https://osm.hpi.de/bookshelf/Details/531

[173] Daniel Roeder. "Recording and Profiling Workload Characteristics". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2017

[198] Karsten Tausche. "Memory Management on IBM Power Systems with NUMA Characteristics based on the PGASUS Programming Framework". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Oct. 2017. URL: https://osm.hpi.de/bookshelf/Details/540

[45] Kai Fabian. "Measuring and Interpreting NUMA Main Memory Latencies". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Sept. 2017. URL: https://osm.hpi.de/bookshelf/Details/536

[211] Christian Wuerz. "Resource Contention of Competing Processes in Parallel Systems". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Oct. 2017. URL: https://osm.hpi.de/bookshelf/Details/534

[181] Robert Schmid. "Using FPGA Performance Counters for Profiling Heterogenous Applications". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Dec. 2018. URL: https://osm.hpi.de/bookshelf/Details/535

[209] Lukas Wenzel. "Operating System Facilities for FPGA Accelerator Designs". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, June 2019. URL: https://osm.hpi.de/bookshelf/Details/498

[10] Yannick Bäumer. "Hardware Accelerated Lossless Compression using High-Level Synthesis". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Nov. 2019. URL: https://osm.hpi.de/bookshelf/Details/538

[22] Joan Bruguera Micó. "Improved Data Transfer Efficiency for Scale-Out GPU Workloads using On-the-Fly I/O Link Compression". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2020. URL: https://osm.hpi.de/bookshelf/Details/539

1.7.2.3 Project-Related Master's Projects

In the master's programs at HPI, the master's thesis is preceded by a larger project comparable to the scope of a final year project. Over the course of the *Hybrid DB* project, I have (co-)supervised the following master's projects:

[124] Balthasar Martin, Robert Schmid, and Lukas Wenzel. "CAPI SNAP Development for Programmers". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Sept. 2017. URL: `https://osm.hpi.de/capi-snap`

1.7.3 Memento: Energy-Efficient Memory Placement

Together with my colleagues Sven Köhler and Lukas Wenzel, I have performed preliminary research in collaboration with the *Bochum Operating Systems and System Software* (BOSS) research group at the *Ruhr University Bochum* (RUB) headed by Prof. Dr. Timo Hönig from 2019 through 2021, which has led to the submission of the joint *Memento* proposal to the *Priority Program 2377* on *Disruptive Memory Technologies* by the *German Research Foundation*. Based on an ISA-agnostic workload representation, our initial goal was to improve energy-efficiency in data centers by placing workloads on the resources that are best suited for the respective workload as outlined in Figure 1.5.

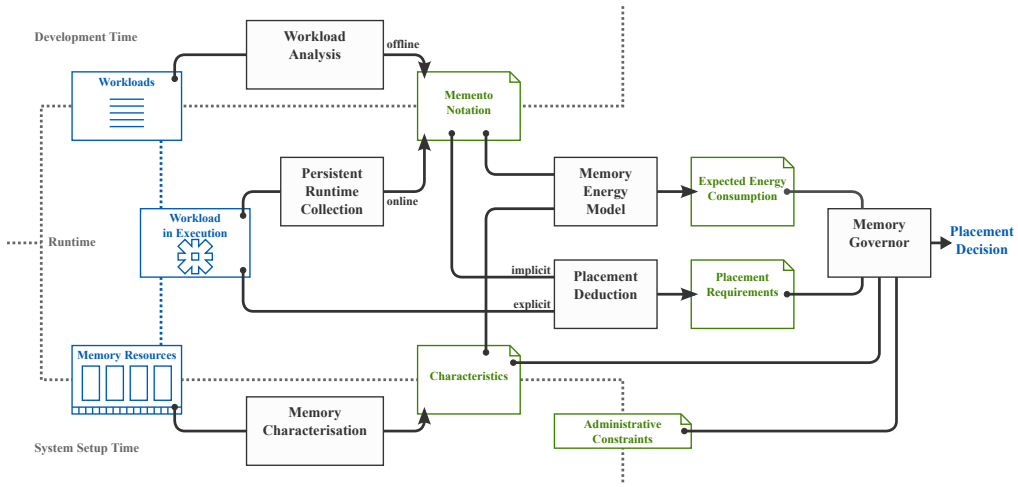

Figure 1.5: The goal of the proposed *Memento* project is to improve energy efficiency based on memory placement strategies that consider the varying characteristics of diverse memory resources available in state-of-the-art computer architectures. Image source: [79].

During the initial phase of the preliminary work, our research efforts were focused on establishing common APIs for measuring the energy demand of workloads across various hardware using the *Pinpoint* utility [104]. However, as the diversification of memory resources has culminated in the approval of the *Priority Program 2377* on *Disruptive Memory Technologies*, we have refocused our research efforts on exploiting the heterogeneity of memory resources with the goal of improving energy efficiency. Even though a funding

decision for *Memento* proposal is still pending at the time of writing, we are hoping to commence the active work on the *Memento* project some time around summer 2022.

1.7.3.1 Project-Related Publications

The joint research efforts I have conducted together both with my colleagues Sven Köhler and Lukas Wenzel from HPI as well as our collaboration partners Timo Hönig and Benedict Herzog from RUB have yielded several peer-reviewed publications. Even though these publications are not necessarily related to my thesis or have only influenced my research efforts indirectly, they are listed hereinafter to highlight the research activities I have conducted alongside my work on this thesis:

[72] Benedict Herzog, Timo Hönig, Wolfgang Schröder-Preikschat, Max Plauth, Sven Köhler, and Andreas Polze. "Bridging the Gap: Energy-efficient Execution of Software Workloads on Heterogeneous Hardware Components". In: *Proceedings of the Tenth ACM International Conference on Future Energy Systems*. June 2019, pages 428–430. DOI: 10.1145/3307772.3330176

[104] Sven Köhler, Benedict Herzog, Timo Hönig, Lukas Wenzel, Max Plauth, Jörg Nolte, Andreas Polze, and Wolfgang Schröder-Preikschat. "Pinpoint the Joules: Unifying Runtime-Support for Energy Measurements on Heterogeneous Systems". In: *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. IEEE. Nov. 2020, pages 31–40. DOI: 10.1109/ROSS51935.2020.00009

[105] Sven Köhler, Lukas Wenzel, Max Plauth, Pawel Böning, Philipp Gampe, Leonard Geier, and Andreas Polze. "Recognizing HPC Workloads Based on Power Draw Signatures". In: *Ninth International Symposium on Computing and Networking (CANDAR)*. Matsue, Japan: IEEE, Nov. 2021, pages 278–284. DOI: 10.1109/CANDARW53999.2021.00053

1.7.3.2 Project-Related Master's Theses

During the preparatory work leading to the submission of the *Memento* proposal, I have (co-)supervised the following master's theses:

[67] Felix Grzelka. "On the Energy Consumption of Deep Learning Workloads". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2021. URL: https://osm.hpi.de/bookshelf/Details/529

1.7.3.3 Project-Related Master's Projects

In the master's programs at HPI, the master's thesis is preceded by a larger project comparable to the scope of a final year project. Over the preparation phase leading to the submission of the *Memento* proposal, I have (co-)supervised the following master's projects:

[14] Lawrence Benson, Fabian Paul, Christian Werling, and Fabian Windheuser. "Real-time Power Monitoring for Heterogenous Data Centers". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Mar. 2019

[18] Pawel Böning, Philipp Gampe, and Leonard Geier. "Power-Based Workload Classification". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Mar. 2021

[62] Erik Griese, Leon Matthes, and Maximilian Stiede. "Save Energy by Monitoring Workload Memory Utilization". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Mar. 2022

1.7.4 Teaching Activities

The research efforts leading to this work were accompanied by various teaching activities specified hereinafter. Especially the lectures on parallel programming and heterogeneous computing are tightly associated with the topics covered in this thesis, whereas the remaining courses are motivated by the research projects highlighted in the preceding sections.

Winter 2015/16	Parallel and Distributed Systems Project Seminar, Master
Summer 2016	IBM Power Systems Block Lecture, Bachelor and Master
Winter 2017/18	File System From Scratch Project Seminar, Master
Winter 2017/18	Embedded Operating Systems Lecture, Master
Winter 2018/19	IBM Power Systems Block Lecture, Bachelor and Master
Summer 2019	Parallel Programming and Heterogeneous Computing Lecture, Master
Winter 2019/20	Energy-Aware Computing in Heterogeneous Data Centers Project Seminar, Master
Summer 2020	Parallel Programming and Heterogeneous Computing Lecture, Master
Summer 2021	Parallel Programming and Heterogeneous Computing Lecture, Master

2 State of the Art and Related Work

This chapter presents the state of the art and related work in the research field of heterogeneous computer architectures. To deepen the appreciation of the heterogeneous system resources considered in this thesis, this chapter starts by elaborating on the origins of using Graphics Processing Unit (GPU) resources for general purpose computations as well as the development of Non-Uniform Memory Access (NUMA) architectures. Afterwards, upcoming trends in heterogeneous computer architectures are discussed to point out upcoming challenges for the use of heterogeneous system resources. Finally, the canonical approaches for interacting with the heterogeneous system resources covered by this thesis are discussed alongside approaches that provide programming abstractions to deepen the understanding of the complexities application developers are confronted with by heterogeneous system resources.

2.1 The Origins of Heterogeneous System Resources

Revisiting the origins of heterogeneous system resources helps to deepen the understanding of today's heterogeneous computer architectures. Therefore, this section begins with elaborating on how GPU resources have gained general purpose computing capabilities. Afterwards, the formation of modern NUMA architectures is revisited to stress the omnipresence of heterogeneous memory resources in most modern server systems.

2.1.1 Non-Uniform Memory Access Architectures

Of the various parallel architectures available in the 1980s, Symmetric Multiprocessing (SMP) and message-passing emerged as the major multiprocessor architectures [71]. Especially for smaller multiprocessor systems, shared memory SMP systems were prevalent, using a bus to interconnect all processors with main memory and I/O resources [71] as illustrated in Figure 2.1. The success of bus-based SMP systems lies in the circumstance, that smaller instances approach the properties of the *Paracomputer* model [106], in which "identical processors (each with a conventional order-code set) share a common memory which they can read simultaneously in a single cycle" ([185]). Using a shared medium to attach the processors to main memory alleviated the introduction of coherent caches based on bus snooping in order to reduce memory access latencies in these Uniform Memory Access (UMA) systems in practice [71]. For a larger number of processors however, the traffic caused by the snooping-based coherence mechanisms results in bus contention, limiting the scalability of this approach to configurations ranging between 4 and 16 processors per system, depending on the system at hands [172, 71].

Aiming towards better scalability, Distributed Shared Memory (DSM) architectures emerged in the late 1970s and early 1980s, giving up cache coherence to overcome the

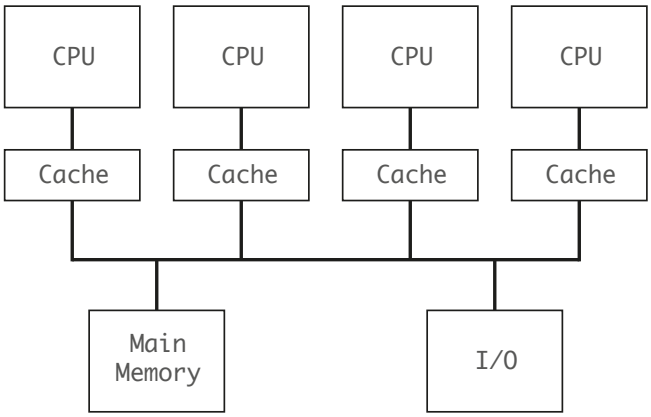

Figure 2.1: In SMP architectures, the bus used to interconnect all processors with main memory and I/O resources can be exploited to implement coherent caches based on bus snooping. However, the shared medium also represents a bottleneck, limiting the scalability of the approach. Figure adapted from [71].

scaling limitations of bus-based SMP systems. Notable DSM multiprocessor approaches include the Carnegie Mellon Cm* [197], the IBM RP3 [147], the *Honeywell Information Systems Italia* (HISI) XPS-100 series [32, 200], and the *Bolt, Baranek, and Newman* (BBN) Butterfly [108, 56]. As depicted in Figure 2.2, each processor in a DSM architecture is equipped with local main memory and I/O resources, and all processors are interconnected using a scalable interconnection network. Even though memory resources are physically distributed in this approach, the shared memory programming model still applies, as all memory resources are accessible through a single shared address space [71]. As a result of the shared address space, each processor can access remote memory resources attached to another processor through the scalable interconnection network using load and store semantics. Without a shared medium as an interconnection network, snooping-based coherency protocols were no longer feasible [71]. From an application developers perspective, DSM systems were much more challenging to develop for, as remote data access operations could not be cached due to the lack of cache coherency, resulting in remote access times that could take 10 times longer compared to local access operations [71]. With varying access times being a performance-critical property of DSM architectures, they are more commonly referred to as NUMA architectures. Due to the vast performance penalty of uncached remote access operations in such NUMA systems, developers had to carefully decide which data should be shared [187].

To combine the scalability of DSM architectures with the application developer productivity of bus-based SMP architectures, directory-based schemes for maintaining cache coherence in large multiprocessor systems have been proposed as a promising approach in 1988 [4]. By augmenting each processor with the local partition of a distributed directory as visualized in Figure 2.3, the coherence state of cache lines can be tracked without congesting the interconnection network. With the Stanford *DASH* multiprocessor, the first Cache Coherent Non-Uniform Memory Access (ccNUMA) multiprocessor architecture has been introduced in 1992 [112]. In the *DASH* prototype, Silicon Graphics Power Station

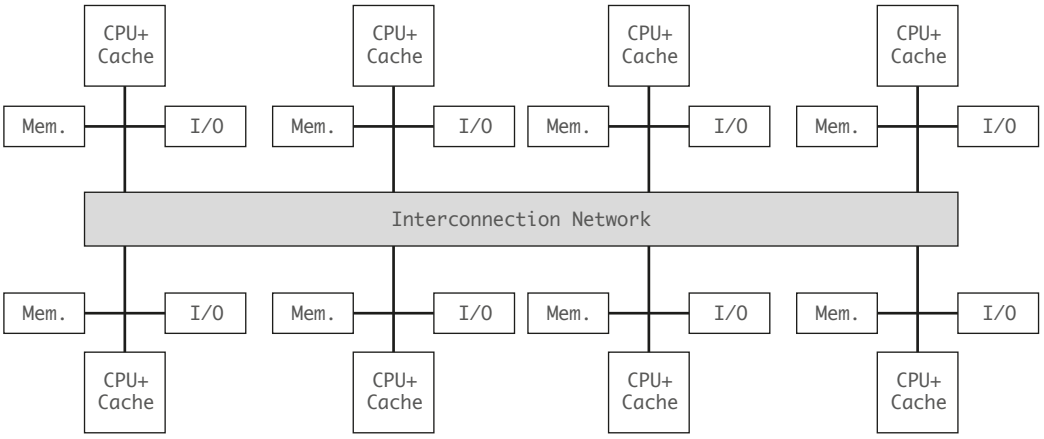

Figure 2.2: DSM architectures avoid the bottleneck of SMP systems by equipping each processor with local main memory and I/O resources at the cost of giving up coherent caches, making remote memory access extremely costly. Figure adapted from [71].

4D/340 4-way SMP systems were used as base clusters. By augmenting the memory bus of a base cluster system with directory memory and an inter-cluster interface, up to 16 base clusters could be interconnected to form a ccNUMA system with up to 64 Central Processing Units (CPUs). Another important contribution of the *DASH* project is that it formalized weak memory consistency models, including release consistency as further means to improve the scalability of multiprocessor architectures [57].

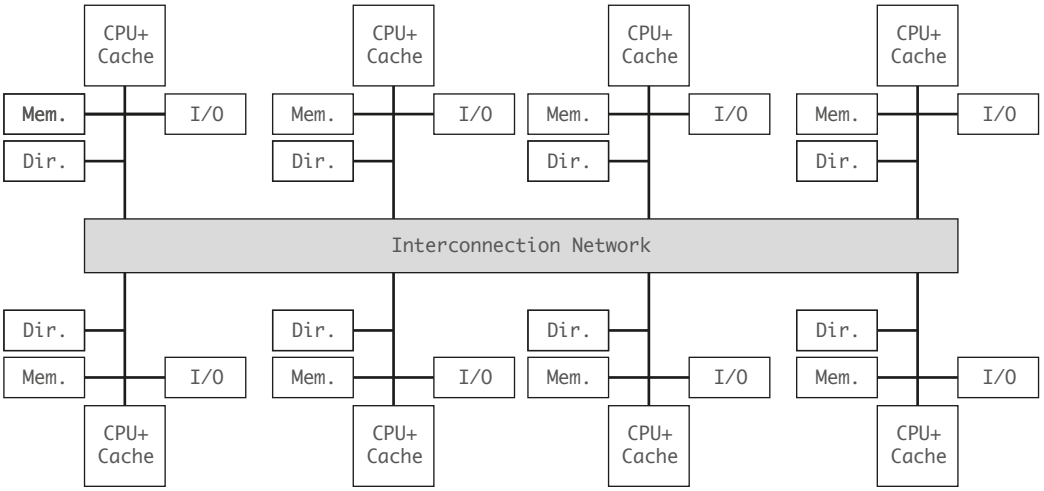

Figure 2.3: By augmenting the local memory resources with directories for tracking the location of cache-lines, ccNUMA systems manage to combine the ease of use of SMP systems with the scalability of DSM architectures. Figure adapted from [71].

Defining a directory-based, coherent high-speed interconnection standard, the approval of the Scalable Coherent Interface (SCI) (IEEE 1596-1992) paved the way for the first wave of commercially available ccNUMA systems. The Convex Exemplar SPP-1000 series intro-

duced in 1994 marks the first commercial ccNUMA system on the market. Based on SCI, the Exemplar SPP-1000 was available in configurations with up to 16 *hypernodes* comprised of 8 HP PA-7100 CPU, each [20]. In 1996, the Sequent NUMA-Q series, internally dubbed *Sequent: The Next Generation (with Intel inside)* (STiNG), has been introduced, which employed SCI to interconnect up to 16 *quads* comprised of 4 Intel Pentium Pro CPUs [119]. A similar approach has been taken with the Data General AViiON AV 2000 series introduced in 1997, which also relied on SCI to interconnect up to 8 *quads* comprised of 4 Intel Pentium Pro CPUs [35]. All three approaches have in common that like the *DASH* prototype, they used conventional, bus-based SMP configurations as building blocks and augmented them with glue logic to interconnect multiple blocks. In this glue-based approach, bus-based snooping to used to implement coherency on the building block level, whereas the directory-based coherence mechanisms of SCI are used to maintain coherence across building blocks.

While the glue-based approaches provided the flexibility that unmodified CPU designs could be used, they came at the cost of increased remote access latencies caused by the two-leveled design. With the Silicon Graphics Origin 2000 series, the first commercially available glueless ccNUMA architecture was introduced in 1997 [110]. Unlike the glue-based approaches, the two CPUs in each node do not form a bus-based SMP cluster, but are directly connected to the interconnection *Hub*. In theory, the employed architecture would have supported a maximum of 1024 CPUs, but customer configurations were only available with up to 128 CPUs, and only one system with 512 CPUs was installed internally at Silicon Graphics.

With a certain delay, all major CPU vendors adopted ccNUMA architectures in their processor designs. IBM followed by introducing a ccNUMA design in their POWER 4 processors in 2002 [202]. In 2003, the *Digital Equipment Corporation* (DEC) adopted a ccNUMA architecture in their Alpha 21364 (EV7) processors [37]. With the introduction of the AMD Opteron series of processors also in 2003, a ccNUMA-based design entered the mass market [92]. Finally, Intel caught up in 2009 by introducing ccNUMA-based designs in their lineup of both x86_64 and Itanium processors with the Nehalem [107] and Tukwila [190] microarchitectures, respectively.

Since ccNUMA approaches have become the predominating manifestation of NUMA architectures since the 1990s, the prefix for indicating the cache coherent variant is commonly omitted and the term NUMA is used to refer to ccNUMA architectures.

2.1.2 GPU Computing

An initial level of programmability has made its entry into professional graphics hardware in the mid 1980s, as programmable graphics architectures have been introduced as an alternative to fixed rendering pipelines [42]. To achieve programmability, early examples such as the Pixar CHAP architecture [113] or the Ikonas platform [42] employed microcodable Single Instruction Multiple Data (SIMD) processors in order to process vertex and pixel data in parallel. In the late 1990s, programmable Multiple Instruction Multiple Data (MIMD) architectures such as PixelFlow [44] or the Silicon Graphics InfiniteReality [132] system became available in the high-end range of professional graphics workstations.

The notion of exploiting GPU hardware for general purpose computations was triggered by the introduction of programmable vertex and fragment shaders in consumer

GPUs. Quickly after the introduction of corresponding hardware such as the NVIDIA GeForce 3 or the ATI Radeon 8500 in 2001, an attempt at offloading matrix multiplications to GPUs has been presented [109]. Even though this first approach was unable to achieve performance improvements compared to CPU-based execution, it marks an important milestone as the first successful attempt to use GPUs for general purpose computations. Only one year later in 2002, another approach has managed to break even with CPU performance, achieving a 3.2× speed-up for GPU-based matrix multiplication using 1500×1500 matrices [205].

Despite the high potential of using GPUs for general purpose computations, widespread adoption of this approach was limited as it was very taxing even for application developers with in-depth knowledge of graphics Application Programming Interfaces (APIs) such as OpenGL to set up compute workflows [23]. To simplify the process, several third party compute libraries started to appear in the mid 2000s, with Brook [23] being one of the most influential examples. However, even with such compute libraries at hands, early generations of consumer GPUs with programmable shaders were still limited in various ways, such as the number of instructions that could be used per shader. On an architectural level, the most limiting factor was that fixed pipelines with separate vertex and pixel processors were used, which limited their flexibility for graphics workloads as well as for general purpose computations.

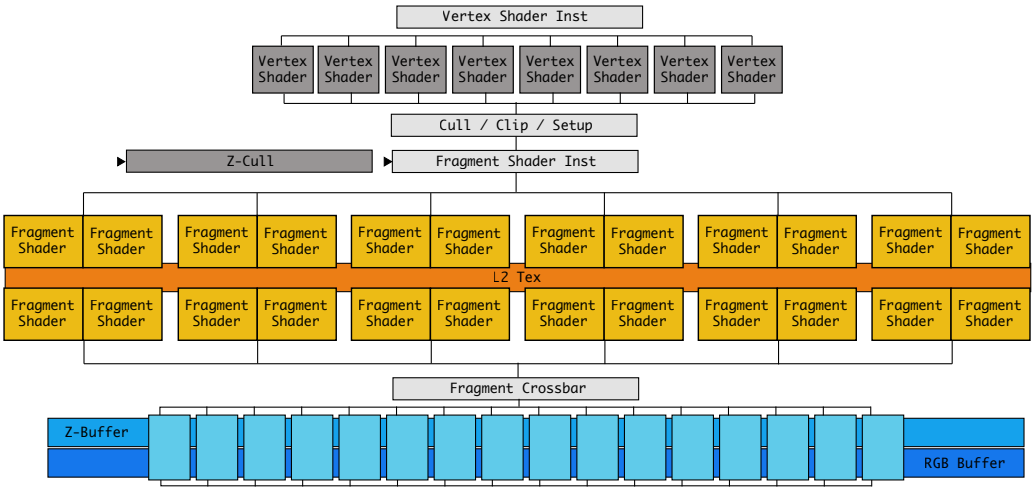

Figure 2.4: Earlier generations of GPUs equipped with programmable shaders suffered from the limitation that each stage of the graphics pipeline required dedicated hardware, making it hard for vendors to decide how much of their chip area they want to spent on what types of resources to provide decent performance across a wide range of applications. Image source: [178].

As the first approach to replace fixed pipeline designs with unified shader architectures, the TerraScale microarchitecture by ATI was introduced in the form of the Xenos GPU employed in the Microsoft Xbox 360 gaming console [39, 5]. The TerraScale microarchitecture employed a SIMD architecture based on Very Long Instruction Word (VLIW) characteristics with the goal of maximizing Instruction-Level Parallelism (ILP), whereas the Tesla microarchitecture presented by competitor NVIDIA in 2006 used a SIMD architecture based on Reduced Instruction Set Computer (RISC) characteristics, which exploits

Thread-Level Parallelism (TLP) [116]. While graphics workloads were able to benefit from unified shader architectures, the novel architecture had an even larger impact on compute tasks and paved the way for a widespread adoption of employing GPUs as general purpose compute accelerators. In 2007, NVIDIA released the initial version of the Compute Unified Device Architecture (CUDA) framework [98] to officially support and promote the use NVIDIA GPUs for general purpose computations. With similar capabilities, the Khronos group has released the initial version of the OpenCL specification [137] as a vendor-independent alternative to CUDA, also including support for other compute resources such as CPUs, Digital Signal Processors (DSPs), Field-Programmable Gate Arrays (FPGAs), and others.

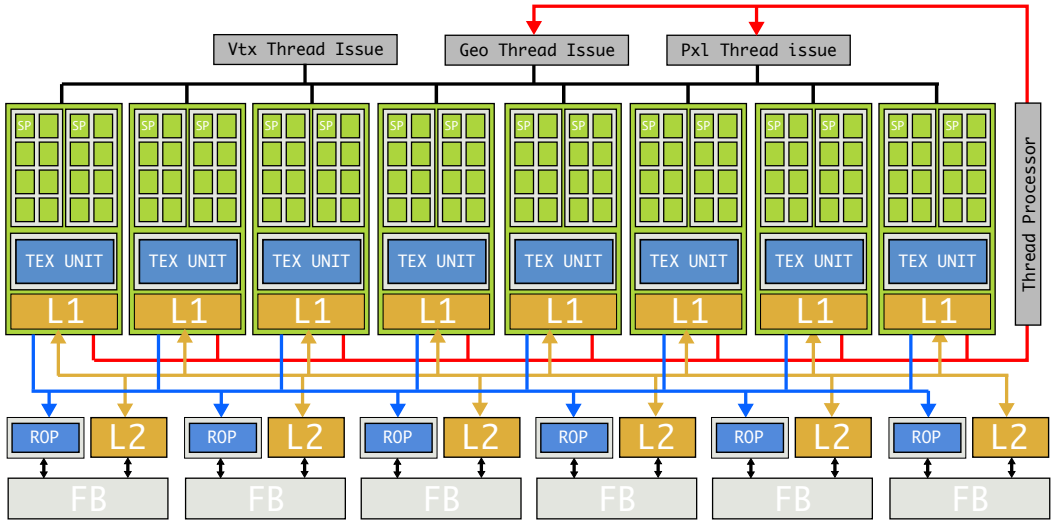

Figure 2.5: In unified GPU architectures, the special-purpose resources employed in fixed-pipeline GPU designs are replaced with generic compute resources that can be used to process most stages of a graphics pipeline, greatly increasing the flexibility of GPUs. Image source: [178].

Over the course of the 2010s, all major GPU microarchitectures adopted SIMD compute resources with RISC characteristics. Not only dedicated GPUs improved continuously, but also integrated GPUs advanced significantly as GPU and CPU resources reached a much tighter level of integration, sharing the same memory hierarchy not only physically but also on a logical level [165]. With the introduction of the *IBM Power System AC922* in 2018, the combination of IBM POWER9 CPUs, NVIDIA V100 GPUs and the cache-coherent NVLink 2.0 interconnection technology have yielded the first system setup to achieve cache-line level integration of dedicated GPUs into the main memory hierarchy of the host CPUs [171].

2.2 Trends in Heterogeneous System Resources

At the time of writing, innovations in the area of interconnection standards are driving extensive transformations in heterogeneous computer architectures. Most notably, the memory hierarchy is about to become more diverse and an increasing number of accelerators such as GPUs, FPGAs, or other domain-specific accelerators are gaining fine-grained, coherent integration into the host system. In this section, the major aspiring interconnection standards are reviewed in Section 2.2.1 and the importance of disruptive memory technologies are outlined in Section 2.2.2.

2.2.1 Coherent Interconnects

Introduced in 2003, the Peripheral Component Interconnect Express (PCIe) standard has served as the standard intra-node interconnect technology for almost two decades. As illustrated in Figure 2.6, frequent releases of new revisions of the PCIe standard roughly every 3.5 years have yielded continuous bandwidth improvements on par with the progression of memory bandwidth available per CPU socket throughout the first three revisions of the standard [63, 64, 65]. After the release of PCIe 3.0 however, the pace of development has slowed down significantly, as the first commercially available product to support the succeeding PCIe 4.0 standard was released seven years later in 2017 with the introduction of the IBM POWER9 CPU [177, 66]. Unfortunately, widespread adoption of the newer standard has only started with the introduction of the AMD Zen 2 CPU microarchitecture in 2019 [196], but even at the time of writing not all major CPU vendors have adopted the PCIe 4.0 standard.

Over the extensive lifespan of PCIe 3.0 as the dominating standard, continuous improvements in the performance of CPUs and accelerators have been accompanied by proportionate improvements of their respective memory subsystems. Due to the overdue progression of interconnect technology, the severe gap between the capabilities of interconnects and the performance of CPUs and accelerators as well as their respective memory subsystems have created a strong need for innovation in the field of interconnect technologies. Consequently, several consortia have formed to bring forward a new generation of interconnection standards such as Open Coherent Accelerator Processor Interface (OpenCAPI) [195], Compute Express Link (CXL) [33], Cache Coherent Interconnect for Accelerators (CCIX) [27], and Gen-Z. Differing in technical details, all of these approaches have in common that they do not only facilitate improved bandwidth and latency, but they also introduce new features such as coherent integration of devices into the main memory hierarchy, as well as serial facilities for attaching memory. While OpenCAPI, CXL, and CCIX are intended for short-reach, intra-node connectivity, the Gen-Z specification has been drafted with rack-scale connectivity in mind. In addition to vendor-independent technologies, proprietary approaches such as NVIDIA NVLink have been introduced.

2.2.2 Disruptive Memory Technologies

Over the last decades, most computer architectures have employed homogeneous memory resources on each layer of the memory hierarchy. In these systems, the properties of memory resources were sufficiently similar to conceal them behind the flat address

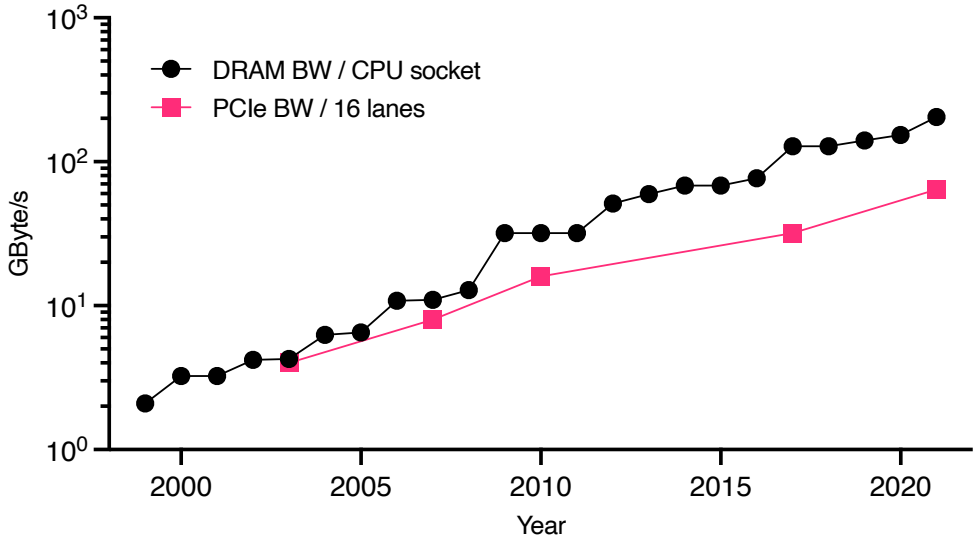

Figure 2.6: Over the years, a certain gap has emerged between the development of Dynamic Random Access Memory (DRAM) bandwidth compared to the bandwidth provided by the PCIe standard. The pressure to innovate created by this discrepancy has fueled a competition for next-generation interconnection technologies. The figure has been adapted from [78] and augmented with additional data from [176].

space of the virtual memory abstraction. However, the recent introduction of various novel memory technologies has initiated a fundamental shift in the design of computer architectures towards supporting heterogeneous memory resources on various layers of the memory hierarchy. The near end of the memory hierarchy is extended with large, potentially self-managed caches based on stacked Static Random-Access Memory (SRAM) or on-package High-Bandwidth Memory (HBM) memory [212, 15]. One step further out in the memory hierarchy, volatile DRAM resources are augmented with Graphics Double Data Rate (GDDR) memory or HBM resources for memory-bound algorithms and non-volatile memory for persistent data, either exclusively [213] or in combination with conventional Double Data Rate (DDR) memory resources. Originating from the respective ecosystems of the OpenCAPI and CXL interconnection standards, the mixed used of different memory technologies is enabled based on the introduction of technology-agnostic memory interfaces [191]. At the far end of the memory hierarchy, the new interconnection technologies discussed in Section 2.2.1 also provide the foundation for disaggregated memory resources, which have the potential to offer enormous memory capacities [148]. The diversity of employable memory technologies implies a disruptive degree of heterogeneity regarding the characteristics of memory resources, which have to be considered.

2.3 Programming Abstractions for Heterogeneous System Resources

The prevalence of heterogeneous system resources has implied a wide range of programming abstractions that have been presented over the last decade. To put the programming abstractions proposed in this thesis in perspective with the latest insights from the field, this section provides an overview of programming abstractions and related approaches for the heterogeneous system resources considered in this work.

2.3.1 Memory Compression and Compressed Data Transfers

Using compression as means for improving utilization of main memory has a well-established history. This is well reflected by the work of Mittal et al. [131], which provides a comprehensive survey of the widely researched field of data compression mechanisms for CPU-based main memory and cache resources. Regarding memory compression on GPUs, a number of approaches have been published as well. Targeting memory-bound applications on GPUs, Sathish et al. [179] have proposed using hardware-based compression to increase the efficiency of access to off-chip device memory, yielding up to 37% better performance compared to the uncompressed case. A similar approach has been published by Vijaykumar et al. [207], who are also employing memory and register compression to increase the utilization of all GPU resources, yielding up to 2.6× speed-up across a variety of memory-bound applications. Following the same goal, Lu et al. [120] have recently proposed a low-latency, hardware-based compression architecture optimized for floating point data that reduces bandwidth demand and energy consumption of GPUs by 44.46% and 44.34%, respectively. Focusing entirely on the register level, Lee et al. [111] have explored register compression on GPUs with the goal of reducing the energy consumption of graphics hardware. All approaches have in common that they are using custom, domain-specific compression algorithms instead of general-purpose compression algorithms and rely on custom hardware designs. With the introduction of their latest A100 GPUs, NVIDIA has introduced hardware support for device-sided memory compression with the *Compute Data Compression* feature [143], promising up to 4× improvements in effective DRAM and L2 bandwidth.

Both CPU-based and GPU-base approaches for memory compression can improve both effective bandwidth and capacity of memory resources significantly. Regrettably, all approaches presented thus far only consider the isolated scope of the memory resources attached to either a CPU or a GPU only and do not consider the scenario of exchanging data across devices in compressed form. Patel et al. [145] have explored the feasibility of on-the-fly data compression for data transfers between CPU and GPU using a generic compression algorithm. The authors conclude that on-the-fly data compression is not feasible using their software-based compression approach implemented on the CPU. Using a data-specific compression algorithm however, Kaczmarski et al. [86] have successfully demonstrated that on-the-fly data compression can be used to speed-up transfers between main memory and GPU memory for GPU-based In-Memory Database (IMDB) use cases. Khavari Tavana et al. [93] have also investigated GPU-based compression approaches using compression algorithms tailored to the characteristics of floating point data. In

contrast to improving transfer efficiency between CPU and GPU, they are using on-the-fly data compression to improve data transfers among multiple GPUs.

2.3.2 General Purpose Computing on GPUs

When the concept of performing general-purpose computations on GPUs emerged, in-depth knowledge of graphics APIs were necessary to set up compute workflows based on fragment shaders. Considering these origins, fundamental frameworks for GPU computing Open Computing Language (OpenCL) and CUDA provide a decent level of abstraction. However, further approaches have been presented with the goal of abstracting the use of GPUs both on the single-node level and in scale-out deployments. The goal of this section is to provide a brief overview of abstractions for GPU compute resources ranging from the basics at the level of fundamental compute frameworks such as OpenCL to high-level abstractions concerned with scale-out GPUs resources.

2.3.2.1 OpenCL

The OpenCL standard [204] defines a framework for executing parallel compute kernels on heterogeneous compute resources available in a single system such as CPUs, GPUs, FPGAs, DSPs, and possibly even further device types. At a conceptual level, the OpenCL standard is built around four models: a *platform model*, an *execution model*, a *memory model*, and a *programming model*.

The *platform model* describes the basic hierarchy of a *host* that is equipped with an arbitrary number of *compute devices*, which are comprised of *compute units* as an organizational structure for grouping individual *processing elements*. An OpenCL implementation must provide a *platform* in the form of an Installable Client Driver (ICD), which enables the *host* to control *compute devices* belonging to the *platform*. Multiple platforms may even coexist on a single host, as multiple ICDs may be available on a system (e.g., for using heterogeneous compute resources from diverse vendors).

The *execution model* distinguishes between the *host* application and *device kernels*. To provide an organizational structure for managing kernel instantiations, the *execution model* defines an *index range* that comprises a hierarchy of *work-items* and *work-groups* in order to specify what data individual *kernel* instances operate on. A *work-item* represents the work performed by a single kernel instance, whereas *work-groups* provide an organizational unit for grouping multiple *work-items*. In a *context*, the *host* application defines *devices*, *kernels*, *program* objects, and *memory* objects. Furthermore, the *host* application is responsible for managing *queues*, which are used to define a sequence of *kernel* executions, operations on *memory* objects (e.g., transfers between *host* and *device* memory), as well as synchronization points.

The *memory model* is tightly complected with the peculiarities of the *platform model* and the *execution model* and defines different types of memory that vary in their scope of accessibility as well as their performance characteristics. Main memory resources of the *host* system are referred to as *host memory*. On the side of compute devices, the memory hierarchy is composed of *global memory*, *constant memory*, *local memory*, as well as *private memory*. The *global memory* and *constant memory* resources comprised by a *context* are typically served by off-chip GDDR memory or HBM and can be accessed by both the *host*

and *kernels*. In contrast to that, *local memory* is often backed by on-chip SRAM and is only shared among the *work-items* belonging to a single *work-group*. *Private memory* refers to memory resources at the register level and is therefore only accessible from the *work-item* it belongs to.

With OpenCL supporting a wide range of heterogeneous compute resources, the employed *programming model* is flexible enough to support both a *data parallel* programming model and a *task parallel* programming model. For the *data parallel* case, a strict one-to-one mapping between a few data elements and *work-items* is used, whereas the *task parallel* case can be achieved by defining an index-range with a single *work-item* per task. OpenCL kernels are implemented using OpenCL C, which is a superset of C99, containing additional keywords that are used specify the employed memory resources defined by the *memory model* as well as synchronization mechanisms. The *host* application is usually implemented in C or C++, however other language bindings exist as well.

Listing 2.1 presents a minimal OpenCL *kernel* implementing the addition of two vectors. Each parameter is a pointer to an array of float values residing in the *global memory* of the heterogeneous compute resource. As the kernel is instantiated for every *work-item*, each kernel instance identifies its *work-item* by querying its index using the in-built function *get_global_id*.

Listing 2.1: Examplary OpenCL vector addition kernel.

```
__kernel void vec_add(__global float *a, __global float *b, __global float *c) {
  int i = get_global_id(0);
  c[i] = a[i] + b[i];
}
```

While the simplistic vector kernel amounts to very few lines of kernel code, OpenCL is notorious for its verbosity regarding the host code necessary to choose an appropriate device, initiate data transfers, execute the kernel, and perform other auxiliary tasks. Even though the host code exemplified in Listing 2.2 uses the C++ bindings of OpenCL which are much less verbose than the native C API, roughly 30 lines of code are necessary to execute the `vec_add` kernel. Using the native C API, the same application requires roughly 50 lines of code.

Listing 2.2: Minimal C++ OpenCL host application necessary to execute the `vec_add` kernel.

```
int main(){
  // Initialize arrays on host
  float array_a[10] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
  float array_b[10] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
  float* array_result = new float[10];
  size_t size = 10 * sizeof(float);

  // OpenCL initialization
  cl::Platform platform;
  cl::Platform::get(&platform);

  std::vector<cl::Device> devices;
  platform.getDevices(CL_DEVICE_TYPE_GPU, &devices);
  cl::Context context(devices);
```

```
15   cl::Program program(context, OPENCL_PROGRAM);
16   program.build(devices);
17   cl::Kernel kernel(program, "vec_add");
18
19   cl::CommandQueue queue(context, devices[0]);
20   cl::Buffer d_a(context, CL_MEM_READ_ONLY, size);
21   cl::Buffer d_b(context, CL_MEM_READ_ONLY, size);
22   cl::Buffer d_c(context, CL_MEM_WRITE_ONLY, size);
23
24   // data transfers & kernel invocation
25   queue.enqueueWriteBuffer(d_a, CL_FALSE, 0, size, array_a);
26   queue.enqueueWriteBuffer(d_b, CL_FALSE, 0, size, array_b);
27   queue.enqueueNDRangeKernel(kernel, cl::NullRange, size, cl::NullRange);
28   queue.enqueueReadBuffer(d_c, CL_TRUE, 0, size, array_result);
29 }
```

2.3.2.2 Single-Node Wrappers for OpenCL

For many application developers and domain experts not well versed in the C or C++ programming languages, the verbosity of OpenCL can be overwhelming. Hence, a wide range of OpenCL wrappers have been presented for various higher-level programming languages such as Java, C#, and Python. As one of the best known OpenCL wrappers, the *Aparapi* framework makes the resources of OpenCL-compatible compute devices available in Java. *Aparapi* is not just a simple wrapper and aims for a transparent integration of heterogeneous compute resources, completely abstracting away device handling tasks and data transfers. Inspired by *Aparapi*, the *Hybrid.Parallel* project attempts to go even one step further by implementing an OpenCL-backed drop-in replacement for the `Parallel.For` construct of the *Task Parallel Library* (TPL) in the .NET ecosystem [50, 48]. In contrast to these tightly integrated approaches, approaches such as *PyOpenCL* offer basic OpenCL wrappers [101].

2.3.2.3 Scale-Out Extensions for OpenCL

Scaling out GPU workloads across multiple compute nodes of a cluster is a recurring issue which has been investigated in a wide range of publications. The canonical approach for distributing GPU workloads relies on mixing GPU compute frameworks such as OpenCL or CUDA with implementations of the Message Passing Interface (MPI) standard. However, this approach requires application developers to deal with multiple levels of abstraction, using different programming models and synchronization semantics.

A commonly employed approach for achieving a uniform developer experience is to present devices scattered across compute nodes as if they were local devices. Relying on means of API forwarding, this strategy exploits the mechanisms available in OpenCL for coordinating work across multiple devices and has been applied by various approaches such as *dOpenCL* [90], *SnuCL* [95], or *VirtualCL* [7]. However, these approaches require developers to partition and schedule their workloads across multiple, potentially heterogeneous devices, increasing the complexity of applications significantly.

To address the problem of increased application complexity, *DistCL* [38] as well as another approach [94] have proposed to fuse multiple compute devices into a single

logical device. *DistCL* creates the illusion of a single, logical device by splitting kernels and the data they operate on it into multiple sub-ranges. To make this automatic splitting process possible, application developers have to supply a meta-function that specifies the memory access pattern. Based on the provided meta-functions, *DistCL* can identify data relevant for a sub-range and transfers it to the corresponding heterogeneous compute resource.

2.3.3 Data Placement in NUMA Architectures

The variety of approaches available for controlling data placement in NUMA systems ranges from approaches that assume a shared memory parallel programming model on one end to approaches that assume a distributed memory parallel programming model on the other end. Both strategies are valid, as the DSM architecture employed by NUMA systems exhibits certain characteristics of a distributed system, all while the cache coherent address space spanning across all NUMA nodes facilitates the system behavior of a shared memory system. Situated somewhere in between, several approaches employ the Partitioned Global Address Space (PGAS) model which enables the differentiation between local and remote memory resources. Hereinafter, popular approaches from all three categories are highlighted.

2.3.3.1 Shared Memory Model

Since many software developers are highly accustomed to the shared memory parallel programming model, the idea of extending this environment with means for controlling data placements on NUMA stands to reason. This section provides an overview of some of the most important approaches that have attempted to achieve that very same goal.

OpenMP The Open Multi-Processing (OpenMP) standard specifies compiler-based extensions for the C/C++ and Fortran programming languages [144]. Based on annotations, OpenMP enables application developers to instruct the compiler to define tasks that can be executed concurrently using a fork-join workflow. The OpenMP runtime library is responsible for mapping tasks onto the threading and synchronization primitives provided by the operating system. As OpenMP has no concept of memory locality, developers have to consider data placement themselves. Even though several approaches have been presented that extend OpenMP with task-to-data associations and a locality-aware scheduling policies [21, 135], none of these approaches have gained traction. As a result thereof, the canonical methods for making OpenMP-based applications NUMA-aware are to rely on the first touch memory allocation policy of the operating system or to use the *libnuma* API [100] to manually manage the placement of memory resources. Both approaches are cumbersome to use as they result in cluttered code that harder to maintain.

Threading Building Blocks The Threading Building Blocks (TBB) C++ template library [169] provides a framework for task and data parallelism for shared memory systems. Based on a parallel tasking infrastructure, synchronization primitives, atomic operations and concurrent data-structures, the library enables application developers to implement

parallel algorithms. Similar to OpenMP, the framework does not consider NUMA-characteristics on its own. Building up on top of TBB, Majo et al. [122] have presented *TBB-NUMA*, which extends the TBB scheduler with task affinities that can be specified manually or automatically by annotating parallel constructs with distribution templates. These task affinities are then considered by the scheduler.

Polymorphic Allocators Standardized in C++-17, *Polymorphic Allocators* [70] can be constructed using specific memory pools which can be used to represent different NUMA domains or different memory characteristics (e.g., volatility, latency, or bandwidth). By serving allocations from the memory pool they have been constructed upon, *polymorphic allocators* enable developers to enforce specific memory placement policies of the active scope. Unfortunately, *polymorphic allocators* cannot be used to modify data placement policies of allocations performed by nested data-structures transparently as polymorphic allocators have to be used explicitly. The explicit use of *polymorphic allocators* makes it necessary to modify nested data-structures in order to pass allocator objects to the nested allocations.

AutoNUMA With memory management being a central aspect of operating systems, it makes sense to facilitate NUMA-awareness through means of operating system facilities. Since version 3.8, the Linux Kernel implements a transparent mechanism called *AutoNUMA* [34]. The mechanism provides two strategies: *memory-follow-cpu* and *cpu-follow-memory*. The former approach unmaps the process pages in regular intervals and tracks the NUMA-node of the CPU which has triggered the page-fault. Based on that information, pages are migrated to the identified NUMA-node to facilitate data locality. The latter strategy uses fault statistics to migrate tasks to CPU cores residing on the same NUMA-node where most of the memory resides. These approaches work well assuming that data-structures are page-aligned. However, issues arise when data-structures are scattered among pages or if they are placed alongside other data-structures that are accessed from CPU cores residing on different NUMA-nodes.

2.3.3.2 Distributed Memory Model

As the de-facto standard for implementing scale-out applications based on the distributed memory parallel programming model [167, 89], the MPI standard [134] defines an extensive list of message-based communication patterns, including point-to-point and group communication as well as reductions. Based on the Single Program Multiple Data (SPMD) paradigm, the MPI execution model assumes one application such as the one exemplified in Listing 2.3 that is deployed on all compute nodes participating in the computation. In a NUMA-agnostic scenario, one process is instantiated per compute node. Each process is identified by a unique identifier, and processes exchange messages to coordinate their work.

An MPI application does not necessarily have to be executed on a set of distinct cluster nodes, but can also launch multiple MPI processes on a single UMA or NUMA system. Since MPI applications are typically designed with the goal of minimizing communication among processes, they scale very well when deployed on a NUMA system [156]. However,

using MPI to implement NUMA-awareness on a shared memory system does not allow application developers to exploit the strengths of large scale-up NUMA systems.

Listing 2.3: Simple message passing example implemented in C using MPI.

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char **argv){
  char msgBuffer[64];
  int rank, processCount;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &processCount);

  if(rank == 0) {
    for(int i = 1; i<processCount;i++) {
      sprintf(msgBuffer, "This is a message from Process 0.");
      MPI_Send(msgBuffer, sizeof(msgBuffer), MPI_CHAR, i, 0, MPI_COMM_WORLD);
    }
  } else {
    MPI_Recv(msgBuffer, sizeof(msgBuffer), MPI_CHAR, 0, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    printf("Process %d receives: %s\n", rank, msgBuffer);
  }

  MPI_Finalize();
  return 0;
}
```

2.3.3.3 Partitioned Global Address Space Model

The PGAS programming model has been designed for data-parallel workloads, maintaining a global address space regardless of the underlying system architecture. The PGAS model has been employed by a multitude of programming languages, libraries and extensions such as X10 [29], Chapel [28], Legion [9], Sequoia [46], Unified Parallel C (UPC) [41], or High Performance Fortran (HPF) [170]. A central aspect of the PGAS model is the distinction between local and remote memory resources, which would make it a perfect fit for NUMA systems. However, even though the PGAS model could be applied to both shared memory architectures and distributed memory architectures, the latter case is much more common and the shared memory level is usually only considered in order to improve the performance of node-local inter-process communication [16].

2.4 Summary

Each type of heterogeneous system resource considered in this thesis has widely varying characteristics, which is well reflected in the way the approaches for providing abstractions vary from one type of heterogeneous system resource to another. While main memory

compression in itself has been researched extensively [131], the different implementations and prototypes available vary on a per-product basis, rendering generic abstractions unreasonable. In the field of GPU computing, many approaches exist for alleviating access to GPUs. However, many approaches offer a similar range of functionality, often times neglecting challenges of distributed computing such as the potential overhead of inter-node data transfers. For NUMA systems, the APIs necessary for controlling data placement are available but very difficult to use correctly. This has resulted in many application developers either hoping for automatic approaches such as *AutoNUMA* to yield decent performance, or to ignore the coherent shared memory view available by treating NUMA systems as distributed systems which is a common strategy in the High-Performance Computing (HPC) community.

3 Programming Abstractions for On-Chip Hardware Compression Resources

This chapter lays out the foundation for investigating the impact of on-the-fly data compression for data transfers across Central Processing Units (CPUs) and Graphics Processing Units (GPUs) in scale-out GPU clusters in Chapter 4. For this purpose, the current chapter is focused on presenting the *lib842* compression library, which provides user-space access to the high-throughput compression facilities of the *NX-842* on-chip compression accelerator available in all IBM POWER processors introduced since the POWER7+ [17]. With the proprietary *842* algorithm [54] employed by the *NX-842* accelerator lacking any software-based implementations available for user-space applications, the *lib842* compression library introduces software-based high-throughput implementations of the algorithm to enable interoperability with arbitrary CPUs and GPUs.

The following master's theses were supervised alongside the research leading to this chapter, fostering scholarly exchange between this work and the supervised theses:

- Sven Köhler. "On-Chip Accelerators on POWER8". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, May 2017. URL: https://osm.hpi.de/bookshelf/Details/531
- Joan Bruguera Micó. "Improved Data Transfer Efficiency for Scale-Out GPU Workloads using On-the-Fly I/O Link Compression". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2020. URL: https://osm.hpi.de/bookshelf/Details/539

Furthermore, partial results of the work presented in this chapter have been published:

- Max Plauth and Andreas Polze. "Towards Improving Data Transfer Efficiency for Accelerators Using Hardware Compression". In: *Proceedings of the Sixth International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. Nov. 2018, pages 125–131. DOI: 10.1109/CANDARW.2018.00031
- Max Plauth and Andreas Polze. "GPU-Based Decompression for the 842 Algorithm". In: *Proceedings of the Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. Nov. 2019, pages 97–102. DOI: 10.1109/CANDARW.2019.00025
- Max Plauth, Joan Bruguera Micó, and Andreas Polze. "Improved Data Transfer Efficiency for Scale-Out Heterogeneous Workloads Using On-the-Fly I/O Link Compression". In: *Concurrency and Computation: Practice and Experience* (Dec. 2020), e6101. DOI: 10.1002/cpe.6101

This chapter is structured as follows. Section 3.1 motivates the demand for hardware-accelerated and software-based high-throughput compression facilities. The *842* compression algorithm employed by the *NX-842* on-chip hardware compression accelerator is introduced in Section 3.2. Section 3.3 then introduces the *lib842* compression library on a conceptual level. The implementation details of all major hardware-based and software-based compression facilities provided by the *lib842* library are detailed in Section 3.4. Both the compression ratio and the throughput of all implementations are evaluated in Section 3.5 using a wide range of test systems. Finally, the chapter is summarized in Section 3.6.

3.1 Motivation and Problem Statement

Combining the strengths of diverse heterogeneous system resources such as CPUs, GPUs, and other accelerators in state of the art heterogeneous computer architectures is vital to keep up with the demand for compute capacity imposed by the abundance of data accumulating in the age of digitization. To solve a given task using the respective strengths of the different heterogeneous system resources at hands, transferring data back and forth among the local memories of the involved system resources is a critical aspect of heterogeneous systems [61]. Except for highly integrated approaches where all major system resources can be accommodated on a single System on a Chip (SoC) or package, data transfers across the local memories of dedicated system resources in heterogeneous systems are usually entailed by a certain level of performance degradation and increased energy demand compared to operations that can be completed without accessing the memory of another compute resource [88]. Unfortunately, the number of workloads exceeding the compute capacity of single nodes increases with growing data volumes, requiring computations to be scaled out across numerous systems. For example, the soaring popularity of deep learning applications has drastically increased the demand for both Cloud-based and private GPU clusters [84]. As the penalties related to data transfers already affect certain workloads executed on a single system, scale-out workloads are more susceptible to the overhead of data transfers as data has to be conveyed across comparatively slow inter-node interconnects. While the forthcoming commoditization of coherent interconnection technologies (cf. Section 2.2.1) has the potential to improve upon the situation, the penalties associated with data transfers will not disappear entirely.

To further improve the efficiency of data transfers, on-the-fly data compression has already been demonstrated as a viable approach both in academic research and in commercial products. On the level of single compute nodes, most of the generally applicable approaches are restricted to individual system resources [179, 207]. Belonging to this category, the compute data compression feature introduced by NVIDIA with their latest generation of Ampere GPUs serves as a notable example of a product-grade approach, yielding up to 4× effective bandwidth improvements for accessing data residing in the local GPU memory or in the L2 cache of the GPU [30]. While all the previously mentioned on-the-fly data compression approaches rely on custom hardware-based compression facilities, a software-based lightweight compression technique has been used to improve data transfer efficiency among system resources on the intra-node level [86]. Regrettably, the compression technique used in this approach is only applicable to workloads operating

on unsigned integer data, as it truncates all unused high-order bits. Similarly, the High-Performance Computing (HPC) community has successfully employed floating point compression schemes for on-the-fly data compression that assume specific properties of the data in order to improve the efficiency of inter-node data transfers [168]. Regarding generic software-based compression facilities however, their limited throughput has thus far restricted beneficial impacts of on-the-fly compression for inter-node data transfers to very slow networks (e.g., gigabit Ethernet) [51].

On-chip hardware compression accelerators for generic compression algorithms have become available in certain Commercial Off-the-Shelf (COTS) CPUs such as the Cavium ThunderX processor family [26] as well as the lineup of IBM POWER processors introduced since the POWER7+ CPU [17]. Delivering up to 36.8 GB/s of compression throughput per processor socket, the *NX-842* accelerators available in POWER CPUs [17] provide sufficient compression throughput to saturate even state-of-the-art high-speed inter-node interconnects [85]. With such potent compression facilities at hands, the idea of applying on-the-fly data compression using a generic compression algorithm to improve data transfer efficiency in heterogeneous systems is intriguing. Except for the confined range of CPUs equipped with on-chip hardware compression accelerators however, the absence of high-throughput generic compression facilities on the side of all other heterogeneous system resources is inhibiting the feasibility of compressed data exchange. Furthermore, even though the Linux kernel contains drivers for the *NX-842* as well as for most available COTS on-chip hardware compression accelerators,[†] their hardware compression facilities are only accessible from other kernel resources using the internal *Linux Kernel Crypto API*. In this chapter, this thesis addresses the latter issue by proposing the first method for enabling user-space applications on Linux to access the compression facilities of the *NX-842* accelerator. To address the former issue, CPU-based and GPU-based high-throughput software compression facilities for the proprietary *842* compression algorithm employed by the *NX-842* accelerator are presented. By making these artifacts available in the form of the *lib842* compression library, this chapter lays out the foundation for investigating the impact of *On-the-Fly I/O Link Compression* for data transfers among CPUs and GPUs in scale-out GPU clusters in Chapter 4.

3.2 The 842 Compression Algorithm

The *842* algorithm [54, 17] is a generic compression algorithm that has been designed with the use case of transparent main memory compression in mind. As this use case requires compression and decompression facilities that offer high throughput and low latency, the algorithm has been designed accordingly to enable hardware-based implementations that can be placed directly on transmission channels [17]. The first implementation of the algorithm is a hardware-based implementation that has been introduced with the *NX-842* on-chip compression accelerator, which is available in all IBM POWER processors introduced since the POWER7+ [17]. The *842* algorithm can be attributed to the family of Lempel-Ziv derivatives [54]. The compression process deviates from the original Lempel-Ziv algorithm [215] in several aspects. However, decompression works very similar compared to LZ'77 [54].

[†] `https://github.com/torvalds/linux/tree/master/drivers/crypto`

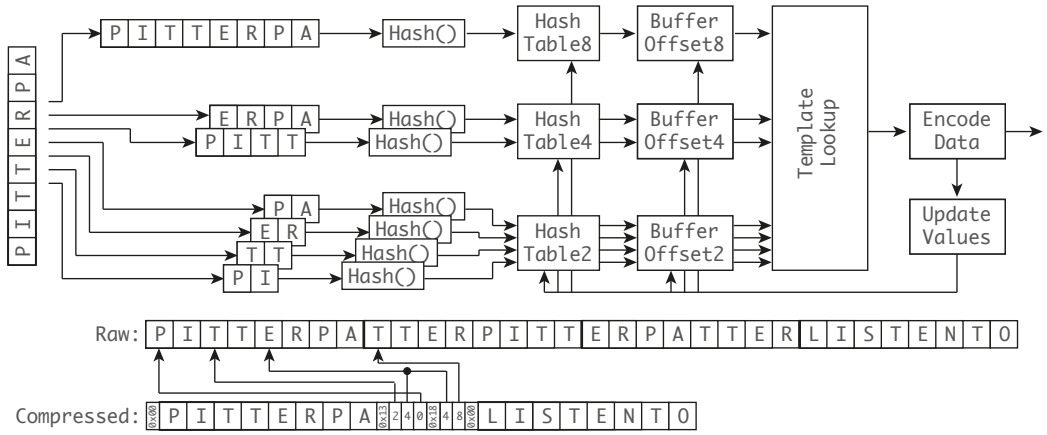

Figure 3.1: The *842* compression algorithm operates in units of 8 bytes, treating the input data as sub-phrases of 8, 4 and 2 bytes length. The algorithm uses a fixed set of template codes (see Table 3.1) to encode 8 bytes of raw data by specifying a permutation of offsets to past occurrences or literals of 8, 4 and 2 bytes length, as demonstrated in the example.

As illustrated in Figure 3.1, the *842* algorithm [54, 17] operates on units of 8 bytes, treating input data as sub-phrases of 8, 4 and 2 bytes length, respectively. For each phrase length, a hash table holds offsets to sub-phrases that have already appeared in the raw data stream within a certain window. Depending on the outcome of the lookup, a compression template is chosen from the fixed list of available templates (see Table 3.1), with each template encoding 8 bytes of raw data. Each 5-bit template encodes a permutation of offsets or literals of 8, 4 and 2 bytes length, followed by the actual offsets and literals. Offsets to 2 and 8-byte phrases are encoded using 8 bits, whereas offsets to 4-byte phrases are encoded using 9 bits, resulting in the parameter sizes specified for each template in Table 3.1. With a clock frequency of 2.3 GHz, and the ability to ingest 8 bytes per cycle, one *NX-842* unit can achieve a maximum throughput of 18.4 GB/s [17]. With two *NX-842* units per socket, the total compression throughput of a POWER processor can be as high as 36.8 GB/s [17].

The example provided in Figure 3.1 demonstrates how the 32-byte string `PITTERPATTERPITTERPATTERLISTENTO` is compressed using four templates. The template `0x00` is used to encode the raw literal `PITTERPA`, since no matching sub-phrases have appeared in the raw data stream beforehand. The second template `0x13` encodes `TTERPITT` by providing offsets to two 2-byte phrases in the uncompressed data stream at the positions 2 (`TT`) and 4 (`ER`), as well as an offset to a 4-byte phrase at the position 0 (`PITT`). The third template `0x18` encodes `ERPATTER` by providing offsets to two 4-byte phrases at the positions 4 (`ERPA`) and 8 (`TTER`). Finally, the last template `0x00` encodes `LISTENTO` as a raw literal.

Table 3.1: A 5-bit template encodes precisely 8 bytes of raw data using four consecutive actions. Actions include raw data phrases `D` and index references `I`, both in variants yielding 8, 4 and 2 bytes respectively. No-op actions `N0` are used to fill up unused action slots (not shown).

Template	Parameters	Actions and corresponding bytes encoded by the template.							
		1	2	3	4	5	6	7	8
`0x00`	64 bits	D8							
`0x01`	56 bits	D4				D2		I2	
`0x02`	56 bits	D4				I2		D2	
`0x03`	48 bits	D4				I2		I2	
`0x04`	41 bits	D4					I4		
`0x05`	56 bits	D2		I2			D4		
`0x06`	48 bits	D2		I2		D2		I2	
`0x07`	48 bits	D2		I2		I2		D2	
`0x08`	40 bits	D2		I2		I2		I2	
`0x09`	33 bits	D2		I2			I4		
`0x0A`	56 bits	I2		D2			D4		
`0x0B`	48 bits	I2			D4			I2	
`0x0C`	48 bits	I2		D2		I2		D2	
`0x0D`	40 bits	I2		D2		I2		I2	
`0x0E`	33 bits	I2		D2			I4		
`0x0F`	48 bits	I2		I2			D4		
`0x10`	40 bits	I2		I2		D2		I2	
`0x11`	40 bits	I2		I2		I2		D2	
`0x12`	32 bits	I2		I2		I2		I2	
`0x13`	25 bits	I2		I2			I4		
`0x14`	41 bits	I4					D4		
`0x15`	33 bits	I4				D2		I2	
`0x16`	33 bits	I4				I2		D2	
`0x17`	25 bits	I4				I2		I2	
`0x18`	18 bits	I4					I4		
`0x19`	08 bits	I8							
`0x1A`	00 bits	Reserved / Unused							
`0x1B`	06 bits	Repeat preceding 8 B for N times							
`0x1C`	00 bits	Emit 8 B of zeros							
`0x1D`	00 bits	Reserved / Unused							
`0x1E`	00 bits	End of compressed bitstream							

3.3 Lib842: A User-Space Library for 842 Compression

The availability of high-throughput and low-latency compression and decompression facilities accessible from user-space are stringently required in order to enable application developers to improve data transfer efficiency among heterogeneous system resources based on data compression. Even though the *842* compression algorithm has been designed with transparent main memory compression in mind (cf. Section 3.2), it should be well suited for compressing data transfers among system resources, too since the requirements of both use cases are virtually identical in that they require high-throughput and low-latency compression and decompression facilities. Prior to this work however, both hardware-based and software-based implementations of the *842* compression algorithm have been widely inaccessible for user-space applications. The only way for user-space applications to leverage the resources of the *NX-842* on-chip compression accelerator was to use IBM's proprietary Advanced Interactive eXecutive (AIX) operating system, where an in-built user-space API exposes the hardware-accelerated compression facilities. On Linux however, both the *NX-842* on-chip compression accelerator and a rudimentary a software-based fallback implementation were only accessible from kernel-space through the Linux Crypto API. To make high-throughput *842* compression and decompression facilities available to user-space applications on Linux, this chapter introduces the *lib842* compression library.

As its most prominent contribution, the *lib842* library introduces the first available approach for making the resources of the *NX-842* on-chip compression accelerator available to user-space applications running on Linux. To provide high-throughput *842* compression and decompression facilities on compute resources that are not equipped with a corresponding compression accelerator, library contributes software-based compression and decompression facilities several execution targets. Hiding the implementation details of all hardware and software-based implementations from any users of the library, *lib842* exposes all available implementations through an implementation-agnostic interface as illustrated in Figure 3.2. Of course, the implementation-agnostic interface also facilitates extensibility, making it relatively easy to add further implementations (e.g., for other heterogeneous system resources such as Field-Programmable Gate Arrays (FPGAs)). A brief overview of all implementations that are currently provided by the *lib842* library is outlined hereinafter:

Hardware-based On-Chip Compression Accelerator (*NX-842*) Based on a custom kernel module, this implementation exposes the compression and decompression facilities of all available *NX-842* on-chip compression accelerators to user-space.

Software-based Compression and Decompression (CPU Baseline) This version is a user-space port of the software-based fallback-implementation of *842* compression and decompression facilities provided by the Linux kernel in case an *NX-842* accelerator is not available. Even though this version merely provides minor compression and decompression throughput, it served as a golden unit during the development of all other implementations.

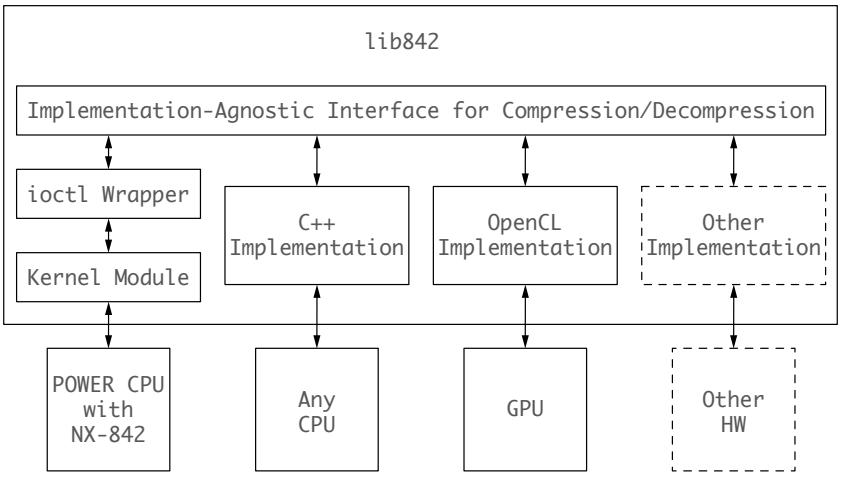

Figure 3.2: The *lib842* compression library provides implementations for hardware-accelerated compression and decompression using *NX-842* on-chip compression accelerators, software-based compression and decompression on arbitrary CPUs, and GPU-based decompression on OpenCL-capable GPUs. The implementation-agnostic interface abstracts away the details of all existing implementations, but also serves as the contact point for adding further implementations.

Software-based Compression and Decompression (CPU Optimized) Building up on top of the CPU-based baseline implementation, this version uses several optimization techniques to yield significantly higher compression and decompression throughput.

Software-based Decompression using OpenCL (GPU) To provide high-throughput decompression facilities for *842* compressed data on a wide range of GPUs, this version is implemented using the Open Computing Language (OpenCL) framework. Relying on many optimization-techniques used in the optimized CPU-based implementation, some additional GPU-specific optimizations are employed in this implementation.

3.4 Implementation

This section provides insights into all major implementations provided by the *lib842* compression library as highlighted in Section 3.3. Section 3.4.1 presents the first available approach for leveraging hardware-accelerated *842* compression on Power CPUs from user-space applications running on Linux. Serving as a golden unit during development, Section 3.4.2 briefly outlines the challenges of porting Linux kernel code to user-space for the software-based baseline implementation. Section 3.4.3 elaborates on the optimization strategies applied to the optimized software-based compression facilities that can be executed on arbitrary CPUs. Finally, Section 3.4.4 documents the optimization techniques used in the GPU-based decompression component implemented in OpenCL.

3.4.1 Hardware-based On-Chip Accelerator (*NX-842*)

On all POWER microarchitectures available at the time of writing, the *NX-842* on-chip compression accelerators can only be accessed directly from kernel-space. While the AIX operating system provides a corresponding user-space Application Programming Interface (API), comparable interfaces are not available in the Linux kernel even though the *NX-842* accelerators have already been in use for some time to implement the *zram* memory compression feature analogue to the Active Memory Expansion (AME) feature in AIX. To the best of the author's knowledge, the approach presented in this section is the first to make the resources of the *NX-842* on-chip compression accelerators available to user-space applications running on Linux. The presented approach involves several layers of indirection, as visualized in Figure 3.3.

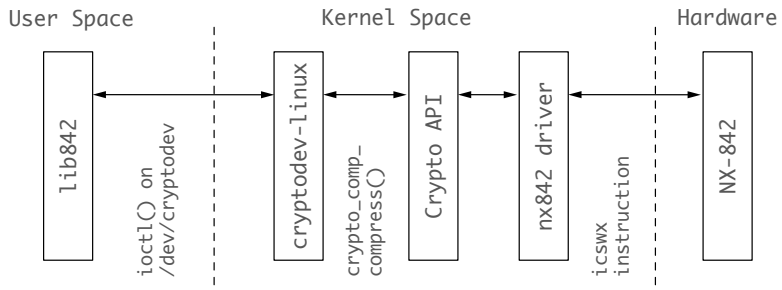

Figure 3.3: On the kernel level, a driver for the *NX-842* accelerator, a wrapper for the Linux Kernel Crypto API, as well as a modified version of the `cryptodev-linux` out-of-tree kernel module are required to expose the resources of the *NX-842* on-chip compression accelerator to user-space applications. These components are augmented by a user-space wrapper provided by *lib842* for interacting with the `/dev/crypto` special file exposed by the cryptodev-linux kernel module through `ioctl()` system calls.

Closest to the hardware, the `nx842` driver available in the Linux kernel interacts with the hardware accelerator. Various code paths are available in the driver to cover different hardware configurations (e.g., to differentiate between native hardware and the *PowerVM* hypervisor). Another aspect the driver needs to provide different code paths for is that the method for interacting with the *NX-842* units differs depending on the employed generation of the POWER microarchitecture. On POWER7+, POWER8, and POWER8+ CPUs, all on-chip accelerators are accessed through the privileged Initiate Coprocessor Store Word Indexed (`icswx`) instruction and its associated communication protocol [17]. From POWER9 onwards, the Virtual Accelerator Switchboard (VAS) facilities have been introduced with the goal of making on-chip accelerator resources accessible from user-space [83]. For user-space access to be enabled, the firmware has to be modified to initialize accelerators and make them available in the device tree. While corresponding modifications are available for other on-chip accelerators such as the *NX-GZIP* [164], comparable firmware modifications are thus far unavailable for *NX-842* accelerators. As a result thereof, the *NX-842* remains to be accessible from kernel-space only even on VAS-enabled POWER CPUs.

A wrapper then hides the various code paths available in the `nx842` driver as well as the software-based fallback implementation of the *842* algorithm available in the Linux kernel [118], utilizing the interface for compression algorithms in the Linux Kernel Crypto API. Unfortunately, the subset of the Linux Kernel Crypto API exposed to user-space through the `AF_ALG` socket type does not include compression facilities. Even though the `cryptodev-linux` out-of-tree kernel module [‡] provides user-space access to a bigger portion of the Linux Crypto API compared to the `AF_ALG` socket type, it too does not provide access to any compression facilities. Therefore, the `cryptodev-linux` kernel module was extended in the context of this thesis to expose the resources of the *NX-842* as well as other hardware accelerated compression facilities available through the Linux Kernel Crypto API through the `/dev/crypto` special file.

On the user-space side, *lib842* takes care of interacting with the special file through `ioctl()` system calls. Since each `ioctl` request on the `/dev/crypto` special file involves a system call, the interface was augmented with a batching method that enables *lib842* to submit multiple chunks for compression or decompression using a single system call. To further optimize the interaction between *lib842* and `/dev/crypto`, session caching was implemented in order to re-use sessions. With these optimizations in place, the proposed approach is able to achieve high throughput for compression or decompression from user-space applications with minimal load on the CPUs.

3.4.2 Software-based Compression and Decompression (CPU Baseline)

Prior to this work, the only software-based implementation of the algorithm available to the public prior has been the fall-back implementation in the Linux kernel [118], which is also only accessible from kernel-space. As a first step, this basic implementation was ported to user-space by replacing all kernel dependencies with corresponding equivalents. The majority of dependencies could be easily resolved by consolidating various preprocessor definitions of constants and simple functions spread across the kernel in a single header file. More complex however was the task of replacing the generic hash table facilities provided by the Linux kernel [40] with the uthash[§] C library. Finally, concatenating strings of bits on the sub-byte level required careful consideration of the execution targets endianness, as the *NX-842* operates in big-endian byte order. Delivering meager performance for both compression and decompression, the biggest value of this version is its use as a golden unit for all other software-based implementations.

3.4.3 Software-based Compression and Decompression (CPU Optimized)

Building up on top of the baseline implementation discussed in Section 3.4.2, various optimization efforts brought forward software-based high-throughput compression and decompression facilities that can be executed on arbitrary CPUs. Recalling the basic workflow of *842* compression illustrated in Figure 3.1, all major operations were optimized as documented hereinafter.

[‡] `https://github.com/cryptodev-linux/cryptodev-linux`
[§] `https://troydhanson.github.io/uthash/`

3.4.3.1 Fast Hash Tables

With efficient hash table lookups being the major potential bottleneck of the compression process, the general-purpose hash tables used in the baseline version were replaced with a very simplistic hashing mechanism. First, the 8, 4, and 2-byte sub-phrases are stored in a vector of 64-bit unsigned integer values. The vector-based representation with a uniform data type enables compilers to perform auto-vectorization of most subsequent operations. Using a vector-scalar-multiplication, all fields of the vector are multiplied with the largest prime number that falls within the range of a 64-bit unsigned integer. A right shift operation truncates the result of the multiplication results to its n_{hash} most significant bits, yielding a vector of hashes. For each sub-phrase lengths, two buffers are used to form a basic hash table structure: an index array with $2^{n_{hash}}$ unsigned short integer values and a FIFO buffer of 2^8, 2^9, and 2^8 elements for 2, 4, and 8-byte sub-phrases, respectively. The latter exponents are fixed constants defined by the FIFO sizes employed by the hardware-based *NX-842* implementation. The index array uses hash-based addressing to store the offsets of corresponding values in the FIFO buffer. To retrieve the best possible performance, the hash size n_{hash} has to be chosen carefully to yield acceptable collision rates at a memory footprint that still fits into the CPU caches. For $n_{hash} = 10$, the total memory footprint amounts to $3 * 2^{10} * \texttt{sizeof(uint16\_t)} + 2^8 * \texttt{sizeof(uint16\_t)} + 2^9 * \texttt{sizeof(uint32\_t)} + 2^8 * \texttt{sizeof(uint64\_t)} = 10.5KiB$ and should fit into the L1 data cache of most recent CPU architectures. Several tests were performed to make sure the presented hashing mechanism has minimal effects on compression ratio.

3.4.3.2 Efficient Template Lookup

In the baseline implementation, the hash tables are queried for known occurrences of a phrase in a complex hierarchy of *if-else* blocks in order to determine the most suitable template code for the data at hand. This mechanism was replaced with a simple look-up mechanism, where the template key is computed as exemplified in Listing 3.1. The resulting template key is used to retrieve the template code (as specified in Table 3.1) at the i-th position of a fixed lookup table. Since the `isInHashTable` flags can be computed branchless, the entire look-up is free of any branches.

Listing 3.1: Prime numbers 13, 53, and 149 are used to encode matches of 2, 4 or 8 byte phrases, respectively. To encode the action slot of the match, prime numbers 3, 5, 7, and 11 are used to encode a matching phrase in the first, second, third, or forth action slot. When a known value is found in a hash table, the primes indicating phrase-length and position are multiplied. The prime numbers have been chosen so that higher template keys indicate more efficient template codes.

```
1 // isInHashTable has been computed earlier on without branches
2 uint16_t templateKey_21 = (13 *  3) & isInHashTable_21;
3 uint16_t templateKey_22 = (13 *  5) & isInHashTable_22;
4 uint16_t templateKey_23 = (13 *  7) & isInHashTable_23;
5 uint16_t templateKey_24 = (13 * 11) & isInHashTable_24;
6 uint16_t templateKey_41 = (53 *  3) & isInHashTable_41;
7 uint16_t templateKey_42 = (53 *  5) & isInHashTable_42;
8 uint16_t templateKey_81 = (149 * 3) & isInHashTable_81;
9
```

```
// prefer one 4-byte matches over two 2-byte matches
uint16_t high = max(templateKey_41, templateKey_21 + templateKey_22);
uint16_t low  = max(templateKey_42, templateKey_23 + templateKey_24);

// prefer one 8-byte match over two 4-byte matches
uint16_t templateKey = max(templateKey_81, high+low);
```

3.4.3.3 Optimized Template Encoder

The unoptimized baseline version encodes the template code and the four action parameters by calling an append function on the output buffer for each data item independently. Calls to the append function require a certain degree of overhead due to bookkeeping tasks for the bitstream writer. To reduce the number of append calls, fused calls to the append function were implemented for each template code, as exemplified in Listing 3.2. As an additional optimization, the append function was replaced with the buffered bitstream writer[¶] employed by the *zfp* library [117]. It accumulates bitstrings until a full 64-bit data sequence can be written to the output buffer. The buffering technique significantly reduces the complexity of appending sub-byte bitstring to the output buffer.

Listing 3.2: For all templates except for `0x00`, the template key and all action parameters are packed into a single value, which reduces the number of calls to the `stream_write_bits()` function from five invocations to a single invocation. Template `0x00` requires two invocations.

```
uint64_t out = 0;
switch(TEMPLATE_KEY) {
  case 0x00:    // { D8, N0, N0, N0 }, 64 bits
      stream_write_bits(p->stream, TEMPLATE_KEY, OP_BITS);
      stream_write_bits(p->stream, rawPhrase_81, D8_BITS);
    break;
  case 0x01:    // { D4, D2, I2, N0 }, 56 bits
    out = (((uint64_t) TEMPLATE_KEY)  << (D4_BITS + D2_BITS + I2_BITS)) |
          (((uint64_t) rawPhrase_41]) << (D2_BITS + I2_BITS))           |
          (((uint64_t) rawPhrase_23)  << (I2_BITS))                     |
          (((uint64_t) indexOffset_24));
    stream_write_bits(p->stream, out, OP_BITS + D4_BITS + D2_BITS + I2_BITS);
    break;
  ...
}
```

3.4.4 Software-based Decompression using OpenCL (GPU)

An important design goal for the GPU-based implementation [158] of *842* decompression is that it must remain fully compatible with the compressed data streams produced by the *NX-842* hardware compression accelerator. With this limitation in mind, the compression format of the *NX-842* unit (see Section 3.2) leaves no obvious venues for parallelism at the intra-chunk level of granularity. Due to the sliding window mechanism used to encode known phrases within the window as index offsets, there are no entry points that guarantee the absence of data dependencies within a chunk of compressed data.

[¶] https://github.com/LLNL/zfp/blob/develop/src/inline/bitstream.c

Therefore, naïve parallel decompression of chunks remains as the only viable venue for parallelization. However, decent decompression throughput can be achieved on various GPU hardware using the optimization strategies explained hereinafter.

3.4.4.1 Avoiding Divergent Execution

Most importantly, the amount of divergent execution among threads had to be reduced to a minimum. For the implementation at hands, divergent execution could be reduced by replacing a naïve case differentiation required to process each template code with a branch-free implementation. As outlined in Listing 3.3, the branch-free implementation strategy relies on a dictionary using the template code as a key for which it provides all parameters necessary to interpret the four actions encoded by a template code (e.g. type of action, parameter length in the compressed bitstream, and the length of the decompressed literal). Furthermore, the bitstream reader yielding an arbitrary number of bits from the compressed data stream has been reformulated to come by with very few case differentiation.

Listing 3.3: The array `dec_templates` serves as a dictionary, specifying the four actions associated with a template code. For each action, it holds the parameter size of the action (specified in bits), a tag specifying whether the action is an index action or not, and the number of raw bytes produced by the action. Based on this information, templates can be decoded without requiring a complex hierarchy of case differentiations.

```
#define OP_DEC_N0   {(N0_BITS | NO_INDEX_OP), 0}
#define OP_DEC_D2   {(D2_BITS | NO_INDEX_OP), 2}
#define OP_DEC_D4   {(D4_BITS | NO_INDEX_OP), 4}
#define OP_DEC_D8   {(D8_BITS | NO_INDEX_OP), 8}
#define OP_DEC_I2   {(I2_BITS | IS_INDEX_OP), 2}
#define OP_DEC_I4   {(I4_BITS | IS_INDEX_OP), 4}
#define OP_DEC_I8   {(I8_BITS | IS_INDEX_OP), 8}

__constant uint8_t dec_templates[26][4][2] = {
  {OP_DEC_D8, OP_DEC_N0, OP_DEC_N0, OP_DEC_N0}, // template code 0x00
  {OP_DEC_D4, OP_DEC_D2, OP_DEC_I2, OP_DEC_N0}, // template code 0x01
  {OP_DEC_D4, OP_DEC_I2, OP_DEC_D2, OP_DEC_N0}, // template code 0x02
  ...
  {OP_DEC_I8, OP_DEC_N0, OP_DEC_N0, OP_DEC_N0}, // template code 0x19
}

__kernel void decompress(__global uint64_t *in, __global uint64_t *out) {
  ...
  uint64_t template;
  do {
    template = read_bits(&buffer, OP_BITS);
    ...
    for(int i = 0; i < 4; i++) { // there are four actions to be decoded
      uint32_t dec_template = dec_templates[template][i][0];
      uint32_t is_index = (dec_template >> 7);
      uint32_t dst_size = dec_templates[template][i][1];

      uint64_t value = read_bits(&buffer, dec_template & 0x7F);

```

```
30        if(is_index) { // compilers may eliminate this conditional using predicates
31            // retrieve value from decompressed stream with index offset
32            ...
33        }
34        // assemble action results in output buffer
35      }
36      // write output buffer to out
37    } while (template != OP_END);
38  }
```

3.4.4.2 Avoid Template Lookups

Even though the approach outlined in Section 3.4.4.1 greatly reduces the occurrence of branching and improves decompression throughput on GPUs to a certain degree, the performance improvements yielded from this approach are much more distinct on CPUs, where the lookup table easily fits the L1 data cache. On most GPUs however, the limited performance improvements gained by this approach are caused by frequently accessing the constant memory for each lookup, which is a far more expensive operation than accessing the L1 data cache on CPUs. However, the regular pattern of data and index actions used in templates `0x00` through `0x18` (cf. Table 3.1) can be used to compute parameter size, action type, as well as the number of bytes produced by each action as demonstrated in Listing 3.4. It should be noted that this technique is not applicable for template `0x19`, however this is not a problem as it can be handled earlier on in the codebase alongside with the special templates `0x1A` through `0x1E`. Even though the computation of all necessary information involves expensive operations such as modulus and division operations, this approach provides higher throughput across all tested GPU hardware (cf. Table 3.2).

Listing 3.4: The regular patterns in the templates (cf. Table 3.1 can be exploided to compute parameter size, action type, as well as the number of bytes produced by each action. With this information available, the `dec_templates` dictionary used in Listing 3.3) can be removed in order to avoid costly memory access operations on each lookup in addition to avoiding diverent execution.

```
1  __kernel void decompress(__global uint64_t *in, __global uint64_t *out) {
2    uint64_t template;
3    do {
4      op = read_bits(&buffer, OP_BITS);
5      ...
6      opbits = 64 - ((op % 5) + 1) / 2 * 8 - ((op % 5) / 4) * 7
7                  - ((op / 5) + 1) / 2 * 8 - ((op / 5) / 4) * 7;
8        uint64_t params = read_bits(p, opbits);
9
10     for (int i = 0; i < 4; i++) {
11       uint8_t opchunk = (i < 2) ? op / 5 : op % 5;
12       uint32_t is_index = (i & 1) * (opchunk & 1)
13                         + ((i & 1) ^ 1) * (opchunk >= 2);
14       uint32_t dst_size = 2 + (opchunk >= 4)
15                         * (1 - 2 * (i % 2)) * 2;
16       uint8_t num_bits = (i & 1) * (16 - (opchunk % 2) * 8
17                        - (opchunk >= 4) * 16) + ((i & 1) ^ 1)
18                        * (16 - (opchunk / 2) * 8 + (opchunk >= 4) * 9);
19
```

```
20        uint64_t bitsmask = ((uint64_t)-(num_bits != 0)) &
21                            (((uint64_t)-1) >> (64 - num_bits));
22        uint64_t value = (params >> (opbits - num_bits)) & bitsmask;
23        ...
24      }
25      // write output buffer to out
26    } while (template != OP_END);
27  }
```

3.4.4.3 Optimized Memory Access Patterns

Another important optimization step was to reduce the number of global memory access operations by modifying the bitstream reader logic borrowed from the *zfp* library [117] to cache data from global memory in registers, using the granularity of a native machine word. Based on this method, significant speed-up was achieved since not every read operation on the compressed input data results in a global memory access operation.

Transposing the compressed input data with the goal of achieving coalesced memory access operations was also evaluated as an optimization technique even though it would break compatibility with the data format generated by the *NX-842* unit. However, this approach did not yield any measurable performance improvements. The reason for this is that each chunk is very likely to use a different series of template codes with differing parameter length each (cf. Table 3.1). Therefore, each thread ingests a differing amount of data in each step so that sooner rather than later threads request data from different offsets in their respective chunks, breaking the coalesced access pattern.

Finally, another attempt at improving memory access efficiency was undertaken by caching the output data in local memory. In theory, this would improve performance for index actions when phrases are copied from earlier positions of the output buffer. While this approach was able to deliver roughly $2\times$ speed-up, it could only do so for very small chunk sizes ($\leq$ 256 bytes). With each thread requiring the equivalent of one chunk of local memory, the overall consumption of local memory becomes too high for reasonable chunk sizes ($\geq$ 4 KiB), resulting in poor occupancy and thus worse performance.

3.5 Evaluation

In this section, compression throughput, the energy demand, and compression ratio of all major implementations contributed by the *lib842* compression library are evaluated. The list of evaluated implementations includes hardware-based compression and decompression using the *NX-842* on-chip compression accelerator, optimized software-based compression and decompression on CPUs, as well as GPU-based decompression implemented in OpenCL. Laying out the foundation for the evaluation, Section 3.5.1 documents the testing environment as well as the benchmark procedures used for the evaluation. Afterwards, the compression ratio delivered by both high-throughput compressors available in *lib842* is investigated in Section 3.5.2. Finally, Section 3.5.3 determines the throughput of compression and decompression operations for all evaluated implementations using the wide range of test systems provided by the testing environment.

3.5.1 Testing Environment & Benchmark Procedure

To evaluate the throughput of the compression and decompression facilities provided by *lib842* across a wide range of CPUs and GPUs, a total number of six different test systems were employed. The detailed hardware configurations used for the evaluation are documented in Table 3.2, ranging from seasoned hardware configurations to state-of-the-art high-end hardware configurations.

All compression throughput measurements presented hereinafter were performed after a fresh reboot in order to ensure a clean system state. Furthermore, no other active users or background tasks were running on the involved servers. Both for the evaluation of compression rate and compression throughput, a chunk size of 64 KiB was used.

In order to retrieve a sufficiently meaningful dataset, each benchmark was executed 25 times. Error bars are used in all plots to report the standard deviation for each measurement. Furthermore, each benchmark was preceded by a warm-up run in order to eliminate any confounding factors. All measurements presented in this work are reported as average values including standard deviation ($n = 25$).

Throughput was determined by dividing the size of the uncompressed test data set through the isolated execution time of the compression and decompression functions. The measured execution times only include the execution of the compression or decompression function, respectively, excluding potential confounding variables such as the time required for setup, data transfers and teardown.

To compare the energy demand of *NX-842*-based and software-based compression facilities in *lib842*, the energy demand of a test application performing a compression operation immediately followed by a decompression operation on the contents of a given file was measured on the *IBM Power System S824L* test system using two Microchip MCP39F511N dual-channel power measurement devices [129] and the *PINPOINT* [104] utility. Since these measurements cover the entire execution of the test application, the compression and decompression cycle is repeated 30 times in the test application in order to reduce the impact of setup, data transfers, and teardown on the overall energy draw measurements. From these measurements, the idle power draw of the test system is deducted in order to only report the share of energy demand caused by the compression and decompression process itself. Similarly, the energy demand of the GPU-based decompression process was measured using the *PINPOINT* utility [104] and the energy readings provided by the NVIDIA Management Library [141].

3.5.2 Compression Ratio

As the performance optimization techniques for compression algorithms can often have an impact on the compression ratio r, the compression ratios achieved by both the hardware-based *NX-842* units (see Section 3.4.1) and the optimized, CPU-based software implementation (see Section 3.4.3) is investigated hereinafter. The basic characteristics such as a brief description, size, and compression ratio of all employed datasets used for this evaluation are documented in Table 3.3. However, the reported compression ratios indicate that the differences in compression efficiency are negligible across all datasets. To facilitate replicability, this investigation employs well-disseminated, publicly available datasets whenever

Table 3.2: Specifications of the test systems used to evaluate the throughput of *lib842*.

	S824L	**m710p**
Model	IBM Power System S824L [81]	HPE ProLiant m710p [75]
CPU	2×IBM POWER8 (Murano), 3.42 GHz, 10C/80T each	Intel Xeon E3-1284Lv4, 2.90 GHz, 4C/8T
Memory	1024 GB DDR3 ECC, 1600 MHz	32 GB DDR3, 1600 MHz
GPU	n/a	Iris Pro Graphics P6300
OS	Ubuntu 20.04.4	Ubuntu 18.04.4
Kernel	5.4.0	4.15.0
Compiler	GCC 10.2.1 (AT 14.0)	GCC 7.4.0
OpenCL	n/a	OpenCL 2.1 NEO
GPU Driver	n/a	20.09.15980 (NEO)
	DL380 G9	**Tyan**
Model	HPE ProLiant DL380 Gen9 [73]	Tyan TN83-B8251 [130]
CPU	2×Intel Xeon E5-2620v4, 2.20 GHz, 10C/20T each	2×AMD EPYC 7282, 2.80 GHz, 16C/32T each
Memory	256 GB DDR4 ECC, 2133 MHz	256 GB DDR4 ECC, 3200 MHz
GPU	8×NVIDIA Tesla K80	NVIDIA Tesla T4
OS	Ubuntu 20.04.4	Ubuntu 20.04.4
Kernel	5.4.0	5.4.0
Compiler	GCC 9.4.0	GCC 9.4.0
OpenCL	OpenCL 1.2 CUDA	OpenCL 1.2 CUDA
GPU Driver	470.103.01	510.47.03
	DGX-1	**DGX A100**
Model	NVIDIA DGX-1 [140]	NVIDIA DGX A100 [139]
CPU	2×Intel Xeon E5-2698v4 2.20 GHz, 10C/20T each	2×AMD EPYC 7742 2.25 GHz, 64C/128T each
Memory	512 GB DDR4 ECC, 2133 MHz	1024 GB DDR4 ECC, 3200 MHz
GPU	8×NVIDIA Tesla V100	8×NVIDIA Tesla A100
OS	Ubuntu 20.04.4	Ubuntu 20.04.4
Kernel	5.4.0	5.4.0
Compiler	GCC 9.4.0	GCC 9.4.0
OpenCL	OpenCL 1.2 CUDA	OpenCL 1.2 CUDA
GPU Driver	470.103.01	470.103.01

possible. For the remaining, artificial data sets, an additional description is provided hereinafter.

The artificial datasets *periodic*, *zeros*, and *random* are self-explanatory and are used to quantify the compression ratio r for extreme cases ranging from the best case (*periodic*, *zeros*) to the worst case (*random*). Distinguishing *periodic* and *zeros* makes sense because *zeros* triggers a special template in the *842* algorithm, whereas the *periodic* dataset has to be encoded using the regular templates described in Section 3.2.

3.5.3 Compression Throughput and Energy Demand Benchmark

To gauge the compression and decompression performance characteristics of all major implementations available in the *lib842* library, throughput measurements were performed on all test systems presented in Table 3.2. The throughput measurements retrieved for hardware-based compression and decompression using the *NX-842* on-chip compression accelerator, optimized software-based compression and decompression on CPUs, and GPU-based decompression are illustrated in Figure 3.4. As a compression payload, the *enwik9* dataset (see Table 3.3) was employed.

Using a total number of four *NX-842* accelerators available in the dual-socket test-system, the throughput measured for the hardware-accelerated implementation approaches the theoretical throughput offered by two *NX-842* accelerators available per Power CPU, yielding a utilization efficiency of roughly 38 %. One might come to expect a certain level of performance loss caused by the many abstraction layers involved in the implementation (cf. Section 3.4.1). However, it appears as if the utilization efficiency of 43.48 % achieved for accessing the *NX-GZIP* accelerator from user-space based on the VAS facilities [3] does only provide slightly improved utilization efficiency. For the software-based compression operation, even dual-socket systems equipped with high-end state-of-the-art CPU models can only provide roughly on third of the compression throughput provided by the *NX-842* units available in our dual-socket Power test-system. While there still may be some room for minor optimizations in the software-based compression process, the general picture is unlikely to change fundamentally even with further optimizations in place. On the side of the decompression operation however, software-based implementations on CPUs and GPUs are able to achieve throughput levels on the high-end test systems comparable to the *NX-842* accelerators available in our dual-socket Power test-system

Additional tests using other datasets did not reveal a significant impact of the dataset on the compression throughput, except for the *zeros* dataset. There, the compression throughput for CPU-based compression roughly doubled across all node test systems. This effect is likely caused by the special compression template used for encoding an eight-byte sequence of zeros as well as the special template for encoding a repeated occurrence of eight-byte sequences. When a special template is encoded, the entire hash-and-lookup operations during the compression process is skipped, likely yielding the observed speed-up. Since both special templates address corner cases, this effect rarely occurs and compression throughput should remain stable across many datasets.

Table 3.3: Characteristics of the evaluated data sets and the respective compression ratios achieved using hardware and software-based compression.

Dataset	Description	Source	Size (Bytes)	Compression Ratio (NX-842)	Compression Ratio (SW)
Periodic	Periodic pattern of 256 bytes (`00 01 02 ... FD FE FF`).	gen.	1,000,000,000	$r = 0.210$	$r = 0.210$
Zeros	Zeroed bytes.	gen.	1,000,000,000	$r = 0.003$	$r = 0.003$
Random	Synthetic dataset consisting of pseudo random data.	gen.	1,000,000,000	$r = 1.000$	$r = 1.000$
enwik9	First 10^9 bytes from the 2006-03-03 Wikpedia dump.	[121]	1,000,000,000	$r = 0.701$	$r = 0.705$
Database	Data for DB query benchmark, inspired by TPC-H Query 1.	[206]	$n \times 2,800,000,000$	$r = 0.410$	$r = 0.410$
Books	Book reviews from Amazon.com.	[128]	22,361,866,685	$r = 0.691$	$r = 0.692$
Wikipedia	The full 2020-03-01 dump of English Wikipedia articles.	[53]	76,154,077,184	$r = 0.681$	$r = 0.683$
OLW	Dump of the Open Library works category.	[115]	10,565,840,859	$r = 0.541$	$r = 0.542$
Curiosity	Stitched panorama from the Curiosity Mars Rover	[138]	8,154,406,104	$r = 0.510$	$r = 0.510$
Telescope	Space photography taken from Spitzer Space Telescope	[201]	4,050,000,000	$r = 0.390$	$r = 0.390$

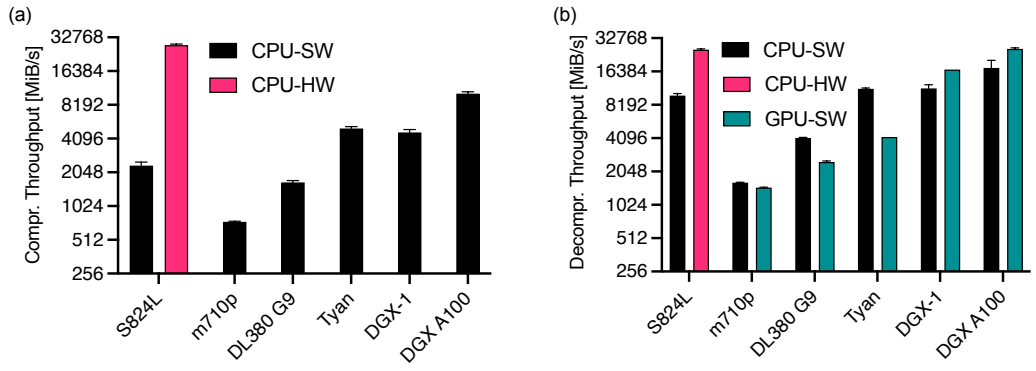

Figure 3.4: Panel (a) depicts the compression throughput measured using software-based (*CPU-SW*) and *NX-842*-accelerated (*CPU-HW*) implementations for each test system. The decompression throughput measured using the software-based implementations on CPUs (*CPU-SW*) and GPUs (*GPU-SW*) as well as the *NX-842*-based implementation (*CPU-HW*) are documented in panel (b).

Even though analyzing the energy-efficiency is not a central concern of this work, the energy demand of all major implementations provided by *lib842* was measured (cf. Figure 3.5) to investigate the energy-efficiency of different implementations and heterogeneous compute resources. The energy draw measurements for a compression/decompression-cycle were only performed on the *IBM Power System S824L* test system with the goal of enabling commensurability among hardware-based and software-based implementations. The optimized software-based implementation requires almost four times the energy required by the *NX-842* on-chip compression accelerator to perform the same compression/decompression cycle on the employed test dataset. While the superior energy-efficiency of the *NX-842* should not come as a surprise, the clear difference in energy consumption underlines the importance of being able to leverage the resources of the dormant on-chip compression accelerators in place of software-based equivalents whenever possible. On the side of GPU-based decompression, the energy required to perform the decompression process was measured on all employed NVIDIA GPU to compare the energy efficiency across various generations of GPU microarchitectures. The measurements demonstrate unequivocally how the energy-efficiency of GPUs improves with each microarchitecture. With the lower-end T4 GPU almost approaching the energy-efficiency of the *NX-842* unit, there seems to be a huge potential for improving the energy-efficiency of certain workloads when end-users are willing to relinquish a certain level of throughput in favor of improving the energy-efficiency of their application.

3.6 Summary

With the introduction of the *lib842* compression library, this chapter presented the first user-space approach for providing compression and decompression facilities based on the proprietary *842* compression algorithm. Relying on a modified version of the `cryptodev-linux` out-of-tree kernel module, the implementation details for making the high-throughput and low-latency compression and decompression facilities of *NX-842*

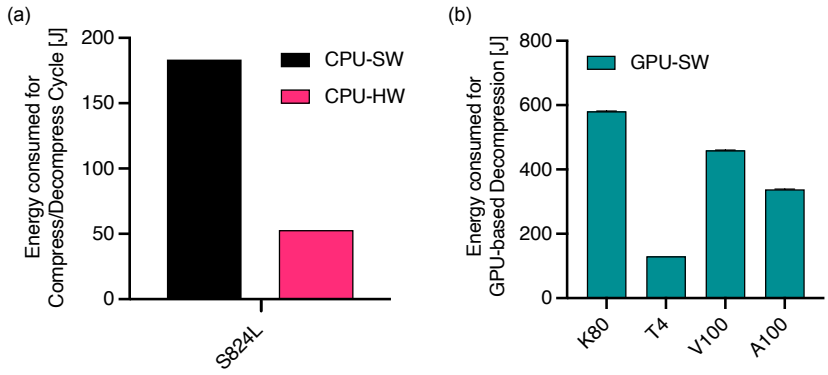

Figure 3.5: Panel (a) compares the energy required to compress and decompress the *enwik9* test dataset using the software-based implementation and the *NX-842*-accelerated approach on the *IBM Power System S824L* test system. In panel (b), the energy required to decompress the compressed *enwik9* test dataset on various generations of NVIDIA GPU microarchitectures is compared.

on-chip compression accelerators accessible to user-space applications through *lib842* were discussed. To enable compressed data exchange across heterogeneous system resources, the hardware-accelerated approach was complemented with the introduction of highly optimized software-based compression and decompression routines for arbitrary CPUs as well as OpenCL-based decompression facilities for arbitrary GPUs. The detailed evaluation of all major implementations available in *lib842* has revealed that while hardware-based compression and decompression clearly outperforms software-based approaches both in terms of throughput and energy-efficiency, the latter still provide decent throughput across the wide range of employed test systems. On higher-end systems, both CPU-based and GPU-based decompression is almost approaching the decompression throughput provided by the hardware-accelerated implementation. By making these high-throughput compression and decompression facilities available in the form of the *lib842* compression library, this chapter lays out the foundation for investigating the impact of *On-the-Fly I/O Link Compression* for data transfers among CPUs and GPUs in scale-out GPU clusters in Chapter 4.

4 Programming Abstractions for Scale-Out Graphics Processing Unit Clusters

This chapter introduces the *CloudCL* framework, which attempts to make scale-out Graphics Processing Unit (GPU) resources more accessible to a wider audience by providing abstractions that hide many aspects of the distributed memory parallel programming model associated with scale-out workloads. Based on these abstractions, the framework enables application developers and domain experts to focus on the data parallel programming model associated with GPUs. By implementing a naïve form of GPU resource disaggregation based on the *dOpenCL* Application Programming Interface (API) forwarding library for the Open Computing Language (OpenCL) ecosystem, the framework also helps operators to improve the utilization of their GPU clusters. To improve the data transfer efficiency of the API forwarding approach in commodity 10 Gbit/s Ethernet networks, the *dOpenCL* library is augmented with transparent on-the-fly data compression for inter-node data transfers based on the *lib842* compression library presented in Chapter 3.

The following master's thesis were supervised alongside the research leading to this chapter, fostering scholarly exchange between this work and the supervised thesis:

- Florian Rösler. "Dynamic OpenCL - Distributed Computing on Cloud Scale". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2017. URL: https://osm.hpi.de/bookshelf/Details/460
- Joan Bruguera Micó. "Improved Data Transfer Efficiency for Scale-Out GPU Workloads using On-the-Fly I/O Link Compression". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2020. URL: https://osm.hpi.de/bookshelf/Details/539

Furthermore, partial results of the work presented in this chapter have been published:

- Karsten Tausche, Max Plauth, and Andreas Polze. "dOpenCL–Evaluation of an API-Forwarding Implementation". In: *Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud"*. Nov. 2016. DOI: 10.13140/RG.2.2.16598.24641
- Max Plauth, Florian Rösler, and Andreas Polze. "CloudCL: Distributed Heterogeneous Computing on Cloud Scale". In: *Proceedings of the Fifth International Symposium on Computing and Networking (CANDAR)*. IEEE. Nov. 2017, pages 344–350. DOI: 10.1109/CANDAR.2017.49
- Max Plauth, Florian Rösler, and Andreas Polze. "CloudCL: Single-Paradigm Distributed Heterogeneous Computing for Cloud Infrastructures". In: *International Journal of Networking and Computing* 8.2 (July 2018), pages 282–301. ISSN: 2185-2847. DOI: 10.15803/ijnc.8.2_282

- Max Plauth, Joan Bruguera Micó, and Andreas Polze. "Improved Data Transfer Efficiency for Scale-Out Heterogeneous Workloads Using On-the-Fly I/O Link Compression". In: *Concurrency and Computation: Practice and Experience* (Dec. 2020), e6101. DOI: 10.1002/cpe.6101

This chapter is structured as follows. Section 4.1 motivates the need for programming abstractions in the context of scale-out GPU workloads. Providing such abstractions, the *CloudCL* framework is introduced in Section 4.2. To demonstrate the developer experience of *CloudCL*, Section 4.3 exemplifies the implementation of two data-intensive GPU scale-out workloads using *CloudCL*. Section 4.4 then proposes a strategy for transparently integrating on-the-fly data compression into the *CloudCL* framework to improve the efficiency of inter-node data transfers. The pipelined approach for implementing the previously proposed integration strategy is elaborated in Section 4.5. To test the impact of transparently compressed data transfers both on the effective data transfer throughput between nodes and on the overall performance of scale-out GPU workloads, a thorough evaluation is conducted in Section 4.6. Finally, the chapter is summarized in Section 4.7.

4.1 Motivation and Problem Statement

Over the last decade, the use of GPUs as a general purpose compute resource has become prevalent across arbitrary domains [24, 87, 188, 174]. Consequently, the demand for GPU compute resources has been steadily increasing over the last few years to the point where many use cases even require multiple GPUs to satisfy their resource demands. The soaring popularity of deep learning applications for example has drastically increased the demand for both Cloud-based and private GPU clusters [84].

As a result of scale-out GPU workloads becoming increasingly common, the following issues arise:

1. Application development for scale-out GPU workloads is becoming very challenging, as developers have to be adept using both data parallel programming models (e.g., OpenCL cf. Section 2.3.2.1) and distributed memory parallel programming models (e.g., Message Passing Interface (MPI) cf. Section 2.3.3.2) in addition to considering the dynamic aspects of Infrastructure as a Service (IaaS)-based resources.
2. To provide dynamic GPU resources based on public or private GPU clusters, their operators are often faced with the problem of resource fragmentation [123] as illustrated in Figure 4.1.

Numerous programming abstractions are available for alleviating access to GPU compute resources in single-node scenarios [6, 101, 50]. To tackle the first issue however, a larger number of GPUs spread out across multiple nodes have to be considered. Even though several approaches exist that make GPUs scattered across compute nodes appear as if they were local resources based on API forwarding techniques [90, 95, 7], programming abstractions targeting this larger scale however have to provide the means for splitting workloads into partitions that can be processed mostly independent of each other, without requiring fine-grained communication between GPUs [193, 13].

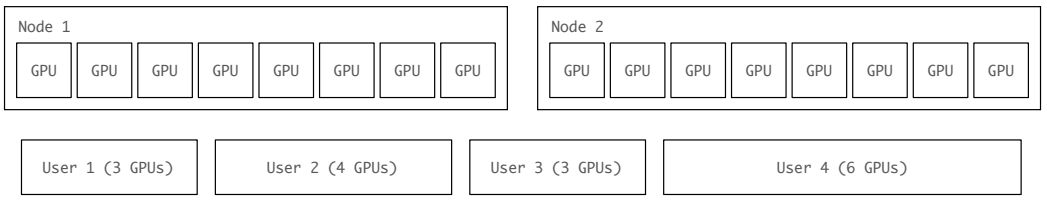

Figure 4.1: Due to resource fragmentation, it may not be possible to satisfy the resource demands of all users even though sufficient GPUs are available.

To resolve the second issue, resource disaggregation is considered a promising approach to improve the efficiency of data centers [31, 125, 80], as resource disaggregation eliminates many resource allocation issues such as fragmentation. With no implementation of resources disaggregation being ready for production yet, simple forms of resources disaggregation can be implemented in software already today. Using API forwarding techniques for example to scale out multi-GPU applications across multiple compute nodes can be considered a naïve form of resource disaggregation [123]. Usually, the lessened data transfer performance available between the host Central Processing Unit (CPU) and remote GPUs in such a naïvely disaggregated setup has to be compensated by either employing expensive high-end inter-node network technology or by restricting the range of employed workloads to compute-bound workloads operating on small datasets.

With the introduction of the *CloudCL* framework, this chapter presents an approach that unites both solution strategies by extending an existing programming abstraction framework for GPUs with the scale-out capabilities offered by an API forwarding library for the OpenCL ecosystem. Furthermore, the programming abstraction mechanisms are extended with the means necessary for defining workload partitions and for managing the resources of an ad-hoc GPU cluster. Building up on top of the efforts presented in Chapter 3, the potential of mitigating the limitations of such a naïvely disaggregated setup based on on-the-fly data compression is investigated.

4.2 CloudCL: Single-Paradigm Scale-Out GPU Computing

In addition to hiding the complexity of the distributed memory programming model for scale-out GPU workloads, the *CloudCL* framework presented in this chapter provides ad-hoc GPU clusters tailored specifically to the resource requirements of each workload. By enabling developers and domain experts to focus on the data parallel programming model, one goal of *CloudCL* is to make scale-out GPU resources accessible to a wider audience using a single-paradigm approach. Simultaneously, *CloudCL* attempts to improve the utilization of public or private GPU clusters by disaggregating GPU resources. As illustrated in Figure 4.2, *CloudCL* heavily relies on the *dOpenCL* library and the *Aparapi* framework as underlying technologies to achieve these goals. Building up on top of these technologies, *CloudCL* provides enhancements to *Aparapi* with the goal of optimizing the framework for scale-out GPU workloads. Both the underlying technologies and the enhancements are detailed hereinafter.

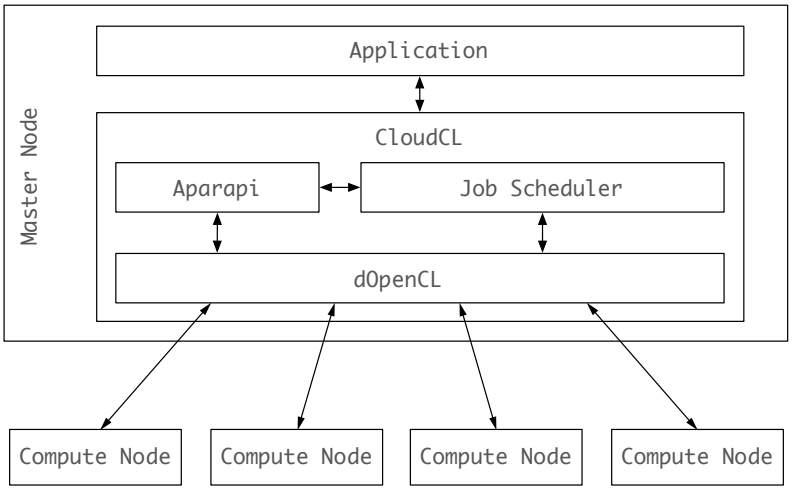

Figure 4.2: The *CloudCL* framework builds up on top of the *dOpenCL* API forwarding library for OpenCL and the *Aparapi* framework for executing native Java code on OpenCL-enabled GPUs. The underlying technologies are extended with a *job* infrastructure to hide most aspects of the distributed memory parallel programming model during the development of scale-out GPU workloads.

4.2.1 Underlying Technologies

CloudCL heavily relies on the *dOpenCL* library and the *Aparapi* framework as underlying technologies. Therefore, both technologies are introduced in greater detail hereinafter.

4.2.1.1 dOpenCL

Serving as the foundation of *CloudCL*, the *dOpenCL* API forwarding library for OpenCL enables applications to transparently utilize OpenCL devices installed in remote machines [90]. The library provides its own Installable Client Driver (ICD), which forwards the API calls to specified remote machines in the network, which run a *dOpenCL* daemon. The calls are received by the daemon and are executed using the available native OpenCL ICDs on the remote compute node with the results being returned via network. With this approach, OpenCL kernels do not require changes to run remotely as *dOpenCL* hides network transfers behind the standard OpenCL API. An overview of the architecture of an exemplary compute cluster based on *dOpencl* is shown in Figure 4.3.

4.2.1.2 Aparapi

Since the verbose nature of OpenCL can still be overwhelming for many domain experts, *CloudCL* employs *Aparapi* as an abstraction layer on top of OpenCL. *Aparapi* is a framework that drastically simplifies the usage of the OpenCL API and that minimizes development efforts of OpenCL kernels [6]. For *CloudCL*, the most important feature of *Aparapi* is that it enables developers to implement kernels in a subset of Java which is then translated to valid OpenCL kernels. Of similar importance to *CloudCL* is that *Aparapi* takes care of tedious setup-tasks and releases developers from the task of moving data back and

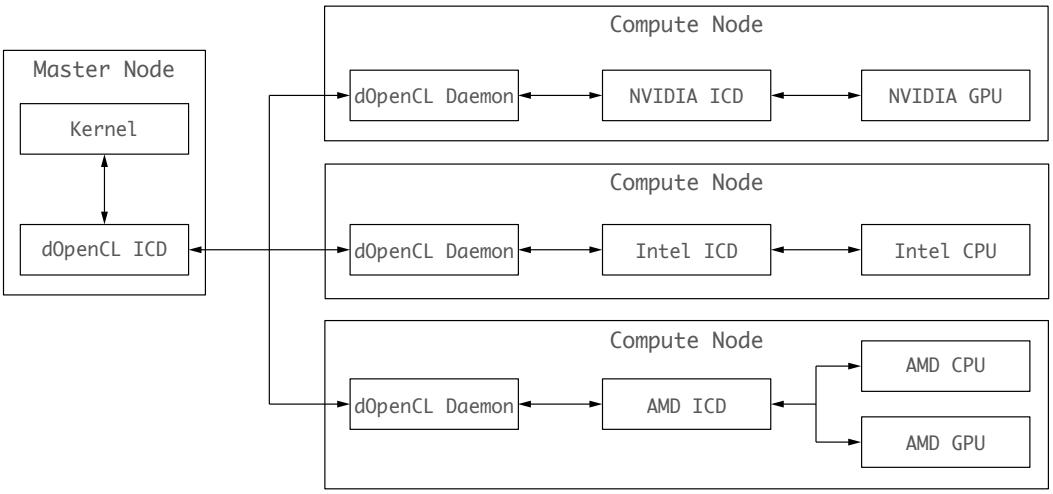

Figure 4.3: On the master node, the *dOpenCL* ICD forwards OpenCL API calls to the native OpenCL ICDs running on compute nodes via the *dOpenCL* daemon. This approach can be considered a naïve form of resources disaggregation.

forth between host and accelerator. Based on these features, the framework enables developers to express the same functionality of native OpenCL with much fewer lines of code, reducing code complexity significantly as demonstrated in Listing 4.1. The reduced complexity enables developers to implement algorithms considerably faster and offers beginners and domain experts easy access to OpenCL features without profound knowledge of low-level mechanisms.

Listing 4.1: Example of a vector addition kernel and the corresponding host code using *Aparapi*.

```
final double[] a = new double[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
final double[] b = new double[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
final double[] c = new double[10];

Kernel kernel = new Kernel() {
  @Override
  public void run() {
     int i = getgId();
     c[i] = a[i] + b[i];
  }
};

kernel.execute(10);
System.out.println(Arrays.toString(c));
```

4.2.2 Enhancements

CloudCL provides several enhancements to *Aparapi* with the goal of optimizing the framework for scale-out GPU workloads. These enhancements are explicated hereinafter.

4.2.2.1 Jobs

CloudCL introduces the concept of *Jobs* not only to serve as a unit of scheduling, but *Job* classes are also the main venue for specifying the strategy how workloads are partitioned into multiple kernel instances operating on independent sub-ranges of the input data as exemplified in Listing 4.2. Providing numerous kernel instances for a workload that can be processed independently is crucial in order for *CloudCL* to scale with the number of available GPUs. Even though semi-automatic approaches for identifying sensible sub-ranges were investigated during the development of *CloudCL* [94, 114], a manual approach where developers have to specify compartmentalization strategies themselves was favored over the immense complexity associated with semi-automatic classification mechanisms. While the manual approach requires the developer's attention, partitioning strategies tailored to a workload are likely to outperform generic approaches.

Listing 4.2: The `VecAddJob` class is responsible for dividing the input data into partitions that can be processed independently by multiple `VecAddKernel` instances.

```
1  public class VecAddKernel extends CloudCLKernel{
2    int[] a, b, result;
3
4    public VecAddKernel(CloudCLJob job, Range range, int[] a, int[] b) {
5      super(job, range);
6      this.a = a;
7      this.b = b;
8      this.result = new int[a.length];
9    }
10
11   @Override
12   public void run() {
13     result[getgId()] = a[getgId()] + b[getgId()];
14   }
15 }
16
17 public class VecAddJob extends CloudCLJob {
18   public VecAddJob(int vectorSize, int partCount, DevicePreference pref,
         ThreadFinishedNotifyable notify) {
19     super("VecAdd", notify);
20     int[] a = new int[vectorSize];
21     int[] b = new int[vectorSize];
22     int[] result = new int[vectorSize];
23
24     int partWidth = vectorSize / partCount;
25     for(int i = 0; i < partCount; i++){
26       int[] aPart= Arrays.copyOfRange(a, i * partWidth, (i + 1) * partWidth);
27       int[] bPart = Arrays.copyOfRange(b, i * partWidth, (i + 1) * partWidth);
28
29       Range range = Range.create(partWidth);
```

```
30        VecAddKernel kernel = new VecAddKernel(this, range, aPart, bPart);
31        kernel.setDevicePreference(pref);
32        addKernel(kernel);
33      }
34    }
35  }
```

Unlike the C API of OpenCL, all commands such as kernel invocations in *Aparapi* are synchronous, requiring developers to use threading in order to launch kernels on multiple OpenCL devices simultaneously. To unburden developers, the *Job* infrastructure takes care of launching all independent kernels concurrently as well as of monitoring their execution status. Furthermore, performance metrics such as data volume, transfer time and execution time are collected for all kernels to provide a solid basis for scheduling decisions.

4.2.2.2 Job Scheduler

CloudCL employs a pluggable two-tiered architecture as outlined in Figure 4.4: The first tier (*Job Scheduler*) operates on the *Job* level and does not consider individual kernel instances and the corresponding data partitions. Operating at this high level of abstraction, the first tier is mainly concerned with fairness and can be configured to use either a *First In, First Out* (FIFO) or a *Round Robin* scheduling strategy to decide which jobs are becoming eligible for being scheduled by the second tier. The first tier then hands over the kernel instances belonging to a job to the second tier (*Device Scheduler*) which is responsible for assigning individual kernels to available compute devices.

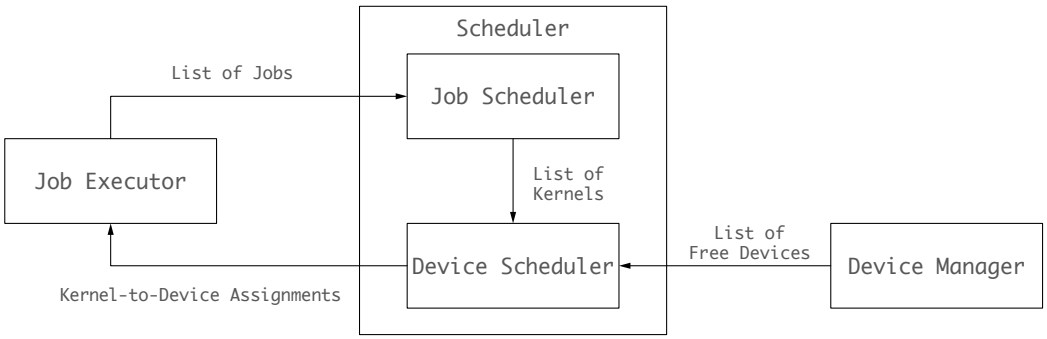

Figure 4.4: The two-tier scheduling mechanism employed by *CloudCL* considers *jobs* on the first-tier to make sure jobs are processed in a fair order. The second tier assigns the individual kernels encapsulated by jobs to compute devices whilst taking into consideration statistics regarding data volume, transfer time, and execution time of preceding runs.

Especially in the context of *CloudCL*, the net performance of a compute-device heavily depends on the speed of the network connection used to tie in the compute device to the *CloudCL* master node. Therefore, the second tier assigns kernels to devices based on the performance metrics collected on the *Job* level. It also considers the fact that the performance of OpenCL kernels can vary significantly depending on what kind of heterogeneous compute resource they are executed on. For example, kernels that may

perform exceptionally well on a GPU may run poorly on a CPU because of varying microarchitectural properties. Therefore, *CloudCL* enables developers to express preferences which device type their kernels should be executed on using the *Device Preference* attribute demonstrated in Listing 4.2. Valid values for the attribute include: `None`, `CPU only`, `CPU preferred`, `GPU only` and `GPU preferred`.

4.2.2.3 Dynamic Scaling Capabilities

To exploit the potential of private or public IaaS-based compute resources, one important goal of *CloudCL* is that resources can be dynamically added to or removed from the pool of compute resources utilized by the framework However, both OpenCL and *Aparapi* are built to run on a single machine and as such assume a fixed topology of compute devices during operation. Fortunately, *dOpenCL* supports adding and removing devices virtually by adding or removing compute nodes at runtime using the custom methods `clCreateComputeNodeWWU` and `clReleaseComputeNodeWWU`. To further provide information about the relation between compute nodes and devices, *dOpenCL* introduces another method called `clGetDeviceIDsFromComputeNodeWWU`. Especially the latter feature is crucial for removing resources dynamically, where it has to ensured that no kernels are running on a device that is about to be removed.

Since *Aparapi* is bound to the standard interface specified by the OpenCL specification, the framework cannot make use of the offered *dOpenCL* extensions. To let *CloudCL* make use the dynamic resource scaling capabilities of *dOpenCL*, *Aparapi* was extended in the context of this work by implementing the two new Java Native Interface (JNI) methods `addNode` and `removeNode`. Both call the respective *dOpenCL* functions, with `addNode` also reporting back the available devices of the added node.

4.3 Developer Experience of CloudCL

One major goal of *CloudCL* is to make resources of GPU clusters more accessible to developers and domain experts. To demonstrate the developer experience of the *CloudCL* framework, this section showcases the implementations of two workloads that are also used to evaluate the performance impact of employing on-the-fly data compression for data transfers in *CloudCL* (cf. Section 4.6). A *Semi-Sparse Matrix Multiplication* workload and an *Analytical Database Query* workload are employed as data-intensive, memory-bound workloads to evaluate the performance of *CloudCL* outside its comfort zone. Except for the Java syntax, *CloudCL* kernels themselves remain fairly similar to kernels implemented directly in OpenCL C. The aspect that changes significantly however is the host code surrounding the kernel. Therefore, the *Job* classes of both workloads are presented hereinafter to demonstrate the brevity and simplicity of *CloudCL*.

4.3.1 Semi-Sparse Matrix Multiplication

The `SemiSparseMatMulJob` class exemplified in Listing 4.3 performs a dense matrix multiplication kernel to of matrix A ($N \times M$) and matrix B ($M \times P$), yielding matrix C ($N \times P$) as a result. Matrices A and B are filled with random data except for a configurable fraction of cells that are zeros. This fraction of zeroed cells is controlled by the `sparsity` argument,

where 0 means all cells are filled with random data and 1 means all cells contain zeros. To focus on the aspect of partitioning the workload across multiple kernel instances, the process of filling the matrices was excluded from the code example for reasons of brevity. The semi-sparse scenario is intended to represent the varying degrees of compressibility encountered across the various uses cases and application domains that rely on efficient matrix multiplication. Even though sparse matrix representations such as *Compressed Sparse Row* (CSR) are employed in use cases with a high-degree of sparsity, there is a certain gray area where the space gains provided by sparse representations are not sufficient to justify the additional complexity introduced by sparse representations. A tiling strategy is employed that splits up matrix A horizontally and distributes it independent workload partitions, whereas the entire second matrix B is used by all workload partitions. The dense matrix multiplication kernel itself is implemented using a naïve implementation strategy.

Listing 4.3: Using *CloudCL*, the `SemiSparseMatMulJob` class initializes all relevant data structures on the host and specifies the strategy for partitioning the *Semi-Sparse Matrix Multiplication* workload into multiple independent kernel instances.

```
1  public class SemiSparseMatMulJob extends CloudCLJob{
2    public SemiSparseMatMulJob(int sizeN, int sizeM, int sizeP, float sparsity,
3        int tiles, DevicePreference pref, ThreadFinishedNotifyable notify) {
4      super("SemiSparseMatMul", notify);
5      final float[] a = new float[sizeN*sizeM];
6      final float[] b = new float[sizeM*sizeP];
7
8      // fill matrices with random data and sparsity
9
10     int tileHeight = sizeN/tiles;
11     for(int tile=0; tile<tiles; tile++){
12       float[] aPart = Arrays.copyOfRange(a, tile*tileHeight*sizeM, (tile+1)*
             tileHeight*sizeM);
13       Range range = Range.create2D(tileHeight, sizeP, 100, 1);
14       DenseMatMulKernel kernel = new DenseMatMulKernel(this, range, aPart, b,
             sizeM);
15       kernel.setDevicePreference(pref);
16       addKernel(kernel);
17     }
18   }
19 }
```

4.3.2 Analytical Database Query

The `DatabaseQueryJob` class presented in Listing 4.4 orchestrates an analytical database query modeled after *Query 1* of the *TPC-H* benchmark [206]. This query involves filtering and grouping operations to perform an aggregation. To focus on the aspect of partitioning the workload across multiple kernel instances, the process of data generation was excluded from the code example for reasons of brevity. At this point, the author wishes to stress that for reasons of simplicity, neither the query nor the data generator fully complies with the very complex TPC-H specification. As such, the *Analytical Database Query* workload must not be mistaken for a subset TPC-H benchmark. To facilitate efficient execution

in scale-out deployments, the table entries are split up horizontally in order to yield partitions that can be processed independently.

Listing 4.4: Using *CloudCL*, the `DatabaseQueryJob` class initializes all table columns on the host and specifies the strategy for partitioning the *Analytical Database Query* workload horizontally to yield multiple independent kernel instances.

```
1  public class DatabaseQueryJob extends CloudCLJob{
2    public DatabaseQueryJob(Integer size, int tiles, DevicePreference pref,
         ThreadFinishedNotifyable notify) {
3      super("DatabaseQueryJob", notify);
4      int[] colQuantity = new int[size];
5      int[] colExtPrice = new int[size];
6      int[] colDiscount = new int[size];
7      int[] colTax = new int[size];
8      int[] colReturnFlag = new int[size];
9      int[] colLineStatus = new int[size];
10     int[] colShippingDate = new int[size];
11
12     // generate line items
13
14     int tileHeight = size/tiles;
15     for(int tile=0; tile<tiles; tile++){
16       int start = tile*tileHeight, end = (tile+1)*tileHeight;
17       int[] colQuantitySpl = Arrays.copyOfRange(colQuantity, start, end);
18       int[] colExtPriceSpl = Arrays.copyOfRange(colExtPrice, start, end);
19       int[] colDiscountSpl = Arrays.copyOfRange(colDiscount, start, end);
20       int[] colTaxSpl = Arrays.copyOfRange(colTax, start, end);
21       int[] colReturnFlagSpl = Arrays.copyOfRange(colReturnFlag, start, end);
22       int[] colLineStatusSpl = Arrays.copyOfRange(colLineStatus, start, end);
23       int[] colShippingDateSpl = Arrays.copyOfRange(colShippingDate, start, end);
24
25       Range range = Range.create(end - start, 256);
26       DatabaseQueryKernel kernel = new DatabaseQueryKernel(this, range,
             colQuantitySpl, colExtPriceSpl, colDiscountSpl, colTaxSpl,
             colReturnFlagSpl, colLineStatusSpl, colShippingDateSpl);
27       kernel.setDevicePreference(pref);
28       addKernel(kernel);
29     }
30   }
31 }
```

4.3.3 Summary

As a central construct of *CloudCL*, the *job* classes take care of initialization, splitting up the workload into independent partitions, and launching kernel instances. The examples demonstrated in Section 4.3.1 and Section 4.3.2 show that only few lines of code are necessary to accomplish these tasks. Considering the verbosity of the native OpenCL API, this level of abstraction unburdens developers while still giving them full control over the data partitioning strategies as a performance-critical aspect.

4.4 Augmenting CloudCL with Data Transfer Compression

The overhead caused by inter-node data transfers in *CloudCL* is the biggest limitation of the framework, restricting its utility to compute-bound workloads operating on small datasets [161]. Preceding efforts of the research community have identified compression as a viable method for improving data transfer efficiency for certain application domains [189, 55]. To work around the issue of insufficient compression throughput, preceding investigations have proposed the use of offline data compression, where the payload for data transfers is available in a pre-compressed form. In Chapter 3, this thesis has demonstrated that both hardware-accelerated and purely software-based compression facilities can deliver throughput levels sufficient to saturate 10 Gbit/s, 25 Gbit/s, and even 40 Gbit/s Ethernet networks which are still the norm in the vast majority of data centers [19]. Based on this observation, this thesis hypothesizes that on-the-fly data compression can be used to improve data transfer efficiency and consequently overall performance of data-intensive scale-out GPU workloads, as illustrated in Figure 4.5.

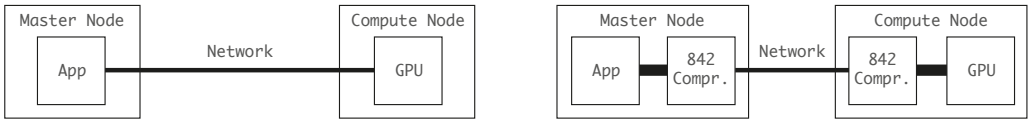

Figure 4.5: Compared to uncompressed data transfers (left), on-the-fly data compression may increase the effective bandwidth (right).

To test this hypothesis and to open up *CloudCL* to a wider range of workloads, an approach is presented hereinafter for augmenting *CloudCL* with on-the-fly data compression with the goal of improving the efficiency of data transfer across the master node and compute nodes. Building up on top of the work conducted in Chapter 3, Section 4.4.1 motivates the reasons for using the *842* compression algorithm to implement on-the-fly data compression in *CloudCL*. Since the concept of *CloudCL* detailed in Section 4.2 envisions a very arbitrary cluster model, the integration of on-the-fly data compression mandates that the assumed cluster model is defined more precisely as outlined in Section 4.4.2. Finally, the strategy for implementing on-the-fly data compression based on the integration of *lib842* into *CloudCL* is elaborated in Section 4.4.3.

4.4.1 Choice of Compression Algorithm

To improve data transfer efficiency in *CloudCL* based on on-the-fly data compression, the *842* algorithm is very well suited as it has been designed for the purpose of transparent main memory compression as discussed in Section 3.2. Unlike lightweight compression approaches that can achieve high throughput by exploiting specific characteristics of datasets such as the employed data type or a restricted range of values, the *842* algorithm is a generic compression algorithm that can be used to compress arbitrary data. As it was demonstrated in Section 3.5.2, this generic approach yields sufficient compression ratios across various data sets, including floating point data.

Another important reason for using *842* compression in this work is the availability of *NX-842* on-chip compression accelerators, which are part of all IBM Power CPUs

introduced since the POWER7+ microarchitecture [17]. The *lib842* compression library presented in Chapter 3 makes the resources of the *NX-842* accelerators accessible from user-space (cf. Section 3.4), providing very high compression throughput without having to spend excessive amounts of CPU cycles on the task. The software-based implementations provided by *lib842* are capable of providing compression throughput high enough to saturate common 10 Gbit/s, 25 Gbit/s, and even 40 Gbit/s Ethernet network links using high-end CPUs (cf. Section 3.5.3). However, the software-based approach is only used as a fall-back option in situations where no *NX-842* on-chip compression accelerators may be available.

4.4.2 Assumed Cluster Model

To saturate fast commodity networks such as 10 Gbit/s, 25 Gbit/s, and even 40 Gbit/s Ethernet or faster, on-the-fly data compression requires sufficiently high compression throughput both on the side of the master node and on the side of compute nodes. As depicted in Figure 4.6, the master node must be able to saturate the network interfaces of all compute nodes, therefore having to deal with much larger traffic volumes than each compute node. Therefore, the cluster model assumed in this work employs a master node equipped with *NX-842* on-chip compression accelerators, which are available in IBM Power CPUs. On the side of compute nodes, arbitrary CPUs types can be used as decompression is handled by the GPU-based *842* decompression facilities provided by *lib842*. Based on the assumption that the results computed by each compute node are usually smaller in volume compared to the input data, CPU-based software compression based on the *lib842* compression library is sufficient to transfer results back to the master node.

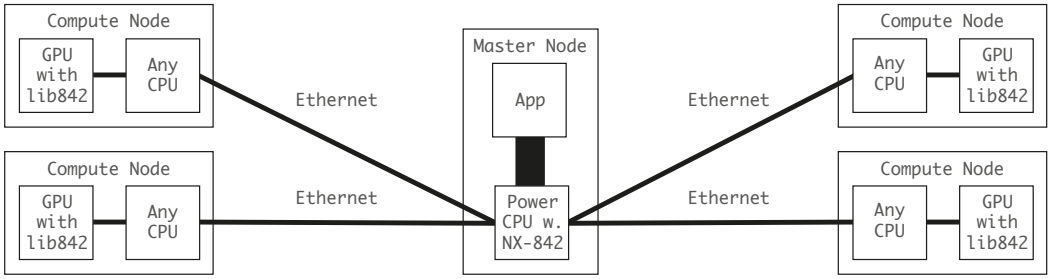

Figure 4.6: The cluster model assumed by this work includes a master node equipped with *NX-842* on-chip compression accelerators available in IBM POWER CPUs to accelerate compression, and compute nodes equipped with GPUs that use *lib842* (cf. Chapter 3) for OpenCL-based decompression on the GPU. All nodes are interconnected using a 10 Gbit/s Ethernet network.

4.4.3 Integration Strategy

The *CloudCL* architecture outlined in Section 4.2 offers various potential venues for integrating transparent on-the-fly data compression into the *CloudCL* software stack. A total number of four integration strategies were identified: On-the-fly data compression could be integrated at the level of *CloudCL*, by introducing extensions at the level of the OpenCL

API, transparently behind the implementation of *dOpenCL*, or at the network level by modifying *Boost.Asio*. While the two former approaches would require the explicit attention of application developers to use compressed data transfers, the latter strategies are completely transparent to the layers above them. Integration at the level of *CloudCL* would increase the complexity of the overall software stack as an additional *CloudCL* component would have to be introduced on the side of compute nodes in order to enable compressed data transfers. Providing custom extensions at the level of the OpenCL API would also necessitate extensive modification of the remaining components, as they would have to be adapted to make use of the modified API. On the level of the networking library, the lack of application knowledge would inhibit sophisticated optimization strategies.

The OpenCL standard defines a series of rules and requirements that an OpenCL implementation must fulfil for the movement of data between the host and the set of devices associated with a buffer via its context. Even though the OpenCL standard leaves ample room for exotic implementations, most implementations (including *dOpenCL*) follow a set of reasonable rules for data movement, aimed at minimizing unnecessary copies, and applications rely on those rules for optimal performance. Such a well-specified environment is an ideal starting position for implementing on-the-fly data compression in *dOpenCL* for all common transfer methods, so the application running on the master node does not need to be adapted to use a certain, specific mechanism in order to use compressed transfers. Since *dOpenCL* also acts as the interface between master node and compute nodes, this strategy allows for the integration of compression facilities without having to introduce any new components. Last but not least, integrating compressed data transfers transparently behind an OpenCL implementation warrants compatibility with regular OpenCL applications that are making use of multiple GPUs.

The resulting architecture for integrating on-the-fly data compression transparently at the level of *dOpenCL* is illustrated in Figure 4.7. For the integration, *dOpenCL* uses the *lib842* compression library introduced in Chapter 3 to provide access to the hardware-based compression and decompression facilities of the *NX-842* on-chip compression accelerators, if available. As a fallback option, the library uses the optimized, software-based implementation for both compression and decompression. On the side of compute nodes, *dOpenCL* is also responsible for coordinating the workflow of compressed data transfers, using the *lib842* compression library to decompress data in GPU memory based on an OpenCL-based decompression kernel. The decompressed *buffers* are left in the GPU device memory, so that the actual application kernel can work on them without any additional overhead. After the execution of the application kernel has completed, *buffers* that should be transferred back to the master node are first copied back to the main memory of the compute node. There, the CPU-based software compressor available as part of the *lib842* compression library is used to compress data prior to being sent back to the master node.

4.5 Implementation

This section is focused on documenting the cornerstones of implementing on-the-fly compression transparently behind the curtains of the *dOpenCL*. From the rules and requirements mandated by the OpenCL specification for moving data, three categories of data transfers can be derived: Data transfers from the host to a device, data transfers from

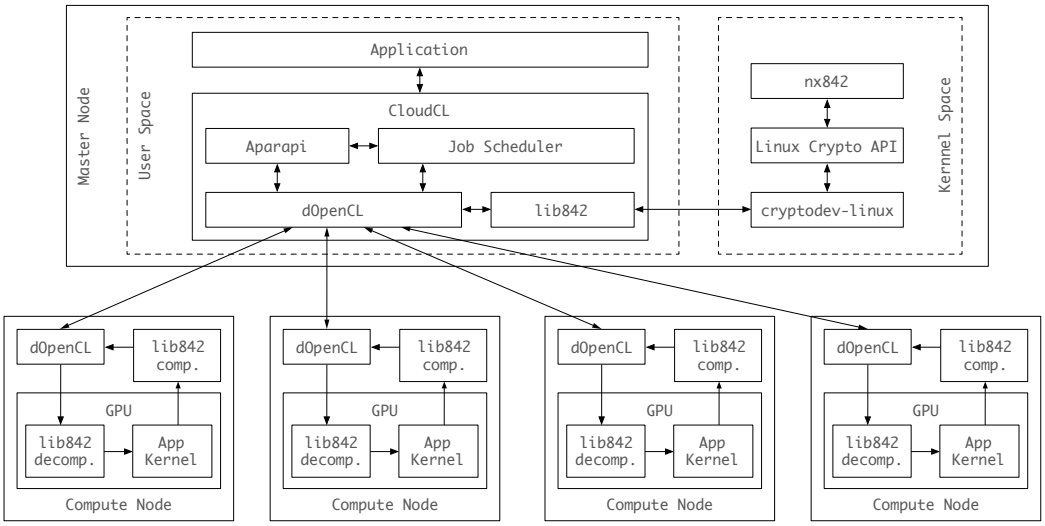

Figure 4.7: On-the-fly data compression is transparently integrated into *CloudCL* by modifying *dOpenCL* to incorporate the compression facilities of *lib842*.

a device to the host, and data transfers from a device to another device. The single-most important aspect that has to be considered for augmenting any of these types of data transfers with transparent on-the-fly data compression is to achieve a maximum level of concurrency of all operations. As illustrated in Figure 4.8, even high-throughput compression facilities are unlikely to yield performance improvements when on-the-fly data compression is implemented naïvely (b) compared to an uncompressed workflow (a). In the illustrated example, a workload is assumed that can be compressed with the ratio $r = 0.5$. Under the simplified assumption that all stages of the compressed workflow (compression, network transfer, device upload, and decompression) are taking equally long, the naïve compressed workflow (b) even may increase transfer time compared to the uncompressed workflow (a). For the compressed workflow to yield notable performance improvements compared to the uncompressed case (a), it is mandatory to introduce pipelining into the compressed workflow by overlapping the individual operations as much as possible (c).

Operating under the assumption that many workloads are ingesting more data than they egress and considering that the master node in *dOpenCL* has to supply data to a potentially larger number of compute nodes, the most important type of transfers in *dOpenCL* that have to be augmented with on-the-fly data compression are data transfers from the master node to compute nodes, as elaborated in Section 4.5.1. Section 4.5.2 then outlines the workflow for realizing compressing data transfers from compute nodes back to the master node in *dOpenCL*. Even though *CloudCL* explicitly targets workloads that can be partitioned into independent tasks that do not have to exchange data, the makeshift strategy for implementing compressed data transfers across compute nodes in *dOpenCL* is documented in Section 4.5.3 to comply with the OpenCL specification.

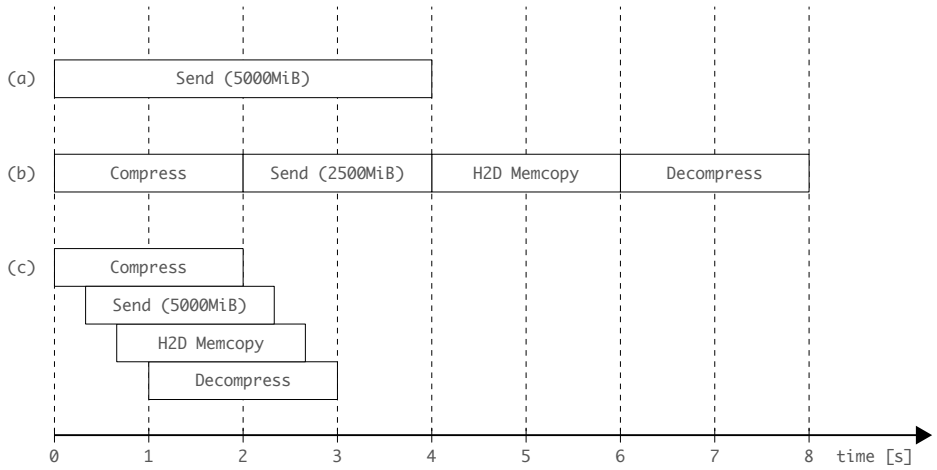

Figure 4.8: Visualization of the workflows for uncompressed data transfers (a), naïvely compressed data transfers (b), and pipelined compressed data transfers (c).

4.5.1 Master Node to Compute Node Data Transfers

In OpenCL, data transfers from the host to a device triggered explicitly by a call to `clEnqueueWriteBufffer`, a call to `clEnqueueMapBuffer` with the `CL_MAP_WRITE` or `CL_MAP_WRITE_INVALIDATE_REGION` bits set in the `map_flags` argument, or a call to `clEnqueueUnmapMemObject`. The same kind of data transfer can be triggered implicitly by the first use of a *buffer* created by calling `clCreateBuffer` with the `CL_MEM_USE_HOST_PTR` or `CL_MEM_COPY_HOST_PTR` bits set in the `flags` argument. To realize host to device transfers in *dOpenCL*, the corresponding workflow for transparently compressed data transfers from the master node to a compute node are visualized in Figure 4.9. On both the side of the master node and the compute node, all opportunities for interleaving the operations are exhausted. The approach for interleaving the compression process with network transfers on the side of the master node as well as the strategy for pipelining network transfers, device upload, and decompression on the side of compute nodes are documented hereinafter.

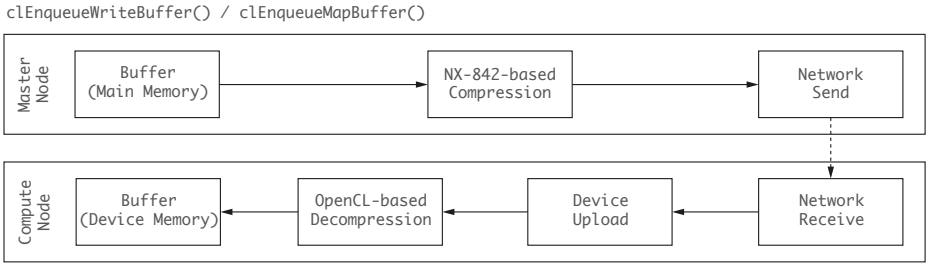

Figure 4.9: The workflow for data transfers from the master node to compute nodes leverages hardware-based compression if available to transparently compress buffers prior to sending them out to compute nodes, where they are decompressed on the GPU.

4.5.1.1 Workflow on the Master Node

To interleave the compression and send stages of the workflow on the master node, an OpenCL *buffer* is partitioned into smaller units, so-called micro-batches, that can be processed independently as illustrated in Figure 4.10. In this approach, a micro-batch is composed of 16 chunks of 64 KiB, each, resulting in a payload of 1 MiB. To fully utilize the resources of the available *NX-842* on-chip compression accelerators, a pool of compression threads is used to compress multiple micro-batches concurrently. In the send stage, a network thread then takes care of sending out all micro-batches that have already cleared the compression stage to the compute node. Especially for the send stage, pooling multiple chunks into micro-batches greatly increases the efficiency of network transfers significantly.

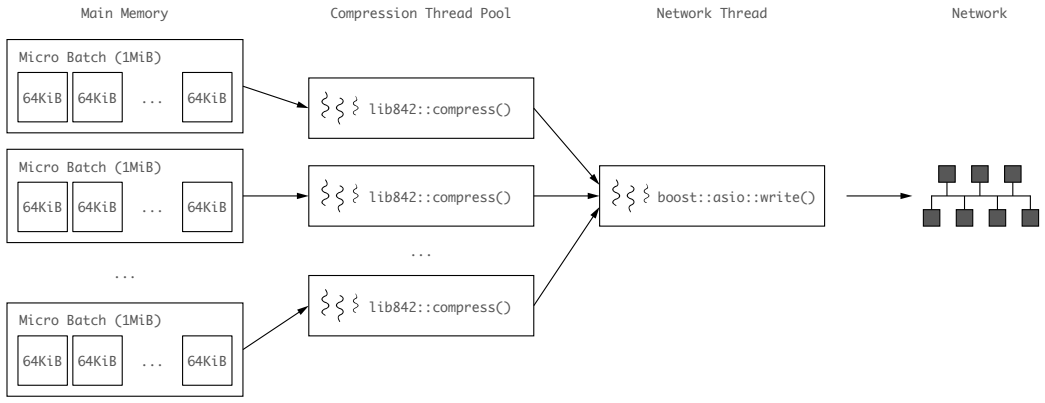

Figure 4.10: On the master node, the compression stage and the send stage are interleaved by sending out compressed micro-batches as soon as they are compressed using a pool of compression threads.

4.5.1.2 Workflow on the Compute Node

Similar to the send stage on the master node, pooling the chunks into micro-batches greatly increases the efficiency of the reception stage. On the side of the compute node, a double buffering mechanism using a pair of receive buffers allocated on the OpenCL device is used to interleave the reception stage with the device upload stage as well as the device upload stage with the decompression stage. At any single moment of time, only one buffer is mapped to the host, allowing the network thread to write received micro-batches directly into device memory. To allow for efficient device-based decompression, both buffers are dimensioned large enough to accommodate up to 512 micro-batches each, yielding a macro-batch of 512 MiB. Once the host-mapped receive buffer is filled up, it is unmapped and its sibling buffer is mapped to the host before the decompression kernel is launched on the former buffer. As demonstrated in Figure 4.11, this process is repeated in alternating order until the entire, decompressed OpenCL buffer is available in device memory.

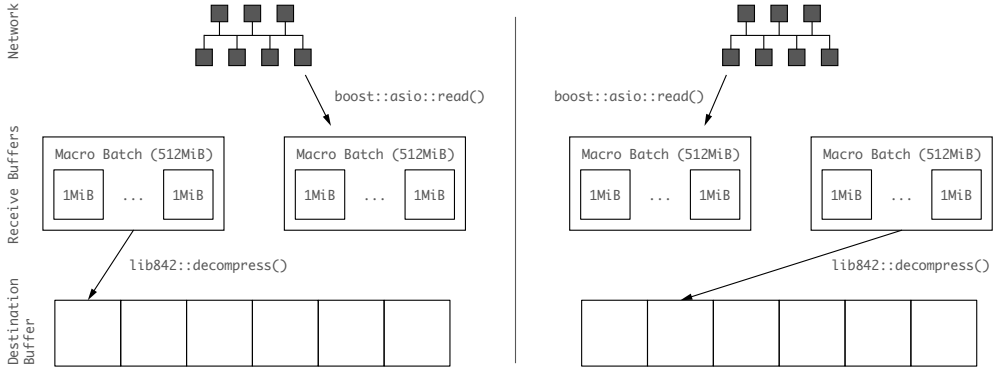

Figure 4.11: On the side of compute nodes, a double buffering mechanism is used to interleave the reception stage, the device upload stage, and the decompression stage.

4.5.2 Compute Node to Master Node Data Transfers

The OpenCL standard specifies that data transfers from a device to the host can be triggered explicitly by a call to `clEnqueueReadBuffer`, a call to `clEnqueueMapBuffer` with the `CL_MAP_READ` bits set in the `map_flags` argument, or a call to `clEnqueueUnmapMemObject`. To implement device to host transfers in *dOpenCL*, the corresponding workflow for transparently compressed data transfers from a compute node to the master node are demonstrated in Figure 4.12.

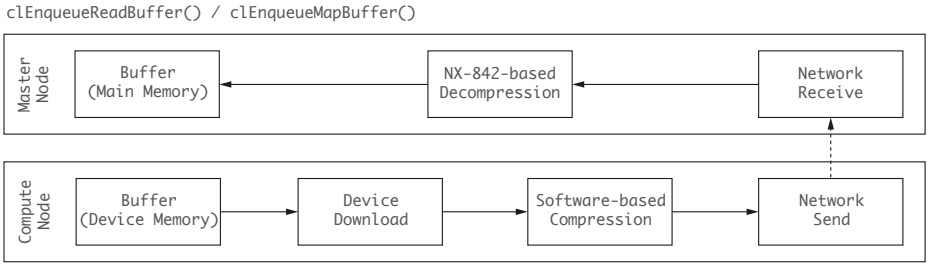

Figure 4.12: The workflow for data transfers from compute nodes to the master node uses software-based compression on CPUs to compress buffers before they are transmitted to the master node.

Both on the side of the compute node and on the side of the master node, the strategies for interleaving stages work analogous to the approaches outlined in Section 4.5.1. Due to the lack of an OpenCL-based compression kernel at the time of writing, the major exception here is that the compression stage is performed on the CPU of the compute node. Since the cluster model assumed in this work (cf. Section 4.4.2) does not assume the availability of on-chip compression accelerators in compute nodes, the software-based compression routine provided by *lib842* are utilized. The software-based compression facilities provided by *lib842* provide high compression throughput on decent CPUs (cf. Section 3.5.3), this means that workloads yielding large result sets are more susceptible to experiencing performance hits for transferring the results from the compute nodes back to the master node.

4.5.3 Compute Node to Compute Node Data Transfers

According to the OpenCL specification, data transfers from one device to another device can be triggered explicitly by calling `clEnqueueCopyBuffer`. Alternatively, device-to-device transfers can be triggered implicitly by any *command* that depends on an *event* generated by another *command* that involves the manipulation of a *buffer*. Even though *CloudCL* specifically targets workloads that can be partitioned into independent tasks that do not require any device-to-device interaction, this case has to be addressed to comply with the OpenCL specification. As such, the resulting rudimentary strategy for enabling compressed data transfers across devices in *dOpenCL* is pointed out in Figure 4.13.

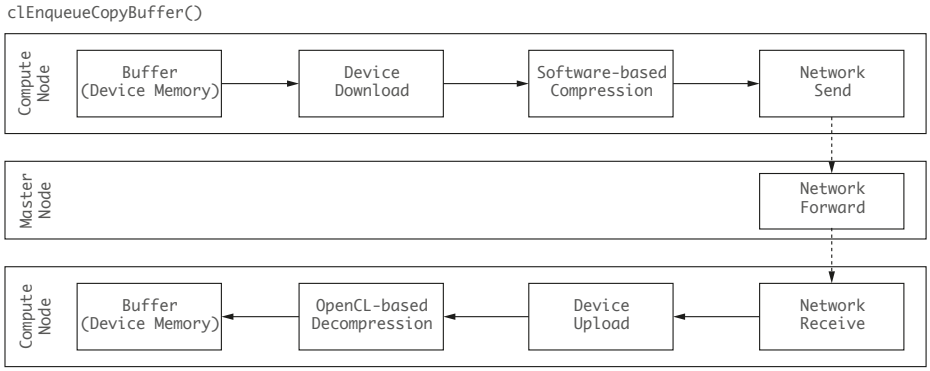

Figure 4.13: Due to the lack of peer-to-peer communication among compute nodes in *dOpenCL*, the workflow for data transfers among compute nodes requires buffers to be transmitted to the master node from which they are forwarded to the respective compute node.

Compressed data transfers among compute nodes re-use the workflows employed on the side of compute nodes for sending (cf. Section 4.5.2) and receiving (cf. Section 4.5.1.2) compressed data. With *dOpenCL* employing a host-centric architecture, peer-to-peer communication is not available and hence both workflows are stitched together using the master node, which merely forwards the compressed micro-batches received from the sending compute node to the receiving compute node. While this approach certainly provides subpar performance for device-to-device transfers, this scenario is merely covered for the sake of completeness.

4.6 Evaluation

The evaluation presented in this section focuses on investigating the performance impact of applying on-the-fly data compression to scale-out GPU workloads implemented using either *CloudCL* or *dOpenCL*. Laying out the foundation for the evaluation, Section 4.6.1 documents the testing environment as well as the benchmark procedures used for the evaluation. The effects of on-the-fly data compression on the effective data transfer performance are investigated in Section 4.6.2. Section 4.6.3 then evaluates the impact of applying data transfer compression on the total execution time of four different scale-out GPU workloads. Finally, Section 4.6.4 summarizes the major findings brought forward by the evaluation.

4.6.1 Testing Environment & Benchmark Procedure

To evaluate the effect of compressed data transfers on the execution time of scale-out GPU workloads across hardware configurations with varying levels of performance, three different classes of compute nodes are employed in addition to the master node to represent potential low, medium, and high-performance configurations of compute nodes. The detailed hardware configurations of each node type are documented in Table 4.1. The medium and high performance compute nodes are equipped with eight GPUs, each, and are connected to the same 10 Gbit/s Ethernet switch as the master node. To simulate scale-out behavior, up to eight Docker containers with one GPU attached to each container were employed as depicted in Figure 4.14. By instantiating a varying number of containers, a varying node count could be emulated.

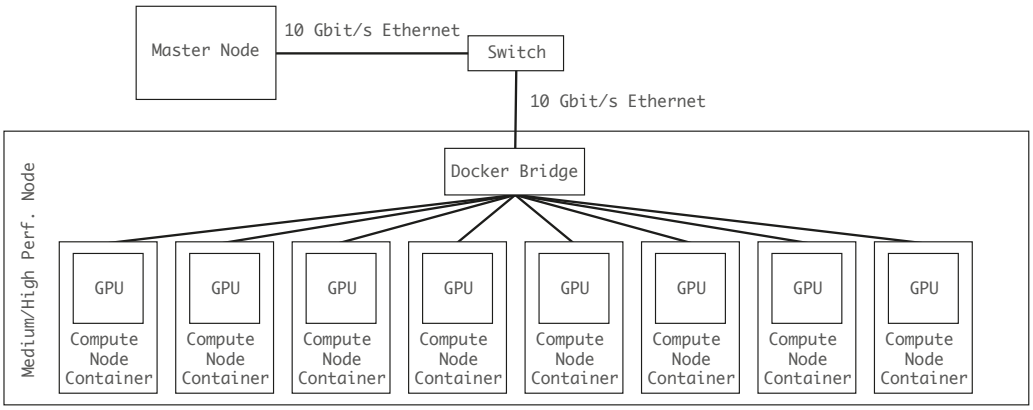

Figure 4.14: To simulate a varying number of compute nodes, the employed GPU servers were partitioned into eight compute nodes with one GPU each using Docker containers.

For the low power compute nodes however, up to eight individual bare-metal micro-servers were used instead of the container approach. All micro-servers are attached to the same 10 Gbit/s Ethernet switch as the master node. Across all tests, the same master node was used to warrant a certain degree of commensurability among the different compute node classes.

All performance measurements presented hereinafter were performed after a fresh reboot in order to ensure a clean system state. Furthermore, no other active users or background tasks were running on the involved servers and the network switch was idle. As discussed in Section 4.5, a chunk size of 64 KiB, a micro-batch size of 1 MiB (16 chunks), and a macro-batch size of 512 MiB (512 micro-batches) were employed.

In order to retrieve a sufficiently meaningful dataset, each benchmark was executed 25 times. Error bars are used in all plots to report the standard deviation for each measurement. Furthermore, each benchmark was preceded by a warm-up run in order to eliminate any confounding factors. All measurements presented in this chapter are reported as average values including standard deviation ($n = 25$).

Execution time was measured from the point where the application is started until it terminated. Therefore, all execution time measurements include the entire execution of a program, including setup, data transfers, computation phases, as well as teardown.

Table 4.1: Specifications of the test systems used to evaluate the performance impact of applying compressed data transfers in scale-out GPU workloads based on *CloudCL* and *dOpenCL*.

	Master Node	**Low Performance Node**
Model	IBM Power System S824L [81]	HPE ProLiant m710p [75]
CPU	2×IBM POWER8 (Murano), 3.42 GHz, 10C/80T each	Intel Xeon E3-1284Lv4, 2.90 GHz, 4C/8T
Memory	1024 GB DDR3 ECC, 1600 MHz	32 GB DDR3, 1600 MHz
iGPU	n/a	Iris Pro Graphics P6300
dGPU	n/a	n/a
NIC	10 Gbit/s	10 Gbit/s
OS	Ubuntu 20.04.4	Ubuntu 18.04.4
Kernel	5.4.0	4.15.0
Compiler	GCC 10.2.1 (AT 14.0)	GCC 7.4.0
OpenCL	n/a	OpenCL 2.1 NEO
GPU Driver	n/a	20.09.15980 (NEO)
	Medium Performance Node	**High Performance Node**
Model	HPE ProLiant DL380 Gen9 [73]	NVIDIA DGX-1 [140]
CPU	2×Intel Xeon E5-2620v4, 2.20 GHz, 10C/20T each	2×Intel Xeon E5-2698v4, 2.20 GHz, 10C/20T each
Memory	256 GB DDR4 ECC, 2133 MHz	512 GB DDR4 ECC, 2133 MHz
iGPU	n/a	n/a
dGPU	8×NVIDIA Tesla K80	8×NVIDIA Tesla V100
NIC	10 Gbit/s	10 Gbit/s
OS	Ubuntu 20.04.4	Ubuntu 20.04.4
Kernel	5.4.0	5.4.0
Compiler	GCC 9.4.0	GCC 9.4.0
OpenCL	OpenCL 1.2 CUDA	OpenCL 1.2 CUDA
GPU Driver	470.103.01	470.103.01

4.6.2 Effective Data Transfer Performance

To compare the effective transfer throughput between the master node and a single compute node with and without on-the-fly data compression, a modified version of the `oclBandwidthTest` sample application from the NVIDIA OpenCL SDK [142] was used. For this test, the synthetic *periodic*, *zeros*, and *random* datasets discussed in Section 3.5.2 were used as compression payloads. These artificial payloads are intended to test effective data transfer bandwidth for worst case and best case edge cases. To include more representative payloads, the *enwik9*, the *OLW*, as well as the *Curiosity* dataset (cf. Section 3.5.2) were included as well.

To evaluate effective transfer throughput in a scale-out scenario, the test application was modified in order to perform data transfers to eight nodes, simultaneously. This test uses the same data sets and only increases the data volume in proportion to the larger number of nodes. The effective transfer throughput is aggregated across all nodes.

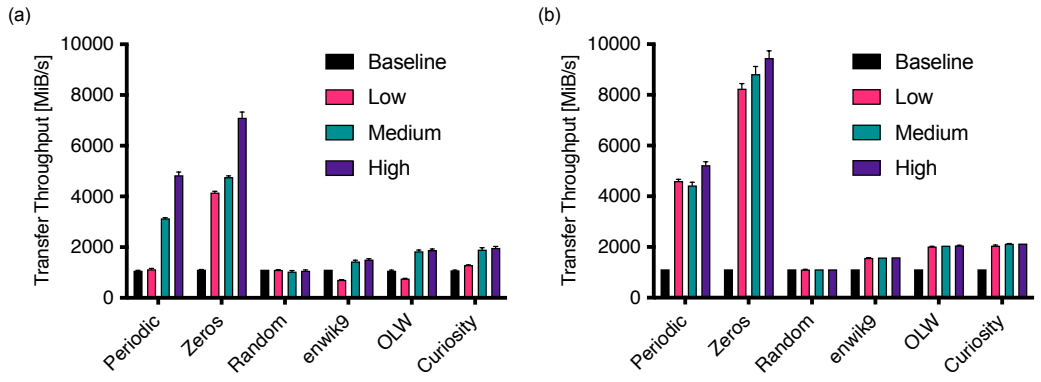

Figure 4.15: The effective transfer throughput with and without on-the-fly data compression is documented in panel (a). Here, measurements were performed between the master node and a single compute node of each class. The aggregated effective bandwidth for simultaneous transfers to eight compute nodes is illustrated in panel (b).

The measurements for the effective single-node transfer throughput and the effective scale-out transfer throughput are presented in the panels (a) and (b) of Figure 4.15, respectively. The transfer throughput tests demonstrate that using the *enwik9*, *OLW*, and *Curiosity* datasets, compressed data transfers improve effective transfer throughput between 1.29× and 1.81× using medium or high performance compute nodes in the single node scenario, and between 1.40× and 1.91× using any compute node configuration in the scale-out scenario. *Random* data as the worst-case payload has no negative impact on the throughput, whereas the benevolent *periodic* and *zeros* datasets can yield drastic performance improvements. A closer look at the scale-out results for real-world payloads reveals that the limited bandwidth, especially on the master node network interface, remains as a major bottleneck, as the aggregated effective bandwidth of the scale-out tests only slightly exceeds the single node test bandwidth.

4.6.3 Workload Benchmarks

To evaluate whether on-the-fly data compression can be used to mitigate the performance overhead caused by scaling out GPU workloads across multiple compute nodes, the best strategy is to use established benchmark suites. To compensate for the limited choice of multi-GPU benchmarks implemented in OpenCL, four custom benchmarks were implemented in the course of this work using either Java and *CloudCL* or the C++ bindings of OpenCL. In the custom benchmarks, independent kernel instances are used to process partitions of the input data in order to avoid inter-GPU communication. For each benchmark, the total execution time is measured using regular, uncompressed data transfers for inter-node communication as a performance baseline. By performing the same measurements with compressed data transfers enabled, the baseline performance measurements can be used to quantify the performance improvements introduced by the transparent integration of on-the-fly data compression discussed in Section 4.4 and Section 4.5.

The list of custom benchmarks includes a *Semi-Sparse Matrix Multiplication* workload, an *Analytical Database Query*, a *Text Search*, as well as an *Image Downscaler* workload. Implemented in Java using *CloudCL*, the *Semi-Sparse Matrix Multiplication* workload and the *Analytical Database Query* were already introduced in Section 4.3. The remaining workloads are implemented in C++, using the C++ bindings of OpenCL. Each benchmark is introduced with a brief description before the results are presented and discussed.

4.6.3.1 Semi-Sparse Matrix Multiplication

As demonstrated in Section 4.3.1, this workload benchmark is implemented in Java using the *CloudCL* framework. It assumes a matrix multiplication workload where a certain fraction of cells can be assumed to hold zero values, but where this fraction is hard to determine or where it is not large enough to justify the use of a sparse matrix representation such as *Compressed Sparse Rows* (CSR). The benchmark performs a dense multiplication of matrix A ($N \times M$) and matrix B ($M \times P$), yielding matrix C ($N \times P$) as a result. The dimensions used for A and B in this benchmark are $N = 13125 * n_{nodes}$, $M = 20000$, and $P = 25$. For node counts larger one, a tiling strategy is employed that partitions matrix A horizontally and distributes it across compute nodes, whereas the second matrix B is sent out to all compute nodes. The matrix multiplication kernel itself is implemented using a naïve implementation strategy. The amount of data to be transferred to compute nodes roughly amounts to $4 * (N * M + M * P * n_{nodes})$ bytes, and the computation requires roughly N*M*P flops.

To regulate the sparsity of the input matrices, both A and B are generated with cells holding either random numbers or a zero value according to the sparsity parameter S. For a value of $S = 0$, all cells hold randomly generated values, whereas $S = 1$ results in a fully zeroed matrix. Therefore, the matrices used in this benchmark can be compressed with a ratio r of roughly $r = 1 - S$. The impact of compressed data transfers was tested for the sparsity parameters $S = 0.33$, $S = 0.50$, and $S = 0.67$.

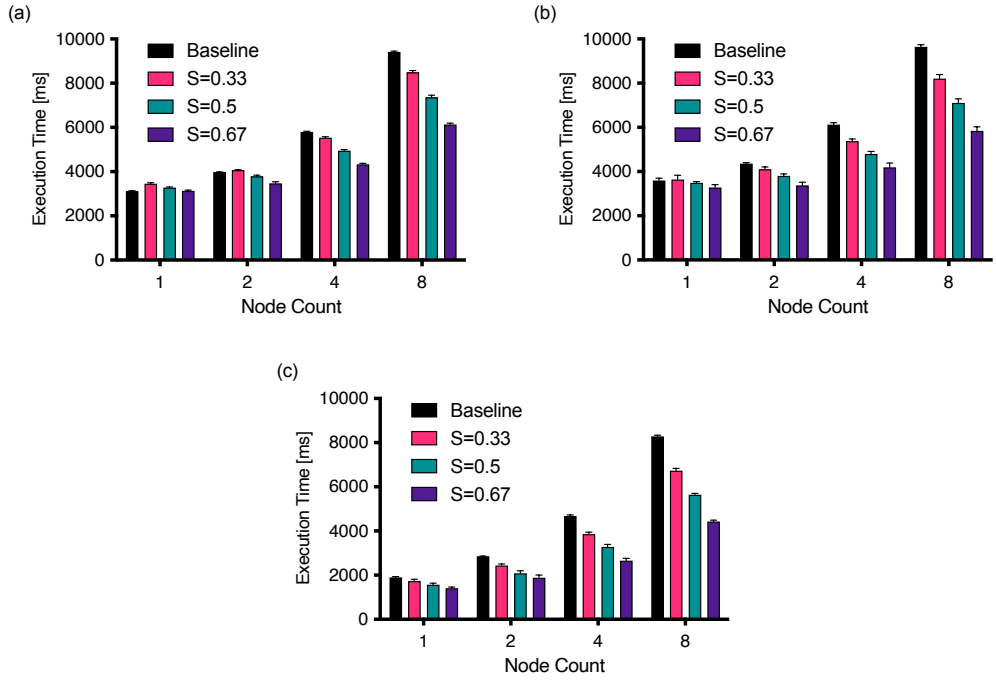

Figure 4.16: Panels (a), (b), and (c) present the execution time measurements for the *Semi-Sparse Matrix Multiplication* workload using low, medium and high power compute nodes, respectively. For each node type, the uncompressed baseline performance is compared to the performance achieved with compression enabled for the sparsity parameters $S = 0.33$, $S = 0.50$, and $S = 0.67$.

The measurements presented in Figure 4.16 demonstrate that compression can provide performance improvements across the entire range of tested sparsity parameters. On the low performance compute nodes, compression only pays off for larger node counts ($n \geq 4$) with performance improvements ranging between 1.11× for $S = 0.33$ and 1.54× for $S = 0.66$.

In contrast to the low performance compute nodes however, the medium and high performance compute node types show slight but consistent performance improvements even for low node counts ($1 \leq n \leq 2$). For larger node counts ($n \geq 4$), compression yields performance improvements between 1.23× for $S = 0.33$ and 1.87× for $S = 0.67$. However, it should be noted that this benchmark is dominated by transfer time and only a fraction of the execution time is spent on computation.

4.6.3.2 Analytical Database Query

For this benchmark, an *Analytical Database Query* was implemented in Java using the *CloudCL* framework as discussed in Section 4.3.2. The implemented workload resembles the characteristics of a column-oriented in-memory database executing an analytical query that resembles *Query 1* of the *TPC-H Benchmark* [206]. Test data for this benchmark is generated loosely following the principles of the *DBGEN* data generator [206], with the minor modification that the Java pseudo random number generator is used instead of the custom pseudo random number generator specified by the *DBGEN* data generator.

At this point, the author wishes to reiterate that for reasons of simplicity, neither the query nor the data generator fully complies with the very complex TPC-H specification. As such, the *Analytical Database Query* benchmark must not be mistaken for a subset TPC-H benchmark. The implemented query is a relatively simple, join-free aggregation query that involves a simple filter statement. Only the relevant data columns are transferred to the GPUs, using a columnar layout. The data volume to be transferred and processed amounts to $28 * 100000000$ bytes per node, and $28 * 100000000 * n_{nodes}$ bytes in total. The GPUs perform the aggregation, to the extent possible, in parallel.

The benchmark results depicted in Figure 4.17 demonstrate that compression allows the query execution to scale almost perfectly on up to four nodes, as the total execution time barely increases compared to a single node. For two to four nodes, on-the-fly data compression facilitates almost perfect scaling behavior across all node types, as the multi-node execution times are barely higher compared to the single-node configurations. For the $n = 8$ nodes, performance improvements of 1.85×, 1.9×, and 2.07× are achieved for low, medium and high performance compute nodes, respectively. Based on these observations, it seems safe to assume that the network interface on the master node is the major bottleneck. With a wider network interface available on the master node, compressed data transfers have the potential to provide perfect scale-out behavior for even larger node counts.

4.6.3.3 Text Search

Here, a simplistic *Text Search* kernel was implemented that checks for a match at each position of a large text file. Unlike the preceding benchmarks, this test was implemented in C++ using the OpenCL C++ bindings, and therefore runs directly on top of the *dOpenCL*

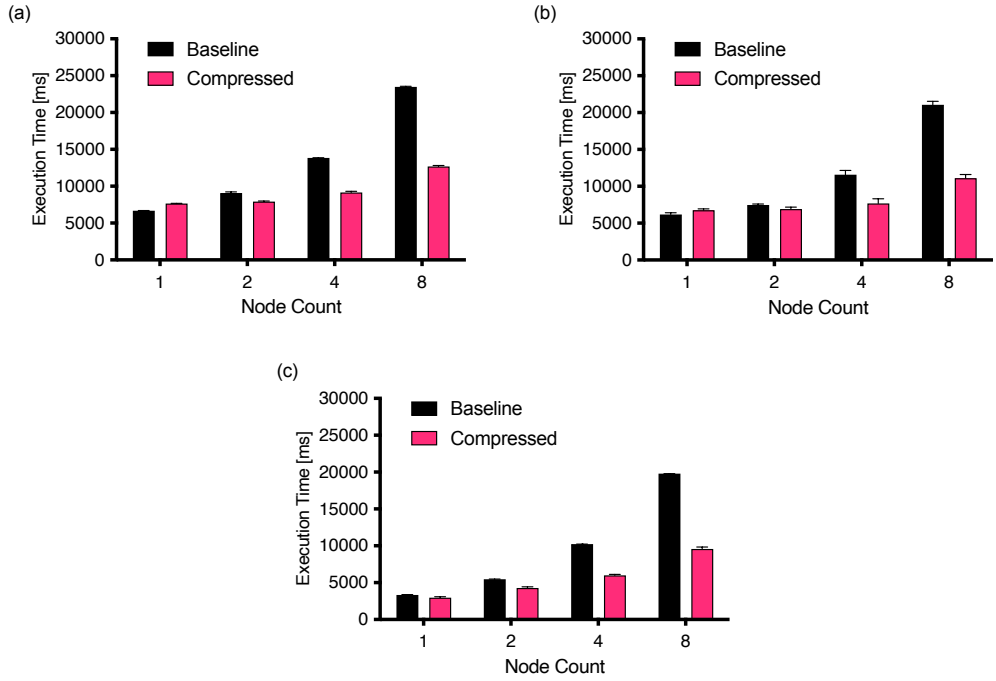

Figure 4.17: The execution time measurements for the *Analytical Database Query* workload are reported in panels (a), (b), and (c) for low, medium and high power compute nodes, respectively. Each node has to process the same volume of $28 * 100000000$ bytes, meaning that a constant execution time across node counts expresses perfect scale-out behavior. Up to a node count of four, this ideal scale-out behavior is approached across all compute node types using on-the-fly data compression.

library. The benchmark is performed using the *Books*, *Wikipedia*, and *OLW* datasets as large text corpora (cf. Section 3.5.2). A simple, computationally expensive but yet powerful implementation strategy is employed that can match any pattern, even non-regular ones. Using this naïve approach also yields a workload that is dominated by compute time instead of data transfer time. Depending on the number of nodes, the first $1000000000 * n_{nodes}$ bytes of an employed data set are transmitted to compute nodes, with each node having to process 1000000000 bytes.

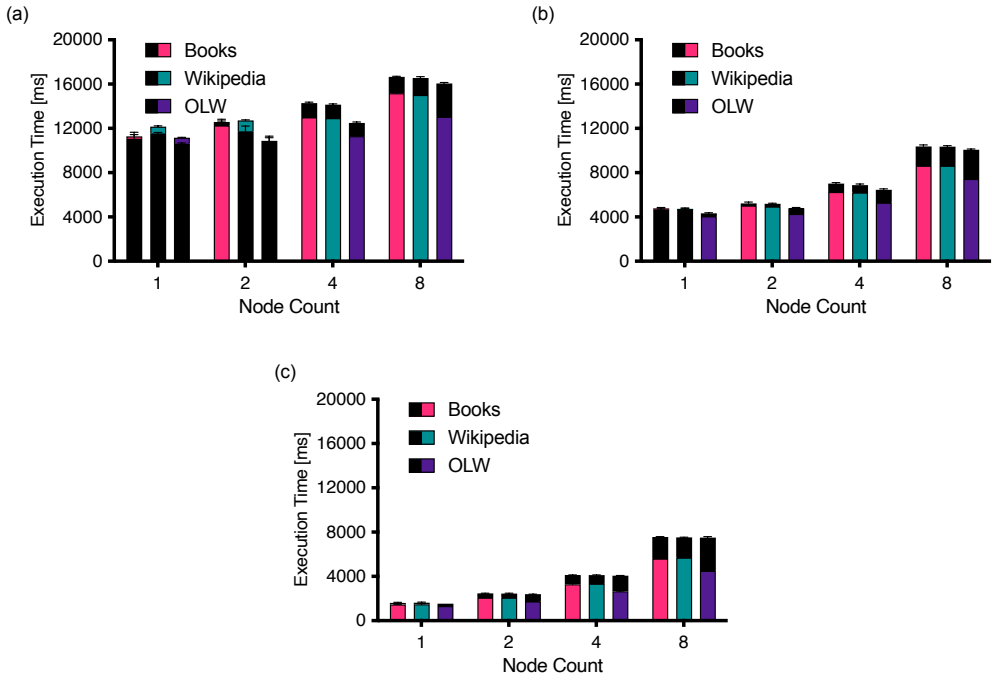

Figure 4.18: Panels (a), (b), and (c) report the execution times measured for the *Text Search* workload on low, medium, and high performance compute nodes, respectively. The black portion of each bar represents the performance baseline of using uncompressed data transfers. The data supports the assumption that this benchmark is dominated by compute time, as the different performance levels of compute nodes can be easily identified. Even though the benefit of on-the-fly data compression for data transfers is not as distinct as in other workloads, compression becomes more beneficial for higher node counts.

Considering the benchmark results provided in Figure 4.18, the first impression might be that compression does not help too much in this use case, especially for faster compute node configurations. However, as mentioned before, this benchmark is more compute-intensive, which can be seen based on the larger performance differences between the different compute node classes. With the tests being less sensitive to data transfer volumes, it is still notable to see that compression yields 1.14×, 1.25×, and 1.43× performance improvements using $n = 8$ low, medium and high performance compute nodes, respectively.

4.6.3.4 Image Downscaler

Last but not least, a simple *Image Downscaler* workload was implemented in C++ using the OpenCL C++ bindings. In this benchmark, a Tag Image File Format (TIFF) image is read and transferred to all available GPUs in the form of an RGBA pixel buffer. To utilized multiple GPUs, the workload is partitioned by segmenting the image horizontally. As reference payloads, the *Curiosity* and *Telescope* datasets are used (cf. Section 3.5.2). In contrast to the other workloads, this test does not clip the datasets proportionally to the number of available nodes n_{nodes}, but the entire image is processed regardless of the employed node count.

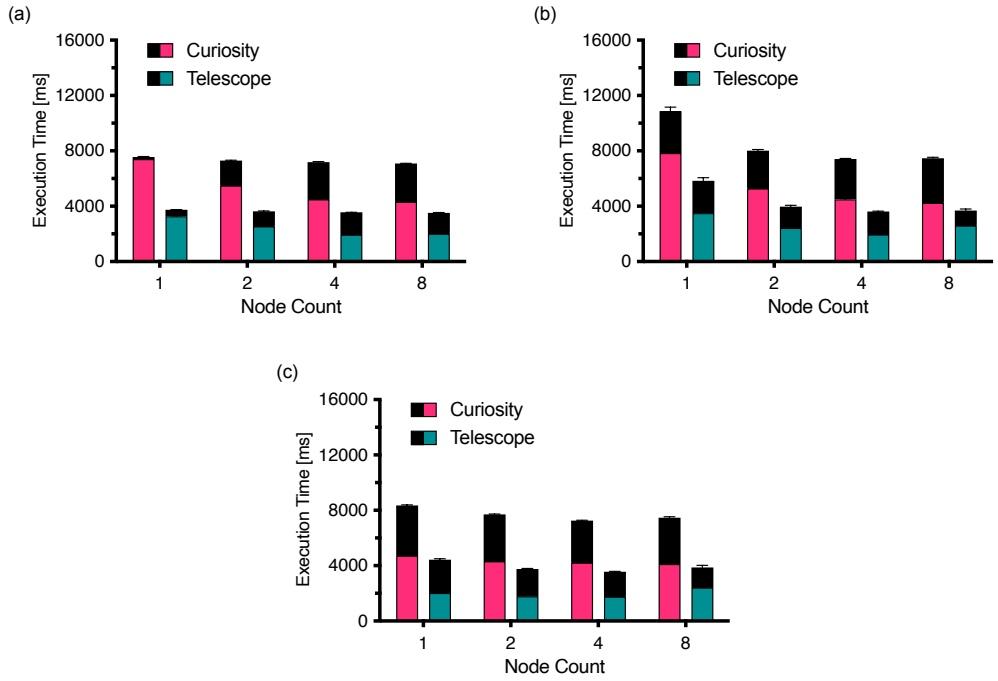

Figure 4.19: Panels (a), (b), and (c) report the execution times measured for the *Image Downscaler* workload on low, medium, and high performance compute nodes, respectively. The black portion of each bar represents the performance baseline of using uncompressed data transfers. The measurements clearly illustrate that the workload is dominated by data transfers, as the execution time does not vary significantly across compute node classes and varying node counts. Here, the use of on-the-fly data compression yields significant performance improvements across most conditions.

The results presented in Figure 4.19 illustrate that this workload is largely dominated by transfer time, as the baseline execution time remains almost constant across varying node counts. With the employed datasets being well compressible, this benchmark makes it easy to gauge the impact of on-the-fly data compression, which improves execution time considerably by up to 1.67×, 1.71×, and 1.89× on low, medium, and high performance compute nodes, respectively. Nevertheless, the test also demonstrates that scalability is ultimately limited by width of the network, even when data is compressed.

4.6.4 Summary

The preceding evaluation has successfully tested the hypothesis that on-the-fly data compression can improve the overall performance of scale-out GPU workloads using various compute node configurations by increasing the effective bandwidth between the master node and compute nodes. Ranging between 1.11× and 2.07×, the performance improvements observed across various workloads may not appear drastic on first sight. However, it should be noted that this speed-up was achieved without assuming any workload-specific knowledge in the compression scheme, without necessitating any modifications in the workloads themselves, or without introducing any other kind of overhead. Considering that the presented approach is capable of introducing even modest speed-up to a very wide range of GPU-based scale-out workloads, performance improvements up to 2.07× appear much more attractive on second sight, especially in the context that the number of applications that require multiple GPUs to satisfy their resource demand is increasing by the day.

4.7 Summary

This chapter has presented two major contributions:

Building up on top of the *dOpenCL* API forwarding library for OpenCL and the *Aparapi* framework for executing native Java code on GPUs, the *CloudCL* framework was introduced. By extending the underlying technologies with a *job* infrastructure including a job scheduler, as well as dynamic scaling capabilities for dynamically available resources, the *CloudCL* framework hides several aspects of the distributed memory parallel programming model during the development of scale-out GPU workloads. These abstractions enable application developers and domain experts to focus on the data parallel programming model associated with GPUs, yielding a single-paradigm development experience which makes scale-out GPU resources more accessible to a wider audience. From an operations point of view, *CloudCL* can also improve resource utilization by disaggregating GPU resources. The improved developer experience provided by *CloudCL* was demonstrated by presenting the *job* class implemented using the *CloudCL* framework for two exemplary workloads.

Serving as the foundation of the *CloudCL* framework, the *dOpenCL* library was augmented with transparent on-the-fly data compression for inter-node data transfers based on the *lib842* compression library presented in Chapter 3. Using a highly pipelined approach to interleave all stages of the workflow for transferring transparently compressed OpenCL buffers from the master node to compute nodes or vice versa, it was possible to improve the effective throughput across nodes. From a workload perspective, the improved data transfer efficiency provided by the integration of transparent compression has yielded performance improvements ranging between 1.11× and 2.07× across various data-intensive scale-out GPU workloads implemented using either the *CloudCL* framework or the OpenCL API directly.

5 Programming Abstractions for Scale-Up Non-Uniform Memory Access Architectures

In this chapter, the *PGASUS* C++ framework is introduced with the goal of alleviating application development for scale-up Non-Uniform Memory Access (NUMA) architectures by providing easy-to-use facilities for memory placement and NUMA-aware task-parallelism. The *PGASUS* framework was originally proposed and extended in the master's theses by Wieland Hagen [68] and Karsten Tausche [198], respectively. Building up on top of the concept and the implementation of the *PGASUS* framework brought forward by these master's theses, the contributions of this work in the context of NUMA architectures are focused on investigating the impact of the programming abstractions provided by *PGASUS* on both the developer experience and performance. The developer experience of the framework is investigated exemplarily based on three different workloads, including a data compression workload that builds up on the *lib842* compression library presented in Chapter 3. Furthermore, a comprehensive evaluation is conducted to investigate the performance-impact of the *PGASUS* framework.

The following master's theses were supervised alongside the research leading to this chapter, fostering scholarly exchange between this work and the supervised theses:

- Patrick Schmidt. "Optimization Guidelines for NUMA Architectures". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Jan. 2016. URL: https://osm.hpi.de/bookshelf/Details/533
- Wieland Hagen. "A Programming Model for C++ Application Development on Non-Uniform Memory Access Architectures". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2016
- Christoph Sterz. "Analyzing NUMA Performance Based on Hardware Event Counters". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2016. URL: https://osm.hpi.de/bookshelf/Details/530
- Kai Fabian. "Measuring and Interpreting NUMA Main Memory Latencies". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Sept. 2017. URL: https://osm.hpi.de/bookshelf/Details/536
- Karsten Tausche. "Memory Management on IBM Power Systems with NUMA Characteristics based on the PGASUS Programming Framework". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Oct. 2017. URL: https://osm.hpi.de/bookshelf/Details/540

Furthermore, partial results of the work presented in this chapter have been published:

- Wieland Hagen, Max Plauth, Felix Eberhardt, Frank Feinbube, and Andreas Polze. "PGASUS: A Framework for C++ Application Development on NUMA architec-

tures". In: *Proceedings of the Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE. Nov. 2016, pages 368–374. DOI: 10.1109/CANDAR.2016.0071

- Max Plauth, Felix Eberhardt, Andreas Grapentin, and Andreas Polze. "Improving the Accessibility of NUMA-Aware C++ Application Development Based on the PGASUS Framework". In: *Concurrency and Computation: Practice and Experience* (Feb. 2022), e6887. DOI: 10.1002/cpe.6887

This chapter is structured as follows. Section 5.1 motivates the demand for programming abstractions that make it easier for developers to exploit data locality in scale-up NUMA systems without disregarding their advantage of providing cache coherency across NUMA domains. After that, Section 5.2 explains why neither the C++ standard library nor operating system Application Programming Interfaces (APIs) such as *libnuma* do not provide suitable means for controlling data placement for C++ objects on NUMA systems. Section 5.3 then introduces the *PGASUS* framework and the facilities it introduces to alleviate NUMA-aware application development in C++. To demonstrate the developer experience of *PGASUS*, Section 5.4 compares *PGASUS*-based implementations of three different workloads with NUMA-aware implementations based on the Open Multi-Processing (OpenMP) interface or a combination of POSIX *pthreads* and the *libnuma* library. The performance impact of the abstractions introduced by *PGASUS* are investigated in a comprehensive evaluation in Section 5.5. Finally, the major insights from this chapter are summarized in Section 5.6.

5.1 Motivation and Problem Statement

Even though Graphics Processing Units (GPUs) have become popular in many data-intensive application domains, many workloads still rely on the flexibility and versatility of multicore Central Processing Units (CPUs) [208]. While several of these CPU-based workloads can be adapted to scale-out across multiple systems to provide sufficient compute resources, certain workloads such as *in-memory databases* [25] or *de Novo genome assembly* [133] are inherently hard to scale out and therefore require as many resources as possible in a single scale-up system.

As elaborated in Section 2.1.1, Uniform Memory Access (UMA) architectures have dominated multiprocessor systems for a long time. From the perspective of an application developer, UMA architectures align conveniently with the shared memory programming model. Unfortunately, sharing the memory subsystem with all other multicore CPUs severely limits the scalability of multiprocessor systems, both in the number of multicore CPUs and in the amount of memory that can be accommodated in a single system.

NUMA architectures avoid this bottleneck, as each multicore CPU is equipped with dedicated memory controllers. Memory attached to other multicore CPUs can still be accessed transparently through inter-CPU interconnects such as Ultra Path Interconnect (UPI), Infinity Fabric (IF), and Power with A-bus, X-bus, OpenCAPI, and NVLink (PowerAXON). However, remote memory access operations incur increased latencies and reduced bandwidth, especially on systems with more than four multicore CPUs where fully meshed connectivity among CPUs is no longer feasible. State-of-the-art NUMA systems support

up to 32 multicore CPUs [76] and 64 TiB of main memory [191] while maintaining a single cache-coherent address space. In the near future, technological novelties such as memory disaggregation are likely to further push the capacities of large NUMA systems [191, 148].

Since remote memory is transparently accessible in cache coherent scale-up NUMA systems, the shared memory parallel programming model is still applicable, enabling application developers can hold on to the programming model they are familiar with. However, performance-critical aspects of NUMA systems such as the distinction between local and remote memory resources are not considered. This is due to the perception of a continuous virtual address space abstracting the heterogeneous memory subsystem. Naturally, locality is covered by the distributed memory parallel programming model, which is employed by the various programming languages and libraries that have been brought forward by the High-Performance Computing (HPC) community. Unfortunately, these distributed memory approaches completely ignore the benefits of a large, cache-coherent scale-up NUMA system. The data parallel programming model, also referred to as the Partitioned Global Address Space (PGAS) model fills the gap between the extremes of the shared memory model and the distributed memory model. A central aspect of the PGAS model is the distinction between local and remote memory resources, which would be the perfect fit for NUMA systems. Even though several programming languages and language extensions based on the PGAS model have been presented, they are all rooted in the HPC community and most are targeting clusters instead of cache-coherent scale-up NUMA systems.

To fill this gap of PGAS-based approaches tailored for scale-up NUMA systems, the *PGASUS* framework has been proposed and improved by Wieland Hagen and Karsten Tausche in their master's theses [68, 198]. *PGASUS* is a C++-based framework that provides easy-to-use facilities for memory placement and NUMA-aware task-parallelism. The framework makes extensive use of the Resource Acquisition is Initialization (RAII) programming idiom [194], which is a powerful concept for managing the resources of a given scope. By embracing the RAII idiom, *PGASUS* makes it easy for developers to specify a memory allocation strategy that serves all allocations of the active scope from a specified *Memory Source* using the *Place Guard* construct. To make it easier for developers to co-locate data and threads, *PGASUS* provides simple NUMA-aware parallel tasking facilities that follow the general concepts of the interfaces for threading and asynchronous calls in C++11 and onwards. However, the impact of using the programming abstractions introduced by *PGASUS* has not been investigated sufficiently. The main contribution of this chapter is that it attempts to fill this gap by conducting a comprehensive evaluation of both the developer experience and the performance impact facilitated by the abstraction mechanisms of *PGASUS*.

5.2 Data Placement in NUMA Systems

For developers intending to factor in the properties of NUMA systems in their applications, several challenges regarding data placement have to be considered during the development of C++ applications. Hereinafter, the implications of object placement and object migration on NUMA-aware application development are identified, where neither

the C++ standard library nor operating system mechanisms provide sufficient means to express data locality on NUMA systems.

5.2.1 Object Placement

The C++ programming language and its standard library do not have any concept for considering data locality in NUMA topologies. As a consequence of the virtual memory abstraction, a flat, homogeneous address space is presented to applications, where regions can be made available through operating system APIs such as *libnuma* [100]. These regions are identified only by location and length, and are otherwise indistinguishable, as application developers are supposed to be indifferent about any details of the underlying hardware. As such, no mechanisms are provided to group data, prevent intra-page fragmentation or otherwise deal with the specific challenges of NUMA topologies, as illustrated in Figure 5.1.

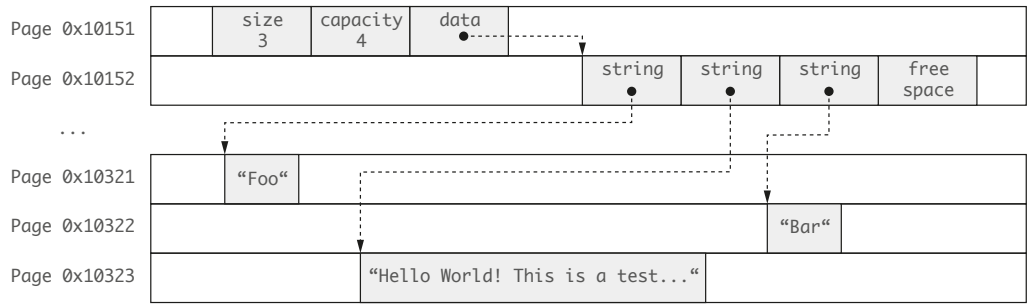

Figure 5.1: Example of how a `std::vector<std::string>` instance and its underlying data structures can be spread across many distinct pages.

To control memory placement decisions at runtime, an application has to either provide a custom implementation of the `new` operator that overrides the default behavior, or memory has to be allocated manually in advance. However, neither method considers that every class used in a context sensitive to object placement needs to be allocated using the modified `new` operation. This cannot be guaranteed for classes that are defined outside of the program such as libraries that may implement custom allocation schemes. Also, many template-based container data-structures rely on the default behavior of `placement new` and implement their own memory management based on `malloc`. Lastly, overwriting `new` has no effect on data-structures implemented in C libraries that use `malloc`.

5.2.2 Object Migration

The issues faced during object migration are very similar to those described for object placement. The page migration mechanisms provided by the operating system for moving data to a different NUMA node does not consider the internal, potentially nested structure of objects. To move an instance of `std::vector<std::string>` to another node for example, not only the pages containing the `std::string` instances have to be moved to the other node, but also the pages holding the string data that are allocated separately on

the heap have to be considered for each `std::string` instance as illustrated in Figure 5.1. In the described scenario, the lack of control over the placement of the `std::vector` and `std::string` data storage is a problem. When more complex objects containing nested object hierarchies have to be migrated using the page migration mechanisms, all associated objects have to be allocated in contiguous memory and occupy a private set of pages that is not shared with other objects.

5.3 PGASUS: NUMA-Aware C++ Application Development

This section provides an overview of the *PGASUS* framework, which has been proposed in the master's thesis by Wieland Hagen [68] and which has been further extended in the course of the master's thesis by Karsten Tausche [198]. *PGASUS* is a C++ framework that employs PGAS semantics on NUMA systems in order to provide developers with the means to specify data placement policies based on the RAII idiom. To furthermore alleviate the co-location of data and tasks, *PGASUS* also provides a simple NUMA-aware tasking infrastructure. The *PGASUS* framework provides five major facilities to alleviate the development of NUMA aware C++ applications: *MemSources* are used to represent logical memory regions that are bound to a specific NUMA nodes and provide the foundation for the concept of *PlaceGuards*, which configure an underlying memory allocator to serve allocations from a specific *MemSource* using the RAII idiom. An interface for discovering the NUMA topology of a system provides the means for developers to respond to the characteristics of a NUMA system at runtime, whereas *NUMA aware Task Parallelism* is used to situate tasks based on the location of the data they are operating on. Finally, *PGASUS* provides a NUMA-aware hash-table to investigate the potential of NUMA-aware drop-in replacements of common data-structures. All five facilities are further detailed hereinafter.

5.3.1 MemSources

The *libnuma* API [100] in Linux provides two methods for influencing the placement of data: Using calls to `numa_alloc_onnode`, memory can be allocated explicitly on the specified node. Memory allocated using this method is always page aligned, which may lead to *internal fragmentation* for small allocations, leaving large parts of the memory in a page potentially unused. For existing code bases to incorporate this method, all allocations have to be replaced with calls to `numa_alloc_onnode` and the size of allocations has to be tracked to replace deallocations with the corresponding `numa_free` call.

Alternatively, using the `numa_set_membind` call implies a contextual approach. This method can be used to specify which NUMA nodes should be used to serve subsequent allocations. Like the other method, `numa_set_membind` also operates on the granularity of pages, which means that the specified placement policy only applies to pages that are faulted into the heap after the `numa_set_membind` call. This behavior can yield *false sharing* effects in situations where an allocator serves small allocations from a page that has been faulted into the heap prior to the call to `numa_set_membind` and therefore may be placed on a different NUMA.

To avoid these pitfalls, *PGASUS* introduces *MemSources* as the central means for representing logical memory regions that are bound to a specific NUMA node. As exemplified in Listing 5.1, *MemSources* can be used to serve allocations for objects of arbitrary size and type. Furthermore, *MemSources* provide a mechanism for conveniently migrating all of its pages to another node, and also making sure that allocations are served from that node thereafter.

Listing 5.1: *MemSources* represent a logical memory region that is bound to a specific NUMA node. The *MemSource* interface can be used to group memory allocations, to control allocation placement, and to migrate groups of objects between NUMA nodes.

```
1 int initialSize = 1 << 24;            // 16 MiB
2 MemSource msource = MemSource::create(targetNode, initialSize);
3
4 Foo *foo = msource->construct<Foo>(); // create object
5 void *buffer = msource->alloc(1024);  // allocate memory
6
7 msource->migrate(newHomeNode);        // migrate pages
```

A `MemSource` maintains a pre-allocated buffer bound to its home node using *libnuma* from which smaller allocations are served using an optimized memory allocator. At first sight, well-known high-concurrency allocators such as *jemalloc* [43] and *TCMalloc* [58] may seem like a good choice from a performance perspective. Considering the *segregated* allocation strategy employed by *jemalloc*, *TCMalloc*, as well as many other high-concurrency allocators, they cannot be applied directly to a single, externally allocated memory block such as the pre-allocated buffer provided by a `MemSource` as *segregated* allocators employ many distinct memory blocks to serve allocations of different size classes. As a result, `MemSources` are implemented using the more conservative *ptmalloc3* [59] allocator employing a *best-fit* allocation strategy. The allocator supports the `mspace_t` construct, which can be created within the pre-allocated buffers provided by the `MemSources`. These pre-allocated buffers and the `mspace_t` constructs therein will be referred to as *Arenas*. Large allocations are allocated directly via `mmap` and are bound to the home node of the `MemSources` using *libnuma*.

For object location querying and, more importantly, object de-allocation, it is imperative that each object allocated from an `MemSource` can be identified. Since the `free` function used for memory de-allocation in C only takes one pointer to the memory chunk, it has to be possible to query the `MemSource` belonging to a memory chunk by just using this pointer. For this purpose a data item called *Footer* is stored before each allocated chunk. This *Footer* contains a pointer to the `MemSource` and the *Arena* that the allocation was made from. For large chunks allocated using `mmap` directly, the arena pointer is set to `NULL` and the footer is extended to also include the block size and links to other mmap-allocated chunks, thus implementing a linked list.

5.3.2 Place Guards

Following the RAII idiom [194], *PGASUS* introduces the `PlaceGuard` construct to control the behavior of the `new` operator and all common memory allocation functions to allocate memory from a *Node* (cf. Listing 5.2) or a *MemSource* (cf. Listing 5.3) instead of using the NUMA-agnostic `malloc` implementation of the C library. After a `PlaceGuard` is created,

its effects are active until the `PlaceGuard` goes out of scope. Their effect can also be nested by specifying additional `PlaceGuard`s within an already guarded scope, allowing for fine-grained control over the allocations of the application, and even of third-party code. When a `PlaceGuard` instance goes out of scope and is deallocated, its effect will end and subsequent allocations will be served by the previously active parent `PlaceGuard`, if any, and otherwise will return to the `malloc` implementation of the C library.

Listing 5.2: `PlaceGuard`s enable developers to easily specify object placement by configuring the behavior of the underlying *stacked malloc* allocator. Both the allocated string object and its underlying string data buffer will be allocated on the specified `targetNode`.

```
1 std::string* createStringOnTargetNone(Node targetNode) {
2     PlaceGuard guard(targetNode);
3     return new std::string("foo");
4     // PlaceGuard leaves scope and loses effect
5 }
```

Listing 5.3: `PlaceGuard`s can also be specified in relation to a previously created `MemSource` instead of a NUMA node. In this case, the data buffer inside the string implementation will be allocated using the given `MemSource`.

```
1 std::string* createStringInMemSource(MemSource source) {
2     PlaceGuard guard(source);
3     return new std::string("foo");
4     // PlaceGuard leaves scope and loses effect
5 }
```

To achieve this behavior, the *PlaceGuard* construct is backed by the *stacked malloc* allocator, which replaces all memory allocation functions defined by the C++ standard including inherited C interfaces as well as specific POSIX interfaces with the behavior described hereinafter. In *stacked malloc*, every thread maintains a stack of places, which can be either references to `MemSources` or NUMA nodes. When the `PlaceGuard` construct is invoked, the specified `MemSource` or NUMA node is pushed onto the stack. Upon leaving scope, the `PlaceGuard` construct removes the corresponding place from the stack. To serve allocations, *stacked malloc* consults the top element of the stack. In case a *MemSource* resides on the top of the stack, allocations are served thereof. When a node is the top element, the default *MemSource* residing on the specified node is used to serve the allocation.

Upon deallocation of a memory chunk, it has to be known from which `MemSource` the memory chunk was allocated. Thus the allocated memory chunks are annotated with information that facilitates a link to the `MemSource` it stems from. It is then possible to ask that `MemSource` to deallocate the given chunk. The basic algorithms for memory allocation and deallocations are shown in Listing 5.4.

Listing 5.4: Simplified operating principle employed by the *stacked malloc* allocator.

```
1 MemSource *getMemSource() {
2     tls = getThreadLocalStorage();
3     places = tls.placeStack;
4     if (!places.empty() && places.top.isMemSource())
5         return places.top.memsource;
```

```
    int node = places.empty() ? localNode : places.top.node;
    if (node == localNode)
        return tls.localMemSource;
    return tls.nodeData.msources[node];
}

static const int offset = sizeof(MemSource::Footer);

void malloc(size_t sz) {
    MemSource *ms = getMemSource();
    return ms->alloc(sz);
}

void free(void *p) {
    void *chunk = p - offset;
    MemSource *ms = *((MemSource**) chunk);
    ms->free(p);
}
```

5.3.3 Topology Discovery

PGASUS provides means for retreiving the topology of a system based on the `Node` and `NodeList` classes outlined in Listing 5.5. The framework creates a model of the machine topology by using the information provided by `hwloc` and `/sys/devices/system/node/node[X]/distance`. Certain hardware configurations can result in NUMA node ids which may not be consecutive. Such situations include hypervisors such as PowerVM, systems with coherently attached accelerator memory, or disaggregated memory setups. To better deal with a non-linear id space, the topology interface employs a logical node mapping using consecutive node ids. Similarly, there are configurations where NUMA nodes may only contain memory resources and no CPU resources. To address the possibility of compute-less NUMA nodes, the helper methods in the `NodeList` class provide additional variants considering only NUMA nodes equipped with CPU resources.

Listing 5.5: *PGASUS* exposes topology information using the `Node` and `NodeList` classes.

```
class Node {
    static Node currentNode();       // node of calling thread
    static Node forCpuId(CpuId id);
    NodeList neighbors();            // sorted by distance
    int physicalId();
    int logicalId();

    vector<int> cpuIds();
    size_t cpuCount() const;
    size_t threadCount() const;
    int indexOfCpuid(CpuId id) const;

    size_t memorySize() const;
    size_t freeMemory() const;
}

class NodeList : public std::vector<Node> {
```

```
18      static NodeList& logicalNodes();      // as detected at runtime
19      static size_t logicalNodesCount();
20
21      static const NodeList& logicalNodesWithCPUs();
22      static size_t logicalNodesWithCPUsCount();
23
24      static int physicalToLogicalId(int physicalId);
25      static size_t physicalNodesCount();
26
27  }
```

5.3.4 NUMA-aware Task-Parallelism

Even though a plethora of very mature parallel tasking libraries are readily available, they are too complex to prototypically incorporate the notion of `MemSources` and the partitions of the global address space they represent. To investigate the developer experience as well as the performance impact of a parallel tasking mechanism that respects the locality aspects of a partitioned global address space, *PGASUS* provides a simple parallel tasking infrastructure. As demonstrated in Listing 5.6, the tasking interface proposed for *PGASUS* follows the general concepts of the interfaces for threading and asynchronous calls in C++11 and onwards. Task functions are defined as a `std::function<T>`, where T is the return value type which may also be `void`. For the specification of a task, a priority level as well as a target NUMA node can be specified that the task should be bound to. In addition to using named functions to define tasks functions, lambda expressions introduced with C++11 may be used to define anonymous task functions. Spawning a task returns a `Future` object that can be used to wait for the task to finish or to obtain the result value. These simpel but versatile tasking facilities enable developers to easily co-locate tasks and data by specifying a target node a task will be bound to, which is typically the home node of a `MemSource` the task should operate on. If no target node is specified, the task is executed on an arbitrary node when no tasks bound to that specific node are available.

Listing 5.6: Spawning tasks using C++ lambdas yields a future object that can be used to wait for the completion of the task. This behavior enables asynchronous and synchronous task parallelism.

```
auto task = numa::async(targetNode, []() {
    std::cout << "Executed on node "
        << Node::current() << std:endl;
    return 42;
}
// do something else in the meantime ...
auto result = task.wait();    // result = 42
```

5.3.5 NUMA-aware Hash Table

To investigate the potential of implementing NUMA-aware drop-in replacement of common data-structures based on *PGASUS*, a NUMA-aware *Hash Table* is provided as a part of the *PGASUS* framework. The custom *Hash Table* was implemented using *PGASUS* and allows an arbitrary amount of concurrent insert, update, read and delete operations.

Developers only have make sure that they only perform delete operations on objects that are not currently read, modified or iterated upon.

The *Hash Table* is divided into 2^N buckets, each of which is resides on a specific NUMA node and is responsible for a part of the index space. The last N bits of a keys hash value are used to identify the bucket responsible for storing that key. Each bucket is furthermore subdivided into 2^M bins. Bins are linked lists that store an arbitrary, but usually a very small amount of key-value pairs.

Synchronization is applied at a very fine-grained level via *Reader-Writer* locks. Bin entries are reference-counted to relax synchronization constraints. The *Hash Table* features a number of hierarchical iterators, allowing the iteration space to be divided into sub-iterations for each node. Automatic parallel iteration over the data-structure is implemented by collecting all bucket iterators for each node. Each worker thread then iterates over iterators from that node. Whenever there are iterators left for remote nodes after all local iterators have been processed, workers start stealing from this remote work pool.

5.4 Developer Experience

A central aspect of this chapter is to demonstrate the developer experience of the *PGASUS* framework. Therefore, this section introduces three carefully selected workloads that are used to demonstrate the capabilities and limitations of the framework: A *Text Histogram* application and a *Data Compression* workload are employed as embarrassingly data parallel workloads using a fine-grained and a coarse-grained task granularity, respectively. Furthermore, a *Datatabase Table Scan* workload is used to represent a more challenging, irregular workload. For the *Text Histogram* and *Data Compression* workloads, three implementations are compared against each other: an implementation entirely based on *PGASUS*, a NUMA-agnostic implementation based on the OpenMP interface, and a NUMA-aware implementation that combines the mature task parallel computing capabilities of OpenMP with the data placement capabilities of *PGASUS*. The *Database Table Scan* workload is presented to compare a C-based implementation based on *pthreads* and *libnuma* with a C++-based implementation based on *PGASUS*.

5.4.1 Text Histogram

Using the massive corpus of public domain text books provided by *Project Gutenberg* [166], an embarrassingly parallel text histogram workload is used as a stress test for the *PGASUS* tasking component for fine-grained tasks. Counting the occurrence of each word, text histograms are computed on a per-book granularity, with each book representing in the order of hundreds of kilobytes of data that needs to be processed. As each book is stored in an individual text file, the proposed interface for each text file is outlined in Listing 5.7.

Listing 5.7: For the *Text Histogram* workload, each book of the *Project Gutenberg*[166] corpus is represented using a `TextFile` object.

```
1 class TextFile {
2   std::string fileName;
3   std::string fileContent;
```

```
4   std::map<std::string, int> wordHistorgram; // occurrences of each word
5
6   void computeHistogram();
7 };
```

To compare the implementation effort for the simplistic parallelization strategy of the *Text Histogram* workload, a NUMA-agnostic OpenMP-based implementation, a NUMA-aware OpenMP-based implementation, and a *PGASUS* implementation are explicated hereinafter. The NUMA-agnostic OpenMP implementation presented in Listing 5.8 serves as a baseline that the NUMA-aware implementations can be compared to.

Listing 5.8: OpenMP-based implementation of the `loadFiles` and `computeHistograms` methods of the *Text Histogram* workload.

```
1 void loadFilesOMP(const std::vector<std::string> &fileNames) {
2   #pragma omp parallel for
3   for (size_t i = 0; i < fileNames.size(); i++) {
4     auto f = std::unique_ptr<TextFile>(new TextFile(fileNames[i]));
5
6     #pragma omp critical(fileaccess)
7     files[fileNames[i]] = std::move(f);
8   }
9 }
10
11 void computeHistogramsOMP(const std::vector<TextFile*> &files) {
12   #pragma omp parallel for
13   for (size_t i = 0; i < files.size(); ++i) {
14     files[i]->computeHistogram();
15   }
16 }
```

During the initialization phase of the *Text Histogram* workload, the books of the *Project Gutenberg* corpus are loaded into main memory. All NUMA-aware implementations employ a simple round robin scheme to evenly distribute the resulting `TextFile` objects across NUMA nodes. The first NUMA-aware implementation outlined in Listing 5.9 combines the mature parallel tasking facilities of OpenMP with the data placement capabilities of *PGASUS*. In the `loadFiles` method, *PGASUS* is used to bind the `nodeStorage` elements to the respective nodes. The *PGASUS* topology interface is used extensively both in the `loadFiles` method as well as the `computeHistograms` method to avoid the complexity of performing topology discovery manually. Assuming the places policy being set to `sockets` upon launch, the NUMA-aware OpenMP-based implementation relies on two nested parallel statements in the `computeHistograms` method, with the outer statement scheduling one master thread per NUMA node. On the level of the nested parallel for statement, the location of each nodes master thread is inherited for the `proc_bind` statement in order to schedule threads for all logical cores on the current node.

Listing 5.9: NUMA-aware OpenMP-based implementation of the `loadFiles` and `computeHistograms` methods of the *Text Histogram* workload.

```
1 void loadFilesPGASOMP(const std::vector<std::string> &fileNames) {
2   const auto& numaNodes = numa::NodeList::logicalNodesWithCPUs();
3   const size_t totalCPUCount = std::accumulate(...)
```

```
4
5   std::vector<std::vector<std::string>> perNodeFileNames(numaNodes.size());
6
7   // Distribute files/jobs to NUMA nodes according to local number of CPU cores
8   const float distFactor = float(fileNames.size()) / totalCPUCount;
9   size_t nextFileName = 0u;
10  for (size_t node = 0; node < numaNodes.size(); ++node) {
11    const size_t localCount = std::ceil(numaNodes[node].cpuCount() * distFactor);
12    for (size_t l = 0; l < localCount && nextFileName < fileNames.size(); ++l,
            ++nextFileName) {
13      perNodeFileNames[node].push_back(fileNames[nextFileName]);
14    }
15  }
16
17  nodeStorages.resize(numa::NodeList::logicalNodesCount());
18
19  #pragma omp parallel proc_bind(spread) num_threads(numaNodes.size())
20  {
21    const auto node = numa::Node::curr();
22    const numa::PlaceGuard placeGuard{ node };
23    const auto nodeId = node.logicalId();
24    nodeStorages[nodeId] = std::unique_ptr<NodeStorage>(new NodeStorage);
25    NodeStorage &nodeStorage = *nodeStorages[nodeId];
26    const auto& localFileNames = perNodeFileNames[nodeId];
27
28    #pragma omp parallel for proc_bind(master) num_threads(node.threadCount())
29    for (size_t i = 0; i < localFileNames.size(); ++i) {
30      const std::string& fileName = localFileNames[i];
31      auto f = std::unique_ptr<TextFile>(new TextFile(fileName));
32      auto fPtr = f.get();
33      #pragma omp critical(fileaccess)
34      {
35        nodeStorage.files.push_back(std::move(f));
36        nodeStorage.filesMap.emplace(fileName, fPtr);
37      }
38    }
39  }
40 }
41
42 void computeHistogramsPGASOMP() {
43  omp_set_nested(1);
44  const auto& numaNodes = numa::NodeList::logicalNodesWithCPUs();
45  #pragma omp parallel proc_bind(spread) num_threads(numaNodes.size())
46  {
47    const auto node = numa::Node::curr();
48    std::vector<const TextFile*> &localFiles
49      = nodeStorages[node.logicalId()]->files;
50    #pragma omp parallel for proc_bind(master) num_threads(node.threadCount())
51    for (size_t i = 0; i < localFiles.size(); ++i) {
52      localFiles[i]->computeHistogram();
53    }
54  }
55 }
```

The *PGASUS*-based implementation exemplified in Listing 5.10 uses the NUMA-aware hash table introduced in Section 5.3.5 to keep the `TextFile` objects balanced across all nodes in the `loadFiles` method. By leveraging the NUMA-aware hash table implementation provided by *PGASUS*, a simple lambda expression in the `computeHistograms` method is sufficient to define tasks which are scheduled for execution on the NUMA node on which the respective `TextFile` object resides. In terms of code complexity, NUMA-awareness based on *PGASUS* can be achieved with minimal effort, whereas the nested parallel statements of the NUMA-aware OpenMP-based implementation are more complex.

Listing 5.10: *PGASUS*-based parallelization of the text histogram workload.

```
numa::HashTable<std::string, std::unique_ptr<TextFile>, 6> files;

virtual void loadFilesPGASUS(const std::vector<std::string> &fileNames) {
  std::list<TriggerableRef> waitList;

  for (const std::string &file : fileNames) {
    waitList.push_back(files.insertAsync(file, [file]() {
      return std::unique_ptr<TextFile>(new TextFile(file));
    }));
  }

  numa::wait(waitList);
}

void computeHistogramsPGASUS(const std::vector<TextFile*> &files) {
  std::list<TriggerableRef> waitList;

  for (const TextFile* : files) {
    waitList.push_back(numa::async<void>([this,file]() {
      files[file]->wordHistogram();
    }, 0, files.where(file).getNode()));
  }
  numa::wait(waitList);
}
```

5.4.2 Data Compression

To complement the fine-grained quality of the *Text Histogram* workload, a *Data Compression* workload is used to provide a coarse-grained task profile. The first 10^9 bytes from the 2006-03-03 Wikipedia dump (*enwik9*)[121] are compressed and decompressed using the 842 compression algorithm [54]. Also belonging to the category of embarrassingly data parallel problems, the *Data Compression* workload employs a coarse-grained task profile with each task compressing or decompressing tens of megabytes of raw data using the highly optimized *lib842* library presented in Chapter 3.

To compare the implementation effort of the parallelized *Data Compression* workload, a NUMA-agnostic OpenMP-based implementation, a NUMA-aware OpenMP-based implementation, and a *PGASUS* implementation are discussed hereinafter. The NUMA-agnostic OpenMP-based implementation of the chunk-wise compression routine of the *lib842* li-

brary outlined in Listing 5.11 serves as a baseline that the NUMA-aware implementations can be compared to.

Listing 5.11: OpenMP-based parallelization of the *Data Compression* workload.

```
#pragma omp parallel for
for (size_t chunkNum = 0; chunkNum < num_chunks; chunkNum++) {
const uint8_t *chunk_in = input_buffer + (chunkNum * CHUNK_SIZE);
uint8_t *chunk_out = compressed_buffer + (chunkNum * CHUNK_SIZE * 2);
...
compress(chunk_in, CHUNK_SIZE, chunk_out, ...);
}
```

During the initialization phase of the *Data Compression* workload, the *enwik9* data set is loaded into main memory. Instead of loading the file into a single buffer, all NUMA-aware implementations split up the file contents into sub-buffers bound to each NUMA node. The NUMA-aware OpenMP-based implementation outlined in Listing 5.12 again combines the mature parallel tasking facilities of OpenMP with the data placement capabilities of *PGASUS* to bind the sub-buffers residing in `input_buffers` and `compressed_buffers` to the respective NUMA nodes. The *PGASUS* topology interface is used in the NUMA-aware OpenMP-based implementation to avoid the complexity of performing topology discovery manually. Like the *Text Histogram* workloads, the places policy has to be set to `sockets` upon launch. This policy is used to schedule one master thread per NUMA node in the outer parallel section. Each nodes master thread then executes a nested parallel for section where the location of each nodes master thread is inherited for the `proc_bind` statement in order to schedule threads for all logical cores on the current node. To make sure that each thread operates on the correct sub-buffers and the respective chunks therein, the NUMA-aware OpenMP-based implementation has to compute the identifiers `nodeId` and `localThreadId` for each thread.

Listing 5.12: NUMA-aware OpenMP-based parallelization of the *Data Compression* workload.

```
omp_set_nested(1);
const auto& numaNodes = numa::NodeList::logicalNodesWithCPUs();

#pragma omp parallel proc_bind(spread) num_threads(numaNodes.size())
{
  #pragma omp parallel proc_bind(master) num_threads(currentNode.threadCount())
  {
    size_t nodeId = numa::Node::curr().logicalId();
    size_t localThreadId = omp_get_thread_num();
    size_t chunkStart = localThreadId * chunks_per_cpu;
    size_t chunkEnd = chunkStart + chunks_per_cpu - 1;
    ...
    for(size_t chunkNum = chunkStart; chunkNum <= chunkEnd; chunkNum++) {
      // buffers are divided into local partitions per node
      const uint8_t *chunk_in = input_buffers[nodeId] + (chunkNum * CHUNK_SIZE);
      uint8_t *chunk_out = compressed_buffers[nodeId] + (chunkNum * (CHUNK_SIZE *
           2));
      ...
      compress(chunk_in, CHUNK_SIZE, chunk_out, compressed_chunk_size);
      }
```

```
20      }
21    }
22  }
```

Similar to the NUMA-aware OpenMP-based implementation, the *PGASUS*-based implementation of the *Data Compression* workload outlined in Listing 5.13 requires some additional effort to compute the identifiers `nodeId` and `localThreadId` each task is operating on. This additional degree of complexity is necessary in order to make sure that each thread operates on the correct sub-buffer elements of `input_buffers` and `compressed_buffers` and the respective chunks therein. Even though these operations require additional complexity compared to a NUMA-agnostic OpenMP-based implementation, they are basic boilerplate operations that can be easily transferred to other workloads. In direct comparison with the NUMA-aware OpenMP-based implementation, *PGASUS* obviates the need for the tedious setup of the nested parallel sections. Considering that a manageable amount of boilerplate code is sufficient to facilitate NUMA-awareness, the slightly increased code complexity should be acceptable. Incorporating the prototypical nature of *PGASUS*, future iterations of the framework may further alleviate the use of the framework by providing thread and node indices through built-in helper functions.

Listing 5.13: *PGASUS*-based parallelization of the *Data Compression* workload.

```
1  std::atomic<size_t> threadIds[numa::NodeList::logicalNodesCount()] = {};
2
3  numa::wait(numa::forEachThread(numa::NodeList::logicalNodesWithCPUs(), [&](){
4    size_t nodeId = numa::Node::curr().logicalId();
5    size_t localThreadId = threadIds[nodeId].fetch_add(1);
6    size_t chunkStart = localThreadId * chunks_per_cpu;
7    size_t chunkEnd = chunkStart + chunks_per_cpu - 1;
8    ...
9    for(size_t chunkNum = chunkStart; chunkNum < chunkEnd; chunkNum++) {
10     // buffers are divided into local partitions per node
11     const uint8_t *chunk_in = input_buffers[nodeId] + (chunkNum * CHUNK_SIZE);
12     uint8_t *chunk_out = compressed_buffers[nodeId] + (chunkNum * CHUNK_SIZE * 2)
           ;
13     ...
14     compress(chunk_in, CHUNK_SIZE, chunk_out, ...);
15   }
16 }, 0));
```

5.4.3 Database Table Scan

The *PRESLEY* benchmark by Felix Eberhardt and Andreas Grapentin [2] implements a *Database Table Scan* workload optionally using an index structure to test the impact of different types of indices on the throughput characteristics of a given workload. Currently implemented are the B-Plus Tree [1] index which is commonly found in conventional relational databases, as well as the Group Key index used in emerging in-memory database systems [47]. The original implementation of the *PRESLEY* benchmark utilizes *pthreads* for parallel execution and *libnuma* for data placement, whereas a new version of the benchmark was implemented in the context of a joint plublication [152] to investigate the developer experience of *PGASUS*, using the framework for both parallel execution and

data placement. In a setup phase, *PRESLEY* creates a main data table of configurable size and uses either *libnuma* or *PGASUS* to place the data on a configurable primary NUMA node. The data is then shuffled using the fisher-yates algorithm [52] to ensure randomness of the data accesses in order to maximize the rate of cache misses. For the configurations relevant in the context of this work, the data is then indexed using one of the implemented index types. The index is either placed on the primary node or is replicated across all NUMA nodes in the system. After the setup phase, the benchmark performs lookups on the data table either by accessing the non-replicated or the replicated index in parallel on all CPUs avaialable in the system.

The version of the benchmark implemented using *pthreads* and *libnuma* outlined in Listing 5.14 contains topology detection functionality to determine how many NUMA nodes and cores are available in the system to place the threads and data accordingly. Both the functionality of *pthreads* and *libnuma* is used to assign affinities to threads and to bind memory allocations on the desired NUMA nodes in order to implement the data placement and index replication. This manual approach proved to be more difficult to implement correctly and has several other drawbacks compared to the implementation of the benchmark based on the *PGASUS* framework as well.

Listing 5.14: Sequential generation of replicated index data across NUMA nodes based on *libnuma*.

```
for (size_t i = 0; i < topology.nodes.n; ++i) {
  // wrapper to numa_membind_to_node
  topology_membind_to_node(topology.nodes.nodes[i].num);

  // explicitly allocate on correct node to avoid reusing existing heap pages
  struct index_t *index =
    numa_alloc_onnode(sizeof(*index), topology.nodes.nodes[i].num);
  memset(index, 0, sizeof(*index));

  populate_data_index(index);
  data_index[topology.nodes.nodes[i].num] = index;
}

// wrapper to numa_membind_to_node
topology_release_membind();
```

In the *PGASUS*-based version of the benchmark demonstrated in Listing 5.15, topology information is already provided by the *PGASUS* framework and the manual topology detection is no longer needed. The *PGASUS* tasking functionality is used to execute the enclosed lambda function once on each NUMA node. *PGASUS* partitions the heap into separately managed spaces per NUMA node, thus operating on an allocation granularity as opposed to the page granularity implemented by *libnuma*. This means that using the *PlaceGuard* construct, the application has fine control over the allocations of third party code without modifications, while also avoiding the internal fragmentation or false sharing problems outlined in Section 5.3.1. The *PlaceGuard* construct ensures that all allocations in the current scope are placed on correct NUMA node. Since the *PlaceGuard* construct loses its effect when it goes out of scope, there is no need to manually undo the memory binding as it is the case in the *pthreads* and *libnuma* implementation. After making the

placement decision using the *PlaceGuard* construct, the remainder of the code uses regular calls to `malloc` and the `new` operator to setup the data-structures.

Listing 5.15: Parallel generation of replicated index data across NUMA nodes based on the *PlaceGuard* and tasking functionality of *PGASUS*.

```
numa::wait(numa::onceForEachNode(numa::NodeList::logicalNodesWithCPUs(), [&]() {
  numa::PlaceGuard mguard(numa::Node::curr());

  struct index_t *index = new struct index_t;
  memset(index, 0, sizeof(*index));

  populate_data_index(index);
  data_index[current.logicalId()] = index;

  return 0;
}, 0));
```

5.4.4 Summary

Across the various workloads demonstrated in the preceding section, the *PGASUS*-based implementations were always less complex compared to the NUMA-aware implementations based on OpenMP or the *pthreads* library. With the programming abstractions provided by *PGASUS* reducing the complexity of NUMA-aware application development, the framework accomplishes its goal of unburdening developers. In terms of simplicity of code, only the NUMA-agnostic implementations can surpass the *PGASUS*-based implementations at the cost of completely ignoring the heterogeneity of the memory resources in NUMA systems.

5.5 Performance Evaluation

The goal of the evaluation presented in this section is investigate the performance impact of the *PGASUS* framework using the workloads discussed in Section 5.4. To achieve this goal Section 5.5.1 specifies all relevant details of the testing environment and the basic benchmark procedures to make the evaluation more repeatable. As the performance of memory allocations can have a big impact on the overall performance of workloads, a synthetic benchmark is presented in Section 5.5.2 which compares the memory allocation performance of the *stacked malloc* allocator provided by *PGASUS* to *ptmalloc3*, *jemalloc*, and *TCMalloc*. The performance measurements yielded for the workloads discussed in Section 5.4 are presented in Section 5.5.3. Finally, Section 5.5.5 summarizes central findings of the evaluation.

5.5.1 Testing Environment & Benchmark Procedure

All hardware configurations used for the evaluation of *PGASUS* are documented in Table 5.1. The set of employed machines covers the range from common two socket configurations up to high-end eight socket configurations. To analyze the behavior of different

processor designs and Instruction Set Architectures (ISAs) under different workloads, configurations using x86_64-based AMD EPYC and Intel Xeon CPUs and ppc64le-based IBM POWER8 CPUs are included.

Table 5.1: Specifications of the test systems used to evaluate the performance impact of *PGASUS*.

	Tyan	**S824L**
Model	Tyan TN83-B8251 [130]	IBM Power System S824L [81]
CPU	2×AMD EPYC 7282, 2.80 GHz, 16C	2×IBM POWER8 (Murano), 3.42 GHz, 10C/80T each
Memory	256 GB DDR4 ECC, 3200 MHz	1024 GB DDR3 ECC, 1600 MHz
OS	Ubuntu 18.04.5	Ubuntu 20.04.1
Kernel	5.4.0	5.4.0
Compiler	GCC 7.5.0	GCC 10.2.1 (AT 14.0)
	DL560	**E880**
Model	HPE ProLiant DL560 Gen10 [74]	IBM Power System E880 [82]
CPU	4×Intel Xeon Gold 6148, 2.40 GHz, 20C each	8×IBM POWER8 SCM (Turismo), 4.00 GHz, 12C each
Memory	1536 GB DDR4 ECC, 2666 MHz	6144 GB DDR3 ECC, 3200 MHz
OS	Ubuntu 18.04.5	Ubuntu 18.04.4
Kernel	4.15.0	5.3.0
Compiler	GCC 7.5.0	GCC 10.2.1 (AT 14.0)

All performance measurements presented hereinafter were performed after a fresh reboot in order to ensure a clean system state. Furthermore, no other active users or background tasks were running on the involved servers. In order to retrieve a sufficiently meaningful dataset, each benchmark was executed 30 times. Error bars are used in all plots to report the standard deviation for each measurement. As an additional measure, simultaneous multithreading was disabled on all systems to reduce the variance of the measurements. Furthermore, each benchmark was preceded by a warm-up run in order to eliminate any confounding factors.

5.5.2 Memory Allocation Performance

In this section, a synthetic benchmark is used to investigate the performance of the *stacked malloc* allocator provided by *PGASUS*. The performance of *stacked malloc* is compared to *ptmalloc3*, *jemalloc*, as well as *TCMalloc*. The *ptmalloc3* allocator is used by PGASUS internally to serve small allocations from the *arenas*. Therefore, *ptmalloc3* serves as measure for the overhead introduced by *PGASUS* itself. Finally, *jemalloc* and *TCMalloc* are included as state-of- the-art high-concurrency memory allocators that are currently used by major software companies.

Measuring the exact time spent in a single `malloc` or `free` call is generally not possible, as the measurement would introduce too much overhead compared to the duration of the

call itself. Therefore, the duration of a large number of `malloc/free` cycles is measured as demonstrated in Listing 5.16 on the IBM Power System S824L [81].

Listing 5.16: Conceptual code of the `malloc/free` benchmark. The volatile pointer variable prevents the compiler from optimizing out the inner loop.

```
for (size_t size = 512; size <= 1024*1024*1024; size *= 2) {
  const auto start = std::chrono::high_resolution_clock::now();
  for (size_t i = 0; i < repetitions; ++i) {
    volatile ptr = malloc();
    free(ptr);
  }
  const auto end = std::chrono::high_resolution_clock::now();
  timings[size] = duration(start, end) / repetitions;
}
```

Due to the repeated, equally sized allocations directly followed by deallocations, the benchmarking setup advantages caching allocators. Such allocators will serve most allocations with reused memory blocks, without requiring an actual system memory allocation. This setup seemed reasonable, as most real-work applications generate characteristic memory access patterns on limited ranges of block sizes [36]. Therefore, it is safe to assume that a well designed caching memory allocator will cover access patterns of realistic applications in most cases.

Furthermore, this benchmark does not access allocated memory. Thus, as far as not accessed by the allocator itself, memory pages are only virtually, but not physically allocated. This, however, is not true for *PGASUS*, which uses *libnuma* to bind every allocated memory page to a specific NUMA node. In Linux, memory bindings and policies are only applied once a page is physically allocated.

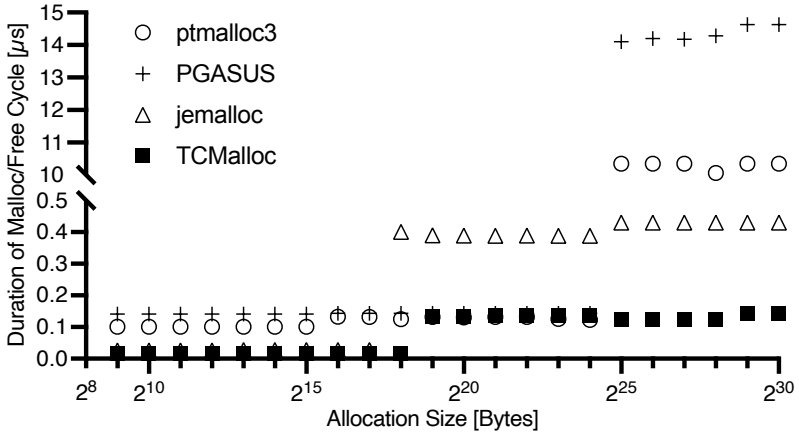

Figure 5.2: Duration of a single `malloc/free` cycle. Measurements were performed for data quantities matching multiples of two, ranging from 512 bytes up to 1 GiB.

An overview of the timings retreived for all allocators is presented in Figure 5.2. At the border between small and large allocations, the duration of a `malloc/free` cycle of *ptmalloc3*, and *PGASUS* increase by 81× and 99×, respectively. Compared to that, *jemalloc*

and *TCMalloc* have a substantially higher performance and much smaller variations across different allocation sizes. At the border between small and large allocations, the duration of a `malloc/free` cycle increases by 15× and 8×, respectively.

For small allocations, *PGASUS* relies on *ptmalloc3* for serving allocations from its *arenas*. As the performance pf *PGASUS* is widely similar to that of *ptmalloc3* for small allocations, it appears as if the overhead introduced by the *stacked malloc* allocator of *PGASUS* is minimal. Regarding large allocations, *PGASUS* is the slowest allocator in comparison. However, it should be noted that *PGASUS* handles large allocations itself in lists of `mmap`-allocated chunks, so that the internally used *ptmalloc3* is not involved at this point. As mentioned before, each large allocation request in *PGASUS* also results in physical allocation and a call to *libnuma* in order to bind the allocated pages to the requested NUMA node. *ptmalloc3* serves large allocations approximately 40% faster than *PGASUS* *jemalloc* and *TCMalloc* implement caching even for large allocations. They use a central page heap implemented based on red-black trees [43] and free lists [58], respectively.

5.5.3 Workload Benchmarks

The main goal of this section is to demonstrate that the use of *PGASUS* can yield performance improvements across a wide range of workloads, including a *Text Histogram* workload, a *Data Compression* workload, and a *Database Table Scan* use case. All workloads benchmarked throughout this section are available in the implementations discussed in Section 5.4. The throughput measurements presented in this section are reported as average values including standard deviation ($n = 30$). For a statistically meaningful evaluation of the collected throughput data, t-tests are performed to assess statistical significance. To further verify that changes in throughput are caused by improved data locality, the performance counters `PM_DATA_FROM_LMEM` (data cache loaded from local memory), `PM_DATA_FROM_RMEM` (data cache loaded from remote memory), and `PM_DATA_FROM_DMEM` (data cache loaded from distant memory) are recorded for 10 repeated executions of each workload on the IBM Power System S824L [81]. Based on this data, the ratio between remote memory access and local memory access (*RMA/LMA*) is computed.

5.5.3.1 Text Histogram

The *Text Histogram* workload employs a very fine-grained task profile, as a task computes the word frequency histograms for one of the 64192 books curated by *Project Gutenberg* [166] by the end of 2020. Each book is stored in a dedicated `.txt` file, with an average file size of 360833 bytes. The total data volume processed by this workload amounts to 21.57 GiB. All three implementations discussed in Section 5.4.1 are used for the evaluation.

As illustrated in Figure 5.3, *PGASUS* achieves between 1.09× and 4.7× performance improvements for the fine-grained per-file task profile compared to the *OMP* baseline implementation. With performance improvements between 0.2× and 5.9×, the *OpenMP+libnuma* implementation yields mixed results, surpassing the performance improvements of *PGASUS* on x86_64-based systems and providing similar or worse performance POWER8-based systems. For each hardware configuration, a t-test has confirmed statistically significant ($p < 0.000001$) performance impact of the respective implementations compared to the *OMP* baseline. *RMA/LMA* ratios of 0.164, 0.024, and 0.005 were

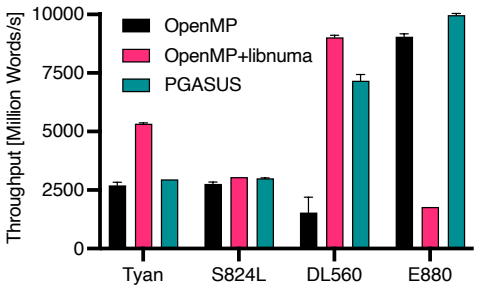

Figure 5.3: The throughput measurements (higher is better) yielded by the *Text Historgram* workload exhibit increased throughput using *PGASUS* compared to the *OpenMP* baseline across all hardware configurations. However, the superior performance of the *OpenMP+libnuma* implementation demonstrates what speed-up might be possible if the simple tasking facilities of *PGASUS* handled the fine-grained task profile more efficiently.

determined for the *OpenMP*, the *OpenMP+libnuma*, and the *PGASUS* implementations, respectively, confirming considerably improved locality for the NUMA-aware implementations. Additional performance profiling sessions have revealed that the simplistic NUMA-aware parallel tasking component of *PGASUS* is overwhelmed by the very fine-grained task profile of the text histogram workload. Furthermore, the huge drop of performance of the *OpenMP+libnuma* implementation on the IBM Power System E880 is caused by an excessive amount of time spent in the OpenMP implementation `libgomp.so`. However, the reason for this behavior could not be identified.

5.5.3.2 Data Compression

Unlike the preceding workload, the *Data Compression* workload exhibits a much more coarse-grained task profile. Each task processes multiple megabytes of data (1 GB / number of CPU cores), performing complex operations. The measurements are performed using the large text compression benchmark [121] as a payload, which is comprised of the first 10^9 bytes from the 2006-03-03 Wikipedia dump. All three implementations discussed in Section 5.4.2 are used for the evaluation. Separate measurements are performed for the compression operation and the decompression operation.

The benchmark results documented in Figure 5.4 demonstrate that *PGASUS* provides performance improvements across all hardware configurations. For the compression operation, the *PGASUS*-based implementation achieves performance improvements between 1.08× and 1.75×. On the side of the decompression operation, the framework yields between 1.02× and 1.54× performance improvements compared to the baseline implementation. The *OpenMP+libnuma* implementation yields improvements in compression throughput ranging between 0.33× and 1.18×. Surprisingly, the same implementation results in consistent slowdowns for the decompression operation, delivering throughput ranging between 0.27× and 0.935× of the *OMP* baseline performance. While additional profiling sessions have not identified the source of the consistent slowdown in decompression performance, the huge drop of performance of the *OpenMP+libnuma* implementation on the IBM Power System E880 could be traced back to an excessive amount of time spent

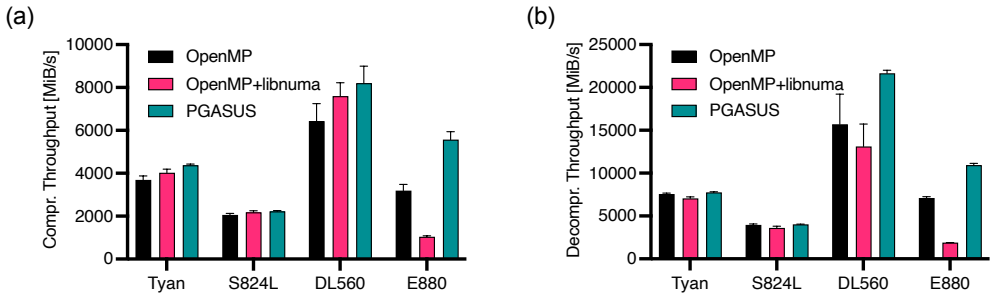

Figure 5.4: The throughput measurements (higher is better) for compression and decompression are reported in panels (a) and (b), respectively. For both operations, employing *PGASUS* provides performance improvements across all hardware configurations, with the compression operation experiencing slightly higher speed-up factors compared to the decompression operation. The *OpenMP+libnuma* version also demonstrates performance improvements for the compression operation, although not as marked as the *PGASUS* implementation. Surprisingly, the *OpenMP+libnuma* implementation fails to exceed the *OpenMP* baseline performance for decompression.

in the OpenMP implementation `libgomp.so`. For each hardware configuration, a t-test has confirmed statistically significant ($p < 0.000001$) performance improvements of the *PGASUS*-based implementation as well as the *OpenMP+libnuma* implementation compared to the *OMP* baseline. Finally, *RMA/LMA* ratios of 8.476, 2.680, and 2.771 were determined for the *OMP*, the *OpenMP+libnuma*, and the *PGASUS* implementations, respectively, confirming considerably improved locality for the NUMA-aware implementations.

5.5.3.3 Database Table Scan

The *PRESLEY* benchmark [2] implements a *Database Table Scan* workload looking for a value in the primary key column of a database table. Scan-threads corresponding to the number of logical cores in the employed machines are used, and the table has a single column with $1,000,000,000$ unique integer values.

In the experiments, the throughput of the original implementation based on *pthreads* and *libnuma* is compared to a *PGASUS*-based implementation. Furthermore, two different configurations are compared for each implementation, with the first configuration performing an index-based scan using a B+ tree residing on a single NUMA node and the second configuration replicating the B+ tree across each NUMA node of the employed system. The lookup of the search-values in the tree resembles a pointer-chasing-based workload. This type of workload is latency-bound since every cacheline is only touched briefly and another cacheline, possibly in a distant location is accessed next. Therefore, varying memory latencies in a NUMA system are relevant.

Using both implementations discussed in Section 5.4.3, the replicated configuration eliminates almost all remote memory access operations, yielding *RMA/LMA* ratios of 0.0041 and 0.0001 for the *pthreads+libnuma* and *PGASUS* implementations, respectively. In comparison, the non-replicated configuration yields *RMA/LMA* ratios of 12.6515 and 4.6707 for the *pthreads+libnuma* and *PGASUS* implementations, respectively. The low *RMA/LMA* ratios of the replicated configurations are well reflected by the throughput measurements

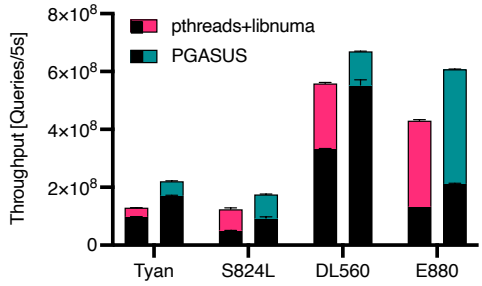

Figure 5.5: The throughput measurements (higher is better) yielded by the *PRESLEY* benchmark indicate improved performance across all hardware configurations when the B+ index is replicated across all NUMA nodes. In all configurations, the *PGASUS*-based implementation outperforms the *pthreads+libnuma* baseline implementation.

presented in Figure 5.5, yielding performance improvements between 1.293× and 3.23× for the replicated configurations in comparison to the non-replicated case. These improvements come at the price of the increased memory footprint caused by the replicated index copies. However, for irregular random access patterns this leads to a significant performance improvement and could very well be viable in situation where memory is abundant. Regarding the performance impact of the respective implementations, the *PGASUS*-based implementation yields performance improvements ranging between 1.19× and 1.83× compared to the *pthreads+libnuma* implementation. For each hardware configuration, a t-test has confirmed statistically significant ($p < 0.000001$) performance improvements.

5.5.4 Energy Demand Analysis

Even though analyzing the energy-efficiency is not a central concern of this work, the impact of NUMA-aware application development on the energy demand was briefly investigated using the *Data Compression* workload. The energy draw measurements for a compression/decompression-cycle of the *enwik9* data set [121] were performed using all three implementations on the *IBM Power System S824L* test system using two Microchip MCP39F511N dual-channel power measurement devices [129] and the *PINPOINT* [104] utility. Since these measurements cover the entire execution of the test application, the compression and decompression cycle was repeated 30 times in the test application in order to reduce the impact of setup, data transfers, and teardown on the overall energy draw measurements. From these measurements, the idle power draw of the test system is deducted in order to only report the share of energy demand caused by the compression and decompression process itself. The results illustrated in Figure 5.6 demonstrate that both NUMA-aware implementations provide considerably improved energy efficiency, using less than half of the energy required by the OpenMP-based, NUMA-agnostic implementation of the *Data Compression* workload.

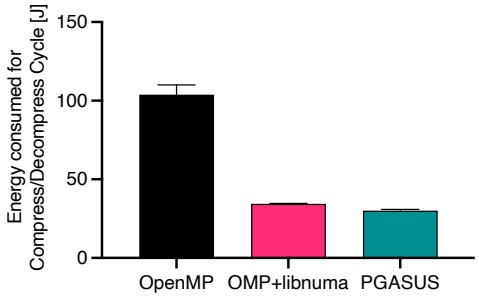

Figure 5.6: For a compression/decompression-cycle of the *enwik9* test dataset, both NUMA-aware implementations consume less than half of the energy required by the NUMA-agnostic OpenMP implementation.

5.5.5 Summary

The comprehensive evaluation conducted in this section has successfully demonstrated that *PGASUS* offers performance improvements compared to the *OpenMP* or *pthreads+libnuma* baseline implementations across all evaluated workloads. For the fine-grained task profile of the *Text Histogram* workload, the simplistic tasking facilities of *PGASUS* are overstrained and cannot provide the same level of performance improvements compared to the *OpenMP+libnuma* implementation on x86_64 based systems. In additional profiling sessions, the busy waiting locks in the *PGASUS* task scheduler were identified as a potential bottleneck. Therefore, it should be possible to improve the performance of *PGASUS* for fine-grained task profiles with some additional optimization work. Both for the *Data Compression* workload and the *Database Table Scan* workload, the use of *PGASUS* yielded notable performance improvements compared to the *OpenMP* or *pthreads+libnuma* baseline implementations. With average performance improvements of 1.56× and peak performance improvements of up to 4.67×, the evaluation demonstrated that *PGASUS* does not only improve the developer experience across all workloads, but that it also capable of outperforming the baseline implementations. Even though energy measurements were only performed for the *Data Compression* workload, it can be assumed that NUMA-aware implementatons of the remaining workloads consume less energy as well. In addition to the improved performance provided by the *PGASUS* framework, the reduced energy demand provides one more reason to make data-intensive applications NUMA-aware.

5.6 Summary

The *PGASUS* framework used in this chapter applies the concept of providing explicit means for distinguishing between different memory partitions from the PGAS model and makes it available to C++ application developers targeting shared memory systems based on the RAII idiom. Even though the *PGASUS* framework has been presented in the master's theses by Wieland Hagen [68] and Karsten Tausche [198], both theses have focused their evaluation efforts on micro-benchmarking aspects of the framework itself. To fill this gap, this chapter contributed a comprehensive evaluation based on three

exemplary workloads. First, the improved developer experience offered by the framework was demonstrated by comparing *PGASUS*-based implementations of all three workloads to NUMA-agnostic as well as NUMA-aware implementations based on OpenMP or the *pthreads* library. For the evaluation of performance aspects, test systems based on both the x86_64 and ppc64le ISAs and ranging from 2 to 8 socket configurations were employed. On these systems, the evaluation demonstrated that *PGASUS* does not only improve the developer experience across all workloads, but that it is also capable of outperforming NUMA-agnostic implementations with average performance improvements of 1.56× and peak performance improvements of up to 4.67×.

6 Discussion and Outlook

In this chapter, the achievements of this work are summarized, and the individual contributions presented in this thesis are reviewed. The chapter reviews how the contributions presented in this thesis address the problem statement formulated in Section 1.2. This chapter also discusses the limitations of the individual approaches and outlines ideas for future research based on the contributions of this thesis.

6.1 Overview

This thesis contributes mitigations to the challenges formulated in Section 1.2 by investigating programming abstractions for on-chip accelerators, off-chip accelerators, and non-uniform memory resources. For each type of heterogeneous resource, one programming abstraction mechanism is presented and evaluated. Chapter 3 introduces the *lib842* compression library. The library does not only make the resources of the *NX-842* compression accelerator accessible from user space, but also introduces the first freely accessible user-space implementations of software-based compression and decompression facilities for Central Processing Units (CPUs) as well as Graphics Processing Unit (GPU)-based decompression facilities. Chapter 4 introduces the *CloudCL* framework with the goal of hiding many aspects of distributed computing during the development of scale-out GPU workloads. To improve the scalability of the framework, the compression facilities of the *lib842* compression library are used to implement transparent compression for data transfers. Targeting Non-Uniform Memory Access (NUMA) systems, Chapter 5 builds up on top of the *PGASUS* framework for NUMA-aware C++ application development brought forward by Wieland Hagen [68] and Karsten Tausche [198]. The chapter contributes a comprehensive evaluation of the impact of the programming abstractions provided by *PGASUS* on both developer experience and application performance.

6.2 Contributions and Future Research

In this section, the contributions presented in this thesis are reviewed and put in perspective with related abstraction mechanisms outlined in Section 2.3. Furthermore, potential starting points for future research efforts are identified, ranging from undertakings that build up on top of the contributions presented in this work to aspects that have not been investigated in the scope this work.

Lib842 Compression Library The *lib842* compression library presented in this thesis is the first user-space approach for providing compression and decompression facilities based on the proprietary *842* compression algorithm. Relying on a modified version of the `cryptodev-linux` out-of-tree kernel module, the implementation details for making the high-throughput and low-latency compression and decompression facilities of *NX-842* on-chip compression accelerators accessible to user-space applications through *lib842* are discussed. To enable compressed data exchange across heterogeneous system resources, the hardware-accelerated approach is complemented with the introduction of highly optimized software-based compression and decompression routines for CPUs as well as Open Computing Language (OpenCL)-based decompression facilities for arbitrary GPUs. In contrast to other approaches that only employ memory compression techniques on the isolated scope of the memory resources attached to either a CPU or a GPU, the *lib842* compression library lays out the groundwork for exchanging data across heterogeneous system resources in compressed form. Another distinctive feature of the employed *842* algorithm is that it provides decent compression ratios across a wide range of workloads (cf. Section 3.5.2) and does not rely on characteristics of specific use cases [86, 93].

To further improve the interoperability of the *lib842* library across heterogeneous system resources, an obvious choice would be to extend the library with GPU-based compression. Furthermore, implementations for additional resource types such as Field-Programmable Gate Arrays (FPGAs) should be included in future revisions of the library. Finally, assuming future optimization efforts can manage to improve the compression and decompression throughput of accelerator-based implementations to levels comparable to the *NX-842*, it might even be possible to improve the efficiency of intra-node data transfers based on compression.

CloudCL Framework for Single-Paradigm Scale-Out GPU Computing The *CloudCL* framework presented in this thesis joins together the *dOpenCL* Application Programming Interface (API) forwarding library [90] for OpenCL and the *Aparapi* framework [6] for executing native Java code on GPUs. By extending the underlying technologies with a *job* infrastructure including a job scheduler, as well as dynamic scaling capabilities for dynamically available resources, the *CloudCL* framework hides several aspects of the distributed memory parallel programming model during the development of scale-out GPU workloads. These abstractions enable application developers to focus on the data parallel programming model associated with GPUs, yielding a single-paradigm development experience which makes scale-out GPU resources more accessible to a wider audience. With the uniform developer experience enabled based on the job infrastructure and the support for adding or removing cluster resources dynamically at runtime, the *CloudCL* framework provides several distinguishing features compared to the use of plain API forwarding approaches [90, 95, 7] that create the illusion of local resources.

To further foster this illusion of local resources, the *dOpenCL* library is augmented with transparent on-the-fly data compression for data transfers based on the *lib842* compression library in order to improve the efficiency of data transfers between the master node and compute nodes. Using a highly pipelined approach to interleave all stages of the workflow for transferring transparently compressed OpenCL buffers from the master node to a compute node or vice versa, it is possible to improve the effective throughput

across nodes. From a workload perspective, the improved data transfer efficiency provided by the integration of transparent compression yielded performance improvements ranging between 1.11× and 2.07× across various data-intensive scale-out GPU workloads implemented using either the *CloudCL* framework or the OpenCL API directly.

In its current form, the manual definition of independent workload partitions can be considered as one of the biggest limitations of the *CloudCL*. Therefore, future revisions of the framework shall investigate semi-automatic approaches such as the concept of meta-functions employed by the *DistCL* library [38]. Finally, another limitation of *dOpenCL* and therefore also the *CloudCL* framework is the lack of peer-to-peer communication among compute nodes, making device-to-device data transfers prohibitively expensive. Therefore, extending *dOpenCL* with support for peer-to-peer communication might open up *CloudCL* for workloads that require inter-device communication.

PGASUS Framework for NUMA-aware data-placement in C++ To investigate the impact of using non-uniform memory resources to its fullest potential, this thesis builds up on top of the *PGASUS* framework which been originally presented in the master's theses by Wieland Hagen [68] and Karsten Tausche [198]. The *PGASUS* framework applies the concept of providing explicit means for distinguishing between different memory partitions from the Partitioned Global Address Space (PGAS) model (cf. Section 2.3.3.3) and makes it available to C++ application developers targeting shared memory systems based on the Resource Acquisition is Initialization (RAII) idiom [194]. The major contribution of this thesis to *PGASUS* is that it provides a comprehensive evaluation of the framework based on three exemplary workloads. First, the improved developer experience offered by the framework is demonstrated by comparing *PGASUS*-based implementations of all three workloads to NUMA-agnostic as well as NUMA-aware implementations based on the Open Multi-Processing (OpenMP) API or the *pthreads* library. For the evaluation of performance aspects, test systems based on both the x86_64 and ppc64le Instruction Set Architectures (ISAs) and ranging from 2 to 8 socket configurations were employed. On these systems, the results of the evaluation suggest that *PGASUS* does not only improve the developer experience across all workloads, but that it is also capable of outperforming NUMA-agnostic implementations with average performance improvements of 1.56× and peak performance improvements of up to 4.67×.

PGASUS provides a rewarding alternative to the approaches for enabling NUMA-aware memory placement discussed in Section 2.3.3. Compared to implementing NUMA-aware applications based on OpenMP, *PGASUS* provides significant improvements in terms of developer experience, enabling them to specify data placement policies with distinctly fewer lines of code. In contrast to *polymorphic allocators* [70], *PGASUS* can transparently influence memory placement of nested data structures without having to modify them in order to make use of *polymorphic allocators*. Finally, *PGASUS* eliminates issues such as unintended inter-page fragmentation and false sharing, which can easily occur when operating system facilities such as *libnuma* [100] or *AutoNUMA* [34] are not used correctly.

In its current implementation, the simplistic tasking facilities of *PGASUS* leave room for improvements. With further optimizations, the combination of NUMA-aware data placement and task scheduling offers a lot of potential for NUMA-aware application development. Even though *PGASUS* already provides certain advantages on today's

NUMA systems for coarse-grained task profiles, the author speculates that abstractions for data placement such as *PGASUS* will become vital to deal with the increasing diversity of memory resources in upcoming state-of-the-art computer architectures, as outlined in Section 2.2.

6.3 Review of Research Question

This section reviews the research question of this thesis (cf. Section 1.3), which seeks for programming abstractions that improve the accessibility of heterogeneous system resources for application developers. For this goal, two hypotheses are constructed: First, it is assumed that a certain degree of the complexity conditioned by the large variety of heterogeneous system resources considered in the context of this thesis can be encapsulated using programming abstractions without obscuring performance-critical system properties. Second, it is presumed that programming abstractions can help to mitigate the performance penalty associated with data transfers across heterogeneous system resources.

The implementations of the contributions presented in this thesis as well as their respective evaluation results demonstrate that programming abstractions can be used to make various heterogeneous system resources more accessible. For the *NX-842* on-chip compression accelerators, exposing their resources to user space through the means of a software library makes them usable for applications in the first place. To test the hypothesis for the *CloudCL* framework as well as the *PGASUS* framework, the showcase of the developer experience of both frameworks demonstrates that they manage to reduce the code complexity necessary to make use of scale-out GPU resources and NUMA systems, respectively.

Similarly, the contributions presented in this thesis show how programming abstractions can contribute to mitigating the performance penalty associated with data transfers. With the tightly integrated on-chip connectivity of the *NX-842* on-chip compression accelerator, data transfers are hardly a bottleneck for this type of heterogeneous system resource. However, the efficient use of the hardware-based compression facilities lays out the groundwork for improving the efficiency of data transfers that cannot be avoided. As such, the transparent integration of on-the-fly compression for inter-node data transfers in *CloudCL* confirms that programming abstractions can help to improve the efficiency of data transfers across heterogeneous system resources. Even more distinctive, the evaluation of the *PGASUS* framework demonstrates that programming abstractions for data placement can avoid unnecessary data transfers, delivering considerable performance improvements.

In summary, the results of this thesis show that programming abstractions can indeed be used to improve the accessibility of heterogeneous system resources for application developers. However, to make efficient use of these abstractions, developers have to provide a decent understanding of the underlying hardware characteristics.

7 Conclusion

Application developers bear a certain responsibility of leveraging the heterogeneous system resources available in state-of-the-art computer architectures. The proper use of heterogeneous resources does not only facilitate sustained performance improvements over the years, but it is also vital to improve the energy-efficiency of workloads across all application domains. Unfortunately, the heterogeneity of today's state-of-the-art computer architectures is confronting application developers with an immense degree of complexity which can be ascribed to two major challenges. First, developers need to acquire profound knowledge about the programming models or the interaction models associated with each type of heterogeneous system resource to make efficient use thereof. Second, developers must take into account that heterogeneous system resources always need to exchange data with each other in order to work on a problem together. However, this data exchange is always associated with a certain amount of overhead, which is why the amounts of data exchanged should be kept as low as possible. To respond to these challenges, application developers cannot and should not expect tools like their compilers to take over the responsibility of making efficient use of heterogeneous system resources. However, application developers should also not be overwhelmed with the immense complexities that are implied by state-of-the-art computer architectures.

Standing on the shoulders of giants, this thesis has contributed to the state of the art in heterogeneous computing by presenting programming abstractions that lessen these burdens for three types of heterogeneous system resource. The *lib842* compression library provides the first method for accessing the compression and decompression facilities of the *NX-842* on-chip compression accelerator available in IBM Power CPUs from user space applications running on Linux. Addressing application development of scale-out GPU workloads, the *CloudCL* framework makes the resources of GPU clusters more accessible by hiding many aspects of distributed computing while enabling application developers to focus on the aspects of the data parallel programming model associated with GPUs. Furthermore, *CloudCL* is augmented with transparent data compression facilities based on the *lib842* library in order to improve the efficiency of data transfers among cluster nodes. The improved data transfer efficiency provided by the integration of transparent data compression yields performance improvements ranging between 1.11× and 2.07× across four data-intensive scale-out GPU workloads. To investigate the impact of programming abstractions for data placement in NUMA systems, a comprehensive evaluation of the *PGASUS* framework for NUMA-aware C++ application development is conducted. On a wide range of test systems, the evaluation demonstrates that *PGASUS* does not only improve the developer experience across all workloads, but that it is also capable of outperforming NUMA-agnostic implementations with average performance improvements of 1.56×. For the contributed programming abstractions, this thesis has demonstrated that they can indeed improve the accessibility of heterogeneous system resources by reducing the code complexity in terms of lines of code necessary to make

use of the respective resources without obscuring performance-critical system properties. Furthermore, the presented abstractions also help developers to reduce the amount of data that has to be exchanged among heterogeneous system resources, improving both the effective throughput and the energy efficiency of data transfers.

Fueled by the competition of coherent next-generation interconnection standards, the performance of both inter-node and intra-node interconnection technologies is finally catching up in upcoming computer architectures. In the light of the diversifying memory resources enabled by these novel interconnection standards, programming abstractions for data placement probably may have the brightest perspective for gaining traction. As such, the characteristics of diversifying memory resources might be exploited with the goal of improving energy efficiency based on memory placement decisions.

Bibliography

[1] *A minimal but extreme fast B+ tree indexing structure demo for billions of key-value storage*. https://github.com/begeekmyfriend/bplustree. (accessed 2021-04-01). 2015.

[2] *A minimal but extreme fast B+ tree indexing structure demo for billions of key-value storage*. https://github.com/osmhpi/presleybench. (accessed 2022-05-02). 2020.

[3] Bulent Abali, Bart Blaner, John J. Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. "Data Compression Accelerator on IBM POWER9 and z15 Processors: Industrial Product". In: *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pages 1–14. DOI: 10.1109/ISCA45697.2020.00012.

[4] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. "An Evaluation of Directory Schemes for Cache Coherence". In: *Proceedings of the 15th Annual International Symposium on Computer Architecture*. ISCA '88. Honolulu, Hawaii, USA: IEEE Computer Society Press, 1988, pages 280–298. ISBN: 0818608617.

[5] J. Andrews and N. Baker. "Xbox 360 System Architecture". In: *IEEE Micro* 26.2 (2006), pages 25–37. DOI: 10.1109/MM.2006.45.

[6] *Aparapi Repository*. (Website).

[7] A. Barak and A. Shiloh. *The VirtualCL (VCL) Cluster Platform*. White Paper. Rachel and Selim Benin School of Computer Science, 2014.

[8] Matthias Bastian. "Entwurf und Integration eines Frameworks zur Einhaltung nutzerdefinierter Policies in OpenStack". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Jan. 2017. URL: https://osm.hpi.de/bookshelf/Details/457.

[9] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing Locality and Independence with Logical Regions". In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12*. Edited by Jeffrey K. Hollingsworth. Salt Lake City, UT, USA: IEEE/ACM, Nov. 2012, page 66. DOI: 10.1109/SC.2012.71.

[10] Yannick Bäumer. "Hardware Accelerated Lossless Compression using High-Level Synthesis". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Nov. 2019. URL: https://osm.hpi.de/bookshelf/Details/538.

[11] Jossekin Beilharz. "Koordinierungssprachen — von NUMA-Knoten bis zu Cloud-Verbünden". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Oct. 2016.

[12] Jossekin Beilharz, Frank Feinbube, Felix Eberhardt Eberhardt, Max Plauth, and Andreas Polze. "Claud: Coordination, Locality And Universal Distribution". In: *Proceedings of the Parallel Computing Conference 2015 (PARCO)*. Sept. 2015, pages 605–614. DOI: 10.3233/978-1-61499-621-7-605.

[13] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. "Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle". In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pages 1–12.

[14] Lawrence Benson, Fabian Paul, Christian Werling, and Fabian Windheuser. "Real-time Power Monitoring for Heterogenous Data Centers". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Mar. 2019.

[15] Arijit Biswas. "Sapphire Rapids". In: *IEEE Hot Chips 33 Symposium (HCS)*. Palo Alto, CA, USA: IEEE, Aug. 2021, pages 1–22. DOI: 10.1109/HCS52781.2021.9566865.

[16] Filip Blagojevic, Paul Hargrove, Costin Iancu, and Katherine A. Yelick. "Hybrid PGAS Runtime Support for Multicore Nodes". In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS) 2010*. Edited by José E. Moreira, Costin Iancu, and Vijay A. Saraswat. New York, NY, USA: ACM, Oct. 2010, page 3. DOI: 10.1145/2020373.2020376.

[17] Bart Blaner, Bülent Abali, Brian M. Bass, Suresh Chari, Ronald N. Kalla, Steven R. Kunkel, Kenneth Lauricella, Ross Leavens, John J. Reilly, and Peter A. Sandon. "IBM POWER7+ Processor On-Chip Accelerators for Cryptography and Active Memory Expansion". In: *IBM Journal of Research and Development* 57.6 (2013). DOI: 10.1147/JRD.2013.2280090.

[18] Pawel Böning, Philipp Gampe, and Leonard Geier. "Power-Based Workload Classification". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Mar. 2021.

[19] Yuval Borenstein. *Choosing the Right Speed for Your Leaf-Spine Data Center Network*. (Website). 2020.

[20] T. Brewer. "A highly scalable system utilizing up to 128 PA-RISC processors". In: *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway*. 1995, pages 133–140. DOI: 10.1109/CMPCON.1995.512376.

[21] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. "ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures". In: *International Journal of Parallel Programming* 38.5-6 (2010), pages 418–439. DOI: 10.1007/s10766-010-0136-3.

[22] Joan Bruguera Micó. "Improved Data Transfer Efficiency for Scale-Out GPU Workloads using On-the-Fly I/O Link Compression". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2020. URL: https://osm.hpi.de/bookshelf/Details/539.

[23] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. "Brook for GPUs: Stream Computing on Graphics Hardware". In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pages 777–786. ISSN: 0730-0301. DOI: 10.1145/1015706.1015800.

[24] Alberto Cano. "A survey on graphic processing unit computing for large-scale data mining". In: *WIREs Data Mining and Knowledge Discovery* 8.1 (2018), e1232. DOI: 10.1002/widm.1232.

[25] Rick Cattell. "Scalable SQL and NoSQL data stores". In: *SIGMOD Record* 39.4 (2011), pages 12–27. DOI: 10.1145/1978915.1978919.

[26] Inc. Cavium. *ThunderX Family of Workload Optimized Processors*. (Product Brief). 2016.

[27] Inc. CCIX™ Consotrium. *An Introduction to CCIX™*. White Paper. 2018.

[28] Bradford L. Chamberlain. "A Brief Overview of Chapel". In: *Pre-Print* (2013).

[29] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: an Object-Oriented Approach to Non-Uniform Cluster Computing". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Edited by Ralph E. Johnson and Richard P. Gabriel. San Diego, CA, USA: ACM, Oct. 2005, pages 519–538. DOI: 10.1145/1094811.1094852.

[30] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. "NVIDIA A100 Tensor Core GPU: Performance and Innovation". In: *IEEE Micro* 41.2 (2021), pages 29–35. DOI: 10.1109/MM.2021.3061394.

[31] Hewlett Packard Enterprise Company. *HPE Demonstrates Worlds First Memory-Driven Computing Architecture*. https://www.hpe.com/us/en/newsroom/press-release/2017/03/hewlett-packard-enterprise-demonstrates-worlds-first-memory-driven-computing-architecture.html. (Press Release). Nov. 2016.

[32] James Connolly. "Honeywell rolls out Unix line with three 68000-based minis". In: *Computerworld* 20.39 (Sept. 1986), pages 10–10. ISSN: 0010-4841.

[33] CXL™ Consortium. *Compute Express Link*. White Paper. 2019.

[34] Jonathan Corbet. *AutoNUMA: the other approach to NUMA scheduling*. (Website). 2012.

[35] Data General Corporation. *AViiON Enterprise Servers*. 1997.

[36] Diego Costa and Rivalino Matias Jr. "Characterization of Dynamic Memory Allocations in Real-World Applications: An Experimental Study". In: *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 2015, pages 93–101. DOI: 10.1109/MASCOTS.2015.28.

[37] Zarka Cvetanovic. "Performance Analysis of the Alpha 21364-Based HP GS1280 Multiprocessor". In: *SIGARCH Comput. Archit. News* 31.2 (May 2003), pages 218–229. ISSN: 0163-5964. DOI: 10.1145/871656.859643.

[38] Tahir Diop, Steven Gurfinkel, Jason Helge Anderson, and Natalie D. Enright Jerger. "DistCL: A Framework for the Distributed Execution of OpenCL Kernels". In: *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. San Francisco, CA, USA: IEEE Computer Society, Aug. 2013, pages 556–566. DOI: 10.1109/MASCOTS.2013.77.

[39] Michael Doggett. "Xenos: XBox360 GPU". In: (Nov. 2005). (accessed 2022-05-02).

[40] Jake Edge. *A Generic Hash Table*. (Website). 2012.

[41] Tarek A. El-Ghazawi and Lauren Smith. "UPC: Unified Parallel C". In: *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*. Tampa, FL, USA: ACM Press, Nov. 2006, page 27. DOI: 10.1145/1188455.1188483.

[42] Nick England. "A Graphics System Architecture for Interactive Application-Specific Display Functions". In: *IEEE Computer Graphics and Applications* 6.1 (1986), pages 60–70. DOI: 10.1109/MCG.1986.276612.

[43] Jason Evans, Dave Watson, Qi Wang, and David Goldblatt. *jemalloc memory allocator*. http://jemalloc.net. (accessed 2022-05-02).

[44] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. "PixelFlow: The Realization". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS '97. Los Angeles, California, USA: Association for Computing Machinery, 1997, pages 57–68. ISBN: 0897919610. DOI: 10.1145/258694.258714.

[45] Kai Fabian. "Measuring and Interpreting NUMA Main Memory Latencies". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Sept. 2017. URL: https://osm.hpi.de/bookshelf/Details/536.

[46] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. "Sequoia: Programming the Memory Hierarchy". In: *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*. Tampa, FL, USA: ACM Press, Nov. 2006, page 83. DOI: 10.1145/1188455.1188543.

[47] Martin Faust, David Schwalb, Jens Krueger, and Hasso Plattner. "Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance." In: *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. Edited by Rajesh Bordawekar and Christian A Lang. Istanbul, Turkey, Aug. 2012, pages 13–22. URL: http://www.adms-conf.org/faust%5C_adms12.pdf.

[48] Frank Feinbube. "Ansätze zur Integration von Beschleunigern ins Betriebssystem". PhD thesis. University of Potsdam, Germany, 2018. URL: https://d-nb.info/1168437628.

[49] Frank Feinbube, Max Plauth, Christian Kieschnick, and Andreas Polze. "Evolving Scheduling Strategies for Multi-Processor Real-Time Systems". In: *Proceedings of the 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. July 2015, pages 57–62. URL: https://www.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-p57.pdf.

[50] Frank Feinbube, Jan-Arne Sobania, Peter Tr"oger, and Andreas Polze. "Hybrid Parallel Light-Weight Programming of Hybrid Systems". In: *Parallel and Cloud Computing* 1 (2 2012).

[51] Rosa Filgueira, Malcolm Atkinson, Alberto Nuñez, and Javier Fernández. "An Adaptive, Scalable, and Portable Technique for Speeding Up MPI-Based Applications". In: *Euro-Par 2012 Parallel Processing*. Edited by Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pages 729–740. ISBN: 978-3-642-32820-6.

[52] Ronald Aylmer Fischer and Frank Yates. "Statistical Tables for Biological, Agricultural, and Medical Research". In: Oliver and Boyd, London, 1938.

[53] The Wikimedia Foundation. *Dump of Articles on the English Wikipedia from 2020-03-01*. https://ftp.acc.umu.se/mirror/wikimedia.org/dumps/enwiki/20200301/. (Website). 2020.

[54] Peter A. Franaszek, Luis A. Lastras-Montaño, Song Peng, and John T. Robinson. "Data Compression with Restricted Parsings". In: *Data Compression Conference (DCC'06)*. IEEE, Mar. 2006, pages 203–212. DOI: 10.1109/DCC.2006.22.

[55] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. "Adaptive Loss-Less Data Compression Method Optimized for GPU Decompression". In: *Concurrency and Computation: Practice and Experience* 29.24 (Dec. 2017), e4283. DOI: 10.1002/cpe.4283.

[56] Edward Gehringer, Janne Abullarade, and Michael H. Gulyn. "A Survey of Commercial Parallel Processors". In: *SIGARCH Comput. Archit. News* 16.4 (Sept. 1988), pages 75–107. ISSN: 0163-5964. DOI: 10.1145/54331.54338.

[57] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors". In: *SIGARCH Comput. Archit. News* 18.2SI (May 1990), pages 15–26. ISSN: 0163-5964. DOI: 10.1145/325096.325102.

[58] Sanjay Ghemawat and Paul Menage. *TCMalloc : Thread-Caching Malloc*. https://gperftools.github.io/gperftools/tcmalloc.html. (accessed 2022-05-02).

[59] Wolfram Gloger. *ptmalloc*. http://www.malloc.de/en/. (accessed 2022-05-02).

[60] Andreas Grapentin, Max Plauth, and Andreas Polze. "MemSpaces: Evaluating the Tuple Space Paradigm in the Context of Memory-Centric Architectures". In: *Proceedings of the Fifth International Symposium on Computing and Networking (CANDAR)*. IEEE. Nov. 2017, pages 284–290. DOI: 10.1109/CANDAR.2017.55.

[61] Chris Gregg and Kim M. Hazelwood. "Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer". In: *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2011, 10-12 April, 2011, Austin, TX, USA*. IEEE Computer Society, 2011, pages 134–144. DOI: 10.1109/ISPASS.2011.5762730.

[62] Erik Griese, Leon Matthes, and Maximilian Stiede. "Save Energy by Monitoring Workload Memory Utilization". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Mar. 2022.

[63] Peripheral Component Interconnect Special Interest Group. *PCI Express Base Specification Revision 1.0a*. Apr. 2003.

[64] Peripheral Component Interconnect Special Interest Group. *PCI Express Base Specification Revision 2.0*. Dec. 2007.

[65] Peripheral Component Interconnect Special Interest Group. *PCI Express Base Specification Revision 3.0*. Nov. 2010.

[66] Peripheral Component Interconnect Special Interest Group. *PCI Express Base Specification Revision 4.0*. Oct. 2017.

[67] Felix Grzelka. "On the Energy Consumption of Deep Learning Workloads". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2021. URL: https://osm.hpi.de/bookshelf/Details/529.

[68] Wieland Hagen. "A Programming Model for C++ Application Development on Non-Uniform Memory Access Architectures". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2016.

[69] Wieland Hagen, Max Plauth, Felix Eberhardt, Frank Feinbube, and Andreas Polze. "PGASUS: A Framework for C++ Application Development on NUMA architectures". In: *Proceedings of the Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE. Nov. 2016, pages 368–374. DOI: 10.1109/CANDAR.2016.0071.

[70] Pablo Halpern. "Polymorphic Memory Resources". In: *C++ Standards Committee Working Group ISOCPP* (2013).

[71] J. Hennessy, M. Heinrich, and A. Gupta. "Cache-coherent Distributed Shared Memory: Perspectives on its Development and Future Challenges". In: *Proceedings of the IEEE* 87.3 (1999), pages 418–429. DOI: 10.1109/5.747863.

[72] Benedict Herzog, Timo Hönig, Wolfgang Schröder-Preikschat, Max Plauth, Sven Köhler, and Andreas Polze. "Bridging the Gap: Energy-efficient Execution of Software Workloads on Heterogeneous Hardware Components". In: *Proceedings of the Tenth ACM International Conference on Future Energy Systems*. June 2019, pages 428–430. DOI: 10.1145/3307772.3330176.

[73] Hewlett Packard Enterprise. *HPE ProLiant DL390 Gen9 Server QuickSpecs*. https://www.hpe.com/h20195/v2/GetDocument.aspx?docname=c04346247. (Website). 2019.

[74] Hewlett Packard Enterprise. *HPE ProLiant DL560 Gen10 Server QuickSpecs*. https://www.hpe.com/h20195/v2/GetDocument.aspx?docname=a00021850enw. (Website). 2020.

[75] Hewlett Packard Enterprise. *HPE ProLiant m710p Server Cartridge QuickSpecs*. https://www.hpe.com/h20195/v2/GetDocument.aspx?docname=c04760473. (Website). 2016.

[76] Hewlett Packard Enterprise. *HPE Superdome Flex QuickSpecs*. https://www.hpe.com/h20195/v2/GetDocument.aspx?docname=a00026242enw. (Website). 2021.

[77] Jens Hiller, Maël Kimmerlin, Max Plauth, Heikkilä Seppo, Stefan Klauck, Ville Lindfors, Felix Eberhardt, Dariusz Bursztynowski, Jesus Llorente Santos, Oliver Hohlfeld, and Klaus Wehrle. "Giving Customers Control Over Their Data: Integrating a Policy Language into the Cloud". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018, pages 241–249. DOI: 10.1109/IC2E.2018.00050.

[78] H. Peter Hofstee. *Distributed Memory on POWER 10*. https://www.clsac.org/uploads/5/0/6/3/50633811/clsac-2020-hofstee.pdf. (accessed 2022-05-02). Oct. 2020.

[79] Timo Hönig and Andreas Polze. *Memento: Energy-Efficient Memory Placement*. Project Proposal. 2021.

[80] Intel. *Intel Rack Scale Design Architecture*. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf. (White Paper). 2018.

[81] International Business Machines Corporation. *IBM Power System S824L Technical Overview and Introduction*. https://www.redbooks.ibm.com/redpapers/pdfs/redp5139.pdf. (Website). 2014.

[82] International Business Machines Corporation. *IBM Power Systems E870 and E880 Technical Overview and Introduction*. (Website). 2014.

[83] Joefon Jann, Paul Mackerras, John Ludden, Michael Gschwind, Wade Ouren, Stuart Jacobs, Brian F. Veale, and David Edelsohn. "IBM POWER9 System Software". In: *IBM Journal of Research and Development* 62.4/5 (2018), 6:1–6:10. DOI: 10.1147/JRD.2018.2846959.

[84] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pages 947–960. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/jeon.

[85] Luke Anthony Kachelmeier, Faith Virginia Van Wig, and Kari Natania Erickson. *Comparison of High Performance Network Options: EDR InfiniBand vs. 100Gb RDMA Capable Ethernet*. Technical report. Los Alamos National Laboratory (LANL), 2016.

[86] Krzysztof Kaczmarski and Piotr Przymus. "Fixed Length Lightweight Compression for GPU Revised". In: *Journal of Parallel and Distributed Computing* 107 (2017), pages 19–36. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2017.03.011.

[87] T. Kalaiselvi, P. Sriramakrishnan, and K. Somasundaram. "Survey of using GPU CUDA programming model in medical image analysis". In: *Informatics in Medicine Unlocked* 9 (2017), pages 133–144. ISSN: 2352-9148. DOI: 10.1016/j.imu.2017.08.001.

[88] Rubasri Kalidas, Mayank Daga, Konstantinos Krommydas, and Wu-chun Feng. "On the Performance, Energy, and Power of Data-Access Methods in Heterogeneous Computing Systems". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. IEEE Computer Society, 2015, pages 871–879. DOI: 10.1109/IPDPSW.2015.131.

[89] Supun Kamburugamuve, Pulasthi Wickramasinghe, Saliya Ekanayake, and Geoffrey C. Fox. "Anatomy of Machine Learning Algorithm Implementations in MPI, Spark, and Flink". In: *International Journal of High Performance Computing Applications* 32.1 (2018), pages 61–73. DOI: 10.1177/1094342017712976.

[90] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch. "dOpenCL: Towards Uniform Programming of Distributed Heterogeneous Multi-/Many-Core Systems". In: *Journal of Parallel and Distributed Computing* 73.12 (2013), pages 1639–1648. DOI: 10.1016/j.jpdc.2013.07.021.

[91] Marvin Keller, Philipp Pajak, Florian Rösler, and Robert Schäfer. "Scalable and Secure Infrastructures for Cloud Operations". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Mar. 2016.

[92] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. "The AMD Opteron Processor for Multiprocessor Servers". In: *IEEE Micro* 23.2 (2003), pages 66–76. DOI: 10.1109/MM.2003.1196116.

[93] M. Khavari Tavana, Y. Sun, N. Bohm Agostini, and D. Kaeli. "Exploiting Adaptive Data Compression to Improve Performance and Energy-Efficiency of Compute Workloads in Multi-GPU Systems". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2019, pages 664–674. DOI: 10.1109/IPDPS.2019.00075.

[94] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. "Achieving a Single Compute Device Image in OpenCL for Multiple GPUs". In: *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. Edited by Calin Cascaval and Pen-Chung Yew. San Antonio, TX, USA: ACM, Feb. 2011, pages 277–288. DOI: 10.1145/1941553.1941591.

[95] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. "SnuCL: an OpenCL Framework for Heterogeneous CPU/GPU Clusters". In: *International Conference on Supercomputing, ICS'12*. Edited by Utpal Banerjee, Kyle A. Gallivan, Gianfranco Bilardi, and Manolis Katevenis. Venice, Italy: ACM, 2012, pages 341–352. DOI: 10.1145/2304576.2304623.

[96] Maël Kimmerlin, Peer Hasselmeyer, Seppo Heikkilä, Max Plauth, Paweł Parol, and Pasi Sarolahti. "Network Expansion in OpenStack Cloud Federations". In: *2017 European Conference on Networks and Communications (EuCNC)*. June 2017, pages 1–5. DOI: 10.1109/EuCNC.2017.7980655.

[97] Maël Kimmerlin, Max Plauth, Seppo Heikkilä, and Tapio Niemi. "A Practical Evaluation of a Network Expansion Mechanism in an OpenStack Cloud Federation". In: *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. 2017, pages 1–6. DOI: 10.1109/CloudNet.2017.8071540.

[98] David Kirk. "NVIDIA CUDA Software and GPU Parallel Computing Architecture". In: *Proceedings of the 6th International Symposium on Memory Management*. ISMM '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pages 103–104. ISBN: 9781595938930. DOI: 10.1145/1296907.1296909.

[99] Stefan Klauck, Max Plauth, Sven Knebel, Marius Strobl, Douglas Santry, and Lars Eggert. "Eliminating the Bandwidth Bottleneck of Central Query Dispatching Through TCP Connection Hand-Over". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019)*. Edited by Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klet-

tke, and Holger Meyer. Gesellschaft für Informatik, Bonn, 2019, pages 97–106. DOI: 10.18420/btw2019-07.

[100] Andi Kleen. *libNUMA — NUMA Policy Library*. Linux Manpage. (accessed 2022-05-02). 2008.

[101] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation". In: *Parallel Computing* 38.3 (2012), pages 157–174. DOI: 10.1016/j.parco.2011.09.001.

[102] Sven Knebel. "Interfaces for New Networking Challenges". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, June 2018. URL: https://osm.hpi.de/bookshelf/Details/528.

[103] Sven Köhler. "On-Chip Accelerators on POWER8". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, May 2017. URL: https://osm.hpi.de/bookshelf/Details/531.

[104] Sven Köhler, Benedict Herzog, Timo Hönig, Lukas Wenzel, Max Plauth, Jörg Nolte, Andreas Polze, and Wolfgang Schröder-Preikschat. "Pinpoint the Joules: Unifying Runtime-Support for Energy Measurements on Heterogeneous Systems". In: *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. IEEE. Nov. 2020, pages 31–40. DOI: 10.1109/ROSS51935.2020.00009.

[105] Sven Köhler, Lukas Wenzel, Max Plauth, Pawel Böning, Philipp Gampe, Leonard Geier, and Andreas Polze. "Recognizing HPC Workloads Based on Power Draw Signatures". In: *Ninth International Symposium on Computing and Networking (CANDAR)*. Matsue, Japan: IEEE, Nov. 2021, pages 278–284. DOI: 10.1109/CANDARW53999.2021.00053.

[106] Robert C Kunz. "Performance Bottlenecks on Large-Scale Shared-Memory Multiprocessors". PhD thesis. Stanford University, 2005.

[107] N. Kurd, P. Mosalikanti, M. Neidengard, J. Douglas, and R. Kumar. "Next Generation Intel® Core™ Micro-Architecture (Nehalem) Clocking". In: *IEEE Journal of Solid-State Circuits* 44.4 (2009), pages 1121–1129. DOI: 10.1109/JSSC.2009.2014023.

[108] BBN Laboratories. *Butterfly Parallel Processor Overview*. BBN Report no. 6148. Cambridge, MA, Mar. 1986.

[109] E. Scott Larsen and David McAllister. "Fast Matrix Multiplies Using Graphics Hardware". In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. SC '01. Denver, Colorado: Association for Computing Machinery, 2001, page 55. ISBN: 158113293X. DOI: 10.1145/582034.582089. URL: https://doi.org/10.1145/582034.582089.

[110] James Laudon and Daniel Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server". In: *SIGARCH Comput. Archit. News* 25.2 (May 1997), pages 241–251. ISSN: 0163-5964. DOI: 10.1145/384286.264206.

[111] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. "Warped-compression: Enabling Power Efficient GPUs Through Register Compression". In: *SIGARCH Computer Architecture News* 43.3 (June 2015), pages 502–514. ISSN: 0163-5964. DOI: 10.1145/2872887.2750417.

[112] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. "The Stanford Dash multiprocessor". In: *Computer* 25.3 (1992), pages 63–79. DOI: 10.1109/2.121510.

[113] Adam Levinthal and Thomas Porter. "Chap - a SIMD Graphics Processor". In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pages 77–82. ISSN: 0097-8930. DOI: 10.1145/964965.808581.

[114] Pei Li, Elisabeth Brunet, François Trahay, Christian Parrot, Gaël Thomas, and Raymond Namyst. "Automatic OpenCL Code Generation for Multi-device Heterogeneous Architectures". In: *44th International Conference on Parallel Processing (ICPP)*. Beijing, China: IEEE Computer Society, Sept. 2015, pages 959–968. DOI: 10.1109/ICPP.2015.105.

[115] Open Library. *Open Library Data Dumps*. https://openlibrary.org/data/ol_dump_works_latest.txt.gz. (Website). 2013.

[116] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (Mar. 2008), pages 39–55. ISSN: 0272-1732. DOI: 10.1109/MM.2008.31.

[117] Peter Lindstrom. "Fixed-Rate Compressed Floating-Point Arrays". In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (2014), pages 2674–2683. DOI: 10.1109/TVCG.2014.2346458.

[118] *Linux Kernel Module for Software-Based 842 Compression/Decompression*. (Website). 2015.

[119] Tom Lovett and Russell Clapp. "STiNG: A CC-NUMA Computer System for the Commercial Marketplace". In: *SIGARCH Comput. Archit. News* 24.2 (May 1996), pages 308–317. ISSN: 0163-5964. DOI: 10.1145/232974.233006.

[120] Meng-Yang Lu, Yu-An Lai, and Chih-Hung Kuo. "A Low-Latency Compression Architecture for Memory I/O Link on GPGPU". In: *International Journal of Eletrical Engineering* 26.5 (Oct. 2019), pages 203–210. ISSN: 1812-3031. DOI: 10.6329/CIEE.201910_26(5).0003.

[121] Matt Mahoney. *Large Text Compression Benchmark*. (Website). (Visited on 2011).

[122] Zoltan Majo and Thomas R. Gross. "A Library for Portable and Composable Data Locality Optimizations for NUMA Systems". In: *ACM Transactions on Parallel Computing* 3.4 (2017), 20:1–20:32. DOI: 10.1145/3040222.

[123] Pak Markthub, Akihiro Nomura, and Satoshi Matsuoka. "Using rCUDA to Reduce GPU Resource-Assignment Fragmentation Caused by Job Scheduler". In: *2014 15th International Conference on Parallel and Distributed Computing, Applications and Technologies*. 2014, pages 105–112. DOI: 10.1109/PDCAT.2014.26.

[124] Balthasar Martin, Robert Schmid, and Lukas Wenzel. "CAPI SNAP Development for Programmers". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Sept. 2017. URL: https://osm.hpi.de/capi-snap.

[125] Dylan Martin. *IBM: Power10 CPU's 'Memory Inception' Is Industry's 'Holy Grail'*. https://www.crn.com/news/components-peripherals/ibm-power10-cpu-s-memory-inception-is-industry-s-holy-grail-. (Website). Aug. 2020.

[126] Fabian Maschler, Jan-Henrich Mattfeld, and Norman Rzepka. "Scalable and Secure Infrastructures for Cloud Operations". Master's Project Report. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Sept. 2016.

[127] Jan-Henrich Mattfeld. "Design and Implementation of a Unified Middleware for Policy Enforcement in Multi-Cloud Infrastructures". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2018. URL: https://osm.hpi.de/bookshelf/Details/480.

[128] Julian McAuley. *Amazon Product Data*. http://jmcauley.ucsd.edu/data/amazon/index_2014.html. (Website). 2018.

[129] Microchip. *MCP39F511N Datasheet*. 2018. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/20005473B.pdf.

[130] MiTAC Computing Technology Corporation. *Tyan Transport HX TN83-B8251*. https://download.tyan.com/pub/datasheets/DataSheet_TN83-B8251.pdf. (Website). 2020.

[131] Sparsh Mittal and Jeffrey S. Vetter. "A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2016), pages 1524–1536. DOI: 10.1109/TPDS.2015.2435788.

[132] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. "InfiniteReality: A Real-Time Graphics System". In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '97. USA: ACM Press/Addison-Wesley Publishing Co., 1997, pages 293–302. ISBN: 0897918967. DOI: 10.1145/258734.258871.

[133] Timothy Prickett Morgan. "Shared Memory Pushes Wheat Genomics To Boost Crop Yields". In: *The Next Platform* (May 2016). http://www.nextplatform.com/2016/05/10/shared-memory-pushes-wheat-genomics-boost-crop-yields/.

[134] *MPI: A Message-Passing Interface Standard (Version 4.0)*. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. (accessed 2022-05-02). 2021.

[135] Ananya Muddukrishna, Peter A. Jonsson, and Mats Brorsson. "Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors". In: *Scientific Programming* 2015 (2015), 981759:1–981759:16. DOI: 10.1155/2015/981759.

[136] Michael Müller, Daniel Kessener, and Olaf Spinczyk. "First Things First: A Discussion of Modelling Approaches for Disruptive Memory Technologies". In: *Tagungsband des FG-BS Herbsttreffens 2021*. Bonn: Gesellschaft für Informatik e.V., 2021. DOI: 10.18420/fgbs2021h-02.

[137] Aaftab Munshi. "The OpenCL Specification". In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009, pages 1–314. DOI: 10.1109/HOTCHIPS.2009.7478342.

[138] NASA. *Curiosity's 1.8-Billion-Pixel Panorama*. https://mars.nasa.gov/resources/24801/curiositys-18-billion-pixel-panorama/. (Website). 2019.

[139] NVIDIA. *NVIDIA DGX A100 Datasheet*. https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf. (Website). 2020.

[140] NVIDIA. *NVIDIA DGX-1 Datasheet*. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-1-rhel-datasheet-nvidia-us-808336-r3-web.pdf. (Website). 2019.

[141] NVIDIA Corporation. *NVIDIA Management Library*. 2021. URL: https://docs.nvidia.com/pdf/NVML_API_Reference_Guide.pdf.

[142] *NVIDIA OpenCL Examples*. (Website). 2012.

[143] NVIIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf. (Whitepaper). 2020.

[144] *OpenMP Application Program Interface (Version 5.0)*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. (accessed 2022-05-02). 2018.

[145] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. "Parallel Lossless Data Compression on the GPU". In: *2012 Innovative Parallel Computing (InPar)*. May 2012, pages 1–9. DOI: 10.1109/InPar.2012.6339599.

[146] Oliver Pell and Oskar Mencer. "Surviving the End of Frequency Scaling with Reconfigurable Dataflow Computing". In: *SIGARCH Computer Architecture News* 39.4 (2011), pages 60–65. DOI: 10.1145/2082156.2082172.

[147] Gregory F. Pfister, William C. Brantley, David A. George, Steve L. Harvey, Wally J. Kleinfelder, Kevin P. McAuliffe, Evelin S. Melton, V. Alan Norton, and Jodi Weiss. "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture". In: *International Conference on Parallel Processing, ICPP'85, University Park, PA, USA, August 1985*. IEEE Computer Society Press, 1985, pages 764–771.

[148] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. "ThymesisFlow: A Software-Defined, HW/SW co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. Athens, Greece, Oct. 2020, pages 868–880. DOI: 10.1109/MICRO50266.2020.00075.

[149] Max Plauth, Matthias Bastian, and Andreas Polze. "Facilitating Policy Adherence in Federated OpenStack Clouds with Minimally Invasive Changes". In: *Proceedings of the Fifth HPI Cloud Symposium "Operating the Cloud"*. Nov. 2017. DOI: 10.13140/RG.2.2.34267.28969.

[150] Max Plauth, Joan Bruguera Micó, and Andreas Polze. "Improved Data Transfer Efficiency for Scale-Out Heterogeneous Workloads Using On-the-Fly I/O Link Compression". In: *Concurrency and Computation: Practice and Experience* (Dec. 2020), e6101. DOI: 10.1002/cpe.6101.

[151] Max Plauth, Felix Eberhardt, Frank Feinbube, and Andreas Polze. "A Survey of Security-Aware Approaches for Cloud-Based Storage and Processing Technologies". In: *Proceedings of the Third HPI Cloud Symposium "Operating the Cloud"*. Nov. 2015, page 33. DOI: 10.13140/RG.2.2.26664.57604.

[152] Max Plauth, Felix Eberhardt, Andreas Grapentin, and Andreas Polze. "Improving the Accessibility of NUMA-Aware C++ Application Development Based on the PGASUS Framework". In: *Concurrency and Computation: Practice and Experience* (Feb. 2022), e6887. DOI: 10.1002/cpe.6887.

[153] Max Plauth, Frank Feinbube, Frank Schlegel, and Andreas Polze. "A Performance Evaluation of Dynamic Parallelism for Fine-grained, Irregular Workloads". In: *International Journal of Networking and Computing* 6.2 (July 2016), pages 212–229. ISSN: 2185-2847. DOI: 10.15803/ijnc.6.2_212.

[154] Max Plauth, Frank Feinbube, Frank Schlegel, and Andreas Polze. "Using Dynamic Parallelism for Fine-Grained, Irregular Workloads: A Case Study of the N-Queens Problem". In: *Proceedings of the Third International Symposium on Computing and Networking (CANDAR)*. IEEE. Dec. 2015, pages 404–407. DOI: 10.1109/CANDAR.2015.26.

[155] Max Plauth, Lena Feinbube, and Andreas Polze. "A Performance Survey of Lightweight Virtualization Techniques". In: *Proceedings of the European Conference on Service-Oriented and Cloud Computing*. Springer. Sept. 2017, pages 34–48. DOI: 10.1007/978-3-319-67262-5_3.

[156] Max Plauth, Wieland Hagen, Frank Feinbube, Felix Eberhardt, Lena Feinbube, and Andreas Polze. "Parallel Implementation Strategies for Hierarchical Non-uniform Memory Access Systems by Example of the Scale-invariant Feature Transform Algorithm". In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. May 2016, pages 1351–1359. DOI: 10.1109/IPDPSW.2016.47.

[157] Max Plauth and Andreas Polze. "Are Low-Power SoCs Feasible for Heterogenous HPC Workloads?" In: *Proceedings of the European Conference on Parallel Processing*. Springer. Aug. 2016, pages 763–774. DOI: 10.1007/978-3-319-58943-5_61.

[158] Max Plauth and Andreas Polze. "GPU-Based Decompression for the 842 Algorithm". In: *Proceedings of the Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. Nov. 2019, pages 97–102. DOI: 10.1109/CANDARW.2019.00025.

[159] Max Plauth and Andreas Polze. "Towards Improving Data Transfer Efficiency for Accelerators Using Hardware Compression". In: *Proceedings of the Sixth International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. Nov. 2018, pages 125–131. DOI: 10.1109/CANDARW.2018.00031.

[160] Max Plauth, Florian Rösler, and Andreas Polze. "CloudCL: Distributed Heterogeneous Computing on Cloud Scale". In: *Proceedings of the Fifth International Symposium on Computing and Networking (CANDAR)*. IEEE. Nov. 2017, pages 344–350. DOI: 10.1109/CANDAR.2017.49.

[161] Max Plauth, Florian Rösler, and Andreas Polze. "CloudCL: Single-Paradigm Distributed Heterogeneous Computing for Cloud Infrastructures". In: *International Journal of Networking and Computing* 8.2 (July 2018), pages 282–301. ISSN: 2185-2847. DOI: 10.15803/ijnc.8.2_282.

[162] Max Plauth, Christoph Sterz, Felix Eberhardt, Frank Feinbube, and Andreas Polze. "Assessing NUMA Performance Based on Hardware Event Counters". In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. May 2017, pages 904–913. DOI: 10.1109/IPDPSW.2017.51.

[163] Max Plauth, Fredrik Teschke, Daniel Richter, and Andreas Polze. "Hardening Application Security using Intel SGX". In: *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. Aug. 2018, pages 375–380. DOI: 10.1109/QRS.2018.00050.

[164] *POWER NX zlib compliant library*. 2020. URL: https://github.com/libnxz/power-gzip.

[165] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. "Heterogeneous System Coherence for Integrated CPU-GPU Systems". In: *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, 2013, pages 457–467. DOI: 10.1145/2540708.2540747.

[166] *Project Gutenberg*. https://www.projekt-gutenberg.org/. (accessed 2022-05-02).

[167] Diego Puppin, Nicola Tonellotto, and Domenico Laforenza. "Using Web Services to Run Distributed Numerical Applications". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting*. Edited by Dieter Kranzlmüller, Péter Kacsuk, and Jack J. Dongarra. Volume 3241. Lecture Notes in Computer Science. Budapest, Hungary: Springer, Sept. 2004, pages 207–214. DOI: 10.1007/978-3-540-30218-6_32.

[168] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. "Fast Lossless Compression of Scientific Floating-Point Data". In: *2006 Data Compression Conference (DCC 2006), 28-30 March 2006, Snowbird, UT, USA*. IEEE Computer Society, 2006, pages 133–142. DOI: 10.1109/DCC.2006.35.

[169] James Reinders. *Intel Threading Building Blocks - Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007. ISBN: 978-0-596-51480-8. URL: http://www.oreilly.com/catalog/9780596514808/index.html.

[170] Harvey Richardson. "High Performance Fortran: History, Overview and Current Developments". In: *Thinking Machines Corporation* 14 (1996), page 13.

[171] S. Roberts, C. Mann, and C. Marroquin. "Redefining IBM power system design for CORAL". In: *IBM Journal of Research and Development* 64.3/4 (2020), 2:1–2:10. DOI: 10.1147/JRD.2019.2963637.

[172] David P. Rodgers. "Improvements in Multiprocessor System Design". In: *SIGARCH Comput. Archit. News* 13.3 (June 1985), pages 225–231. ISSN: 0163-5964. DOI: 10.1145/327070.327215.

[173] Daniel Roeder. "Recording and Profiling Workload Characteristics". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2017.

[174] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. "Query Processing on Heterogeneous CPU/GPU Systems". In: *ACM Computing Surveys* 55.1 (Jan. 2022). ISSN: 0360-0300. DOI: 10.1145/3485126.

[175] Florian Rösler. "Dynamic OpenCL - Distributed Computing on Cloud Scale". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Apr. 2017. URL: https://osm.hpi.de/bookshelf/Details/460.

[176] Karl Rupp. *CPU, GPU and MIC Hardware Characteristics over Time*. https://github.com/karlrupp/cpu-gpu-mic-comparison. (accessed 2022-05-02). 2019.

[177] Satish Kumar Sadasivam, Brian W. Thompto, Ron Kalla, and William J. Starke. "IBM Power9 Processor Architecture". In: *IEEE Micro* 37.2 (2017), pages 40–51. DOI: 10.1109/MM.2017.40.

[178] Fabien Sangalard. *A History of the NVIDIA Stream Multiprocessor*. https://fabiensanglard.net/cuda/index.html. (accessed 2022-05-02). 2020.

[179] Vijay Sathish, Michael J. Schulte, and Nam Sung Kim. "Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. Minneapolis, Minnesota, USA: ACM, 2012, pages 325–334. ISBN: 978-1-4503-1182-3. DOI: 10.1145/2370816.2370864.

[180] *Scalable and Secure Infrastructures for Cloud Operations*. https://cordis.europa.eu/project/id/644866. (accessed 2022-05-02). 2018.

[181] Robert Schmid. "Using FPGA Performance Counters for Profiling Heterogenous Applications". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Dec. 2018. URL: https://osm.hpi.de/bookshelf/Details/535.

[182] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. "Accessible Near-Storage Computing with FPGAs". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, Apr. 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387557.

[183] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. "Orchestrating Near-Data FPGA Accelerators Using Unix Pipes". In: *Proceedings of the Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. Nov. 2019, pages 125–128. DOI: 10.1109/CANDARW.2019.00030.

[184] Patrick Schmidt. "Optimization Guidelines for NUMA Architectures". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Jan. 2016. URL: https://osm.hpi.de/bookshelf/Details/533.

[185] Jacob T. Schwartz. "Ultracomputers". In: *ACM Trans. Program. Lang. Syst.* 2.4 (Oct. 1980), pages 484–521. ISSN: 0164-0925. DOI: 10.1145/357114.357116.

[186] Vincent Schwarzer. "Evaluierung von Unikernel-Betriebssystemen für Cloud-Computing". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, June 2016.

[187] Steven L. Scott. "Synchronization and Communication in the T3E Multiprocessor". In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VII. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1996, pages 26–36. ISBN: 0897917677. DOI: 10.1145/237090.237144.

[188] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. "Graph Processing on GPUs: A Survey". In: *ACM Computing Surveys* 50.6 (Jan. 2018). ISSN: 0360-0300. DOI: 10.1145/3128571.

[189] Evangelia A. Sitaridi, René Müller, Tim Kaldewey, Guy M. Lohman, and Kenneth A. Ross. "Massively-Parallel Lossless Data Decompression". In: *45th International Conference on Parallel Processing (ICPP)*. Philadelphia, PA, USA: IEEE Computer Society, Aug. 2016, pages 242–247. DOI: 10.1109/ICPP.2016.35.

[190] B. Stackhouse, S. Bhimji, C. Bostak, D. Bradley, B. Cherkauer, J. Desai, E. Francom, M. Gowan, P. Gronowski, D. Krueger, C. Morganti, and S. Troyer. "A 65 nm 2-Billion Transistor Quad-Core Itanium Processor". In: *IEEE Journal of Solid-State Circuits* 44.1 (2009), pages 18–31. DOI: 10.1109/JSSC.2008.2007150.

[191] William J. Starke, Brian W. Thompto, Jeffrey Stuecheli, and José E. Moreira. "IBM's POWER10 Processor". In: *IEEE Micro* 41.2 (2021), pages 7–14. DOI: 10.1109/MM.2021.3058632.

[192] Christoph Sterz. "Analyzing NUMA Performance Based on Hardware Event Counters". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2016. URL: https://osm.hpi.de/bookshelf/Details/530.

[193] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 2012, pages 1858–1865. DOI: 10.1109/IPDPSW.2012.229.

[194] Bjarne Stroustrup. "A Tour of C++: Abstraction Mechanisms". In: *The C++ Programming Language*. 4th edition. Addison-Wesley, May 2013.

[195] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison. "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI". In: *IBM Journal of Research and Development* 62.4/5 (2018), 8:1–8:8. DOI: 10.1147/JRD.2018.2856978.

[196] David Suggs, Mahesh Subramony, and Dan Bouvier. "The AMD "Zen 2" Processor". In: *IEEE Micro* 40.2 (2020), pages 45–52. DOI: 10.1109/MM.2020.2974217.

[197] Richard J. Swan, Andy Bechtolsheim, Kwok-Woon Lai, and John K. Ousterhout. "The Implementation of the Cm* Multi-Microprocessor". In: *Proceedings of the June 13-16, 1977, National Computer Conference*. AFIPS '77. Dallas, Texas: Association for Computing Machinery, 1977, pages 645–655. ISBN: 9781450379144. DOI: 10.1145/1499402.1499516.

[198] Karsten Tausche. "Memory Management on IBM Power Systems with NUMA Characteristics based on the PGASUS Programming Framework". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Oct. 2017. URL: https://osm.hpi.de/bookshelf/Details/540.

[199] Karsten Tausche, Max Plauth, and Andreas Polze. "dOpenCL–Evaluation of an API-Forwarding Implementation". In: *Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud"*. Nov. 2016. DOI: 10.13140/RG.2.2.16598.24641.

[200] SGM2 Development Team. *SGM2 Product Design Description*. Technical report. Honeywell Information Systems Italia, Sept. 1986.

[201] Spitzer Space Telescope. *GLIMPSE360: Spitzer's Infrared Milky Way*. http://www.spitzer.caltech.edu/glimpse360/. (Website).

[202] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. "POWER4 System Microarchitecture". In: *IBM Journal of Research and Development* 46.1 (2002), pages 5–25. DOI: 10.1147/rd.461.0005.

[203] Fredrick Teschke. "Hardening Applications with Intel SGX". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, July 2017.

[204] *The OpenCL Specification (Version 3.0)*. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf. (accessed 2022-05-02). Khronos OpenCL Working Group, Nov. 2021.

[205] C.J. Thompson, Sahngyun Hahn, and M. Oskin. "Using Modern Graphics Architectures for General-Purpose Computing: a Framework and Analysis". In: *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* 2002, pages 306–317. DOI: 10.1109/MICRO.2002.1176259.

[206] Transaction Processing Performance Council. *TPC Benchmark H*. (Website). 2018.

[207] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. "A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps". In: *SIGARCH Computer Architecture News* 43.3 (June 2015), pages 41–53. ISSN: 0163-5964. DOI: 10.1145/2872887.2750399.

[208] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. "On the Limits of GPU Acceleration". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism (HotPar)*. Berkeley, CA, USA: USENIX Association, 2010, page 13.

[209] Lukas Wenzel. "Operating System Facilities for FPGA Accelerator Designs". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, June 2019. URL: https://osm.hpi.de/bookshelf/Details/498.

[210] Lukas Wenzel, Robert Schmid, Balthasar Martin, Max Plauth, Felix Eberhardt, and Andreas Polze. "Getting started with CAPI SNAP: Hardware Development for Software Engineers". In: *Euro-Par 2018: Parallel Processing Workshops*. Springer. Aug. 2018, pages 187–198. DOI: 10.1007/978-3-030-10549-5_15.

[211] Christian Wuerz. "Resource Contention of Competing Processes in Parallel Systems". Master's thesis. Potsdam, Germany: Hasso Plattner Institute, University of Potsdam, Oct. 2017. URL: https://osm.hpi.de/bookshelf/Details/534.

[212] John J. Wuu, Rahul Agarwal, Michael Ciraula, Carl Dietz, Brett Johnson, Dave Johnson, Russell Schreiber, Raja Swaminathan, Will Walker, and Samuel Naffziger. "3D V-Cache: the Implementation of a Hybrid-Bonded 64MB Stacked Cache for a 7nm x86-64 CPU". In: *IEEE International Solid-State Circuits Conference (ISSCC) 2022*. San Francisco, CA, USA: IEEE, Feb. 2022, pages 428–429. DOI: 10.1109/ISSCC42614.2022.9731565.

[213] Shuji Yamamura, Yasunobu Akizuki, Hideyuki Sekiguchi, Takumi Maruyama, Tsutomu Sano, Hiroyuki Miyazaki, and Toshio Yoshida. "A64FX: 52-Core Processor Designed for the 442PetaFLOPS Supercomputer Fugaku". In: *IEEE International Solid-State Circuits Conference (ISSCC) 2022*. San Francisco, CA, USA: IEEE, Feb. 2022, pages 352–354. DOI: 10.1109/ISSCC42614.2022.9731627.

[214] Mohamed Zahran. "Heterogeneous Computing: Here to Stay: Hardware and Software Perspectives". In: *Queue* 14.6 (Dec. 2016), pages 31–42. ISSN: 1542-7730. DOI: 10.1145/3028687.3038873.

[215] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (May 1977), pages 337–343. ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714.

Glossary

`icswx` Initiate Coprocessor Store Word Indexed. 4, 40

AIX Advanced Interactive eXecutive. 38, 40

AME Active Memory Expansion. 40

API Application Programming Interface. 5, 8, 14, 21, 26–29, 32, 40, 53–57, 59, 62, 65, 80, 82, 84, 85, 108, 109

CAIA Coherent Accelerator Interface Architecture. 12

CAPI Coherent Accelerator Processor Interface. 12

CAPP Coherent Accelerator Processor Proxy. 12

CCIX Cache Coherent Interconnect for Accelerators. 23

ccNUMA Cache Coherent Non-Uniform Memory Access. 18–20

COTS Commercial Off-the-Shelf. 35

CPU Central Processing Unit. iii, 1–5, 12, 19–23, 25, 26, 30, 33–35, 38–41, 45–47, 49, 51, 52, 55, 60, 63–65, 69, 82, 83, 88, 96, 98, 107, 108, 111

CUDA Compute Unified Device Architecture. 22, 26, 28

CXL Compute Express Link. 23, 24

DDR Double Data Rate. 24

DRAM Dynamic Random Access Memory. 24

DSM Distributed Shared Memory. 17–19, 29

DSP Digital Signal Processor. 22, 26

FPGA Field-Programmable Gate Array. 12, 22, 23, 26, 38, 108

GDDR Graphics Double Data Rate. 24, 26

GPU Graphics Processing Unit. iii, iv, 1–6, 8, 12, 17, 20–23, 25, 26, 28, 32–35, 39, 44–47, 49, 51–56, 58, 60, 63–65, 67, 70–72, 74, 76, 79, 80, 82, 107–111

HBM High-Bandwidth Memory. 3, 24, 26

HPC High-Performance Computing. 32, 35, 83

IaaS Infrastructure as a Service. 54, 60

ICD Installable Client Driver. 26, 56, 57

IF Infinity Fabric. 82

ILP Instruction-Level Parallelism. 21

IMDB In-Memory Database. 4, 12, 25

ISA Instruction Set Architecture. 2, 14, 98, 105, 109

JNI Java Native Interface. 60

MIMD Multiple Instruction Multiple Data. 20

MPI Message Passing Interface. 4, 28, 30, 31, 54

NUMA Non-Uniform Memory Access. iii, iv, 1, 3–6, 8, 11, 17, 18, 20, 29–32, 81–91, 93–97, 99–105, 107, 109–111

OpenCAPI Open Coherent Accelerator Processor Interface. 23, 24

OpenCL Open Computing Language. 4, 5, 8, 26–28, 39, 46, 52–57, 59, 60, 62, 64–70, 74, 76, 79, 80, 108, 109

OpenMP Open Multi-Processing. 6, 29, 30, 82, 90, 91, 93–95, 97, 100–105, 109

PCIe Peripheral Component Interconnect Express. 3, 4, 23, 24

PGAS Partitioned Global Address Space. 29, 31, 83, 85, 104, 109

PowerAXON Power with A-bus, X-bus, OpenCAPI, and NVLink. 82

PSL POWER Service Layer. 12

RAII Resource Acquisition is Initialization. 83, 85, 86, 104, 109

RISC Reduced Instruction Set Computer. 21, 22

SCI Scalable Coherent Interface. 19, 20

SDRAM Synchronous Dynamic Random-Access Memory. 3

SIMD Single Instruction Multiple Data. 1, 2, 20–22

SMP Symmetric Multiprocessing. 17–20

SoC System on a Chip. 2, 34

SPMD Single Program Multiple Data. 30

SRAM Static Random-Access Memory. 24, 27

TBB Threading Building Blocks. 29, 30

TIFF Tag Image File Format. 79

TLP Thread-Level Parallelism. 22

UMA Uniform Memory Access. 1, 17, 30, 82

UPI Ultra Path Interconnect. 82

VAS Virtual Accelerator Switchboard. 4, 40, 49

VLIW Very Long Instruction Word. 21

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Dissertation “Improving the Accessibility of Heterogeneous System Resources for Application Developers using Programming Abstractions” (“Verbesserung der Zugänglichkeit heterogener Systemressourcen für Anwendungsentwickler durch Programmierabstraktionen”) selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, den 2. August 2022,

(Max Frederik Plauth)