PADERBORN UNIVERSITY

An OpenCL- and HLS-based Benchmark Suite for Reconfigurable Hardware in HPC: Performance Evaluation and Application

Dissertation

Marius Meyer

A thesis submitted to the *Faculty for Computer Science, Electrical Engineering and Mathematics* of *Paderborn University* in partial fulfillment of the requirements for the degree of Dr. rer. nat.

October 9, 2024

Für Ruth

Acknowledgements

This thesis would not have been possible without the support of colleagues, family, and friends. First of all, I would like thank my supervisor Prof. Dr. Christian Plessl for his guidance as well as for the engagement to promote collaborations and internships during my time as a student. They facilitated the progress of my work and helped me to see further than the end of my nose, especially during Covid-times, where other platforms for exchange where rather scarce. Also, I thank Prof. Dr. Marco Platzner for serving as a reviewer for this thesis as well as all other people serving on the oral examination committee.

Moreover, I would like to thank all colleagues from the High-Performance Computing research group and the Paderborn Center for Parallel Computing that helped me debugging nasty problems in the FPGA network infrastructure and provided me (most of the time) a working FPGA partition and software stack. A special thanks goes to Tobias Kenter for his valuable guidance, intensive discussions, collaboration on joint projects, and for taking care of my office plants when I wasn't able to. Additionally, I thank Arjun Ramaswami for the intense discussions and support during our time together at PC2, especially during Covid and seemingly endless months of home office. Our stand-up sessions helped me a lot to stay focussed also during times with limited work group interactions. Also thanks to Gerrit Pape, who contributed in various ways to my research as a student assistant by extending the HPCC FPGA benchmark suite and working on various inter-FPGA network stacks.

Last but not least, I would like to thank my family and especially my wife Lena for the support and encouragement to eventually complete this thesis.

Abstract

In recent years, FPGAs have gained attention in HPC systems due to their high performance and energy efficiency. Advances in HLS tools have made FPGA development more accessible by enabling programming in languages like OpenCL C or C/C++. However, HLS introduces abstraction layers that can have an effect on the synthesis results, increasing the impact of the programming tools on the final application performance. Not only because of that, comparing FPGAs only based on hardware resources often provides limited insight into the overall achievable HLS application performance. At this point, no existing tool supports comprehensive, empirical evaluation of FPGAs for HPC, especially with regards to emerging multi-FPGA systems and inter-FPGA communication, which takes an important role in scaling FPGA accelerated applications beyond single boards. A benchmark suite targeting FPGAs in the HPC domain is essential not only for performance comparison between boards but also for evaluating and advancing multi-FPGA systems as well as inter-FPGA communication strategies and gaining a better understanding how this novel infrastructure can be utilized best to further scale FPGA applications in context of HPC.

Within this work, we develop an OpenCL-based open-source benchmark suite for Intel and Xilinx FPGAs, which includes a build infrastructure that automates the compilation and synthesis of benchmarks based on configuration files to improve compatibility with the provided synthesis tools. Our suite is the first one featuring carefully optimized implementations of benchmark kernels that ensure comparable results across a broad spectrum of FPGA accelerator boards from both major vendors without manual code modifications. As another novelty, we also focus on multi-FPGA execution and inter-FPGA communication, going beyond the scope of a single FPGA board to whole multi-FPGA systems. All benchmarks are scalable within distributed memory systems using MPI, and we employ a flexible MPI/PCIe communication approach for inter-FPGA interactions. Additionally, we offer optimized benchmark implementations based on existing inter-FPGA communication frameworks that utilize the built-in networking ports of recent FPGA accelerator boards. Our benchmark suite is the first supporting the parallelization of benchmark applications over multiple FPGAs

up to complete multi-FPGA systems. With that, the suite is the first enabling the empirical evaluation of whole multi-FPGA systems and offering new opportunities in acquisition planning, application optimization, and the development of inter-FPGA communication approaches.

Next to the benchmark suite, we also develop highly scalable and performant FPGA-accelerated applications. Within the High Performance LINPACK (HPL) benchmark, we present one of the fastest LU decomposition implementations for multi-FPGA systems, achieving over 48 TFLOP/s single-precision floating-point performance on 64 FPGAs in the Cygnus system, fully utilizing one of the world's largest academic multi-FPGA systems. Furthermore, we conduct a detailed evaluation of inter-FPGA communication approaches using the ACCL communication framework with our b_eff benchmark, and we demonstrate our findings by porting a shallow water simulation on unstructured meshes to Xilinx FPGAs. Our shallow water implementation achieves the same per-FPGA performance while overcoming the scalability limitations of the original implementation, exhibiting linear scaling across all 48 Xilinx FPGAs of the Noctua 2 system at PC^2.

The evaluation of the benchmark suite demonstrates that the benchmarks effectively measure important device characteristics across a diverse range of FPGAs. Consequently, the empirical evaluation capabilities make our benchmark suite a valuable tool for acquisition planning and system maintenance. Furthermore, we show that the provided benchmarks are instrumental in evaluating current and future inter-FPGA communication frameworks, promoting broader adoption and improved scalability of multi-FPGA applications in the context of HPC.

Zusammenfassung

In den letzten Jahren haben FPGAs aufgrund ihrer hohen Leistung und Energieeffizienz vermehrt Einzug in HPC-Systemen erhalten und Fortschritte in den HLS-Werkzeugen haben die FPGA-Entwicklung zugänglicher gemacht, indem sie die Programmierung in Sprachen wie OpenCL C oder C/C++ ermöglichen. HLS führt jedoch auch Abstraktionsebenen ein, die sich auf die Syntheseergebnisse auswirken können, wodurch die Nutzung dieser Programmierwerkzeuge die Ressourcennutzung erhöhen und die Leistung der resultierenden Anwendungen reduzieren kann. Nicht nur deshalb bietet ein Vergleich von FPGAs nur auf der Grundlage von Hardwareressourcen oft nur begrenzte Einblicke in die insgesamt erreichbare HLS-Anwendungsleistung. Insbesondere im Hinblick auf aktuelle Multi-FPGA-Systeme und deren Infrastruktur für direkte Inter-FPGA-Kommunikation, die eine wichtige Rolle bei der Skalierung FPGA-beschleunigter Anwendungen spielt, existiert momentan kein Werkzeug, das eine umfassende, empirische Bewertung von FPGAs für HPC ermöglicht. Eine Benchmark-Suite für FPGAs im HPC-Bereich ist nicht nur für den Leistungsvergleich zwischen verschiedenen FPGA-Karten unerlässlich, sondern auch für die Bewertung und Weiterentwicklung von Multi-FPGA-Systemen und Inter-FPGA-Kommunikationsstrategien sowie für ein besseres Verständnis, wie diese neuartige Infrastruktur am besten für HPC Anwendungen genutzt werden kann. Im Rahmen dieser Arbeit entwickeln wir eine OpenCL-basierte quelloffene Benchmark-Suite für Intel- und Xilinx-FPGAs, die Werkzeuge enthält, welche die die Kompilierung und Synthese von Benchmarks basierend auf Konfigurationsdateien automatisieren, um die Kompatibilität mit diversen Synthesewerkzeugen zu verbessern. Unsere Suite ist die erste mit sorgfältig optimierten Implementierungen von Benchmark-Kerneln, die vergleichbare Ergebnisse über ein breites Spektrum von FPGA-Beschleunigerkarten beider großer hersteller ohne manuelle Codeänderungen gewährleisten. Als weitere Neuheit konzentrieren wir uns auch auf die Multi-FPGA-Ausführung und die Inter-FPGA-Kommunikation und gehen dabei über ein einzelnes FPGA-Board hinaus und skalieren die Ausführung unserer Benchmarks auf ganze Multi-FPGA-Systeme. Alle Benchmarks sind innerhalb verteilter Speichersysteme mit MPI skalierbar, und wir verwenden einen flexiblen MPI/PCIe-Kommunikationsansatz für Inter-

FPGA-Interaktionen. Darüber hinaus bieten wir optimierte Benchmark-Implementierungen auf Basis bestehender Inter-FPGA-Kommunikationsframeworks an, die die integrierten Netzwerkports aktueller FPGA-Beschleunigerkarten nutzen. Unsere Benchmark-Suite ist die erste, welche die Parallelisierung von Benchmark-Anwendungen über mehrere FPGAs bis hin zu vollständigen Multi-FPGA-Systemen unterstützt. Damit ist die Suite die erste, die die empirische Bewertung ganzer Multi-FPGA-Systeme ermöglicht und neue Möglichkeiten bei der Beschaffungsplanung, Anwendungsoptimierung und der Entwicklung von Inter-FPGA-Kommunikationsansätzen bietet.

Neben der Benchmark-Suite entwickeln wir auch hoch skalierbare und leistungsfähige FPGA-beschleunigte Anwendungen. Im Rahmen des High Performance LINPACK (HPL)-Benchmarks präsentieren wir eine der schnellsten LU-Zerlegungsimplementierungen für Multi-FPGA-Systeme, die eine Gleitkommaleistung mit einfacher Genauigkeit von über 48 TFLOP/s auf 64 FPGAs erreicht und damit alle FPGAs eines der weltweit größten akademischen Multi-FPGA-Systeme nutzt. Darüber hinaus führen wir eine detaillierte Bewertung von Inter-FPGA-Kommunikationsansätzen unter Verwendung des ACCL-Kommunikationsframeworks mit unserem b_eff-Benchmark durch und demonstrieren die Nützlichkeit unserer Ergebnisse, indem wir eine Flachwassersimulation auf unstrukturierten Meshes auf Xilinx-FPGAs portieren. Unsere Flachwasserimplementierung erreicht die gleiche Leistung pro FPGA und überwindet gleichzeitig die Skalierbarkeitsbeschränkungen der ursprünglichen Implementierung. Sie weist eine lineare Skalierung über alle 48 Xilinx-FPGAs des Noctua 2-Systems des PC^2 auf.

Unsere Evaluation der Benchmark-Suite zeigt, dass die Benchmarks wichtige Geräteeigenschaften über eine breite Palette von FPGAs hinweg effektiv messen. Die empirischen Evaluierungsfunktionen machen unsere Benchmark-Suite daher zu einem wertvollen Werkzeug für die Beschaffungsplanung und Systemwartung. Darüber hinaus zeigen wir, dass die bereitgestellten Benchmarks bei der Bewertung aktueller und zukünftiger Inter-FPGA-Kommunikationsansätze von entscheidender Bedeutung sind und eine vereinfachte Nutzbarkeit und verbesserte Skalierbarkeit von Multi-FPGA-Anwendungen im Kontext von HPC fördern.

List of Figures

List of Tables

List of Listings

Table of Contents

1

Introduction

Over recent years, Field Programmable Gate Arrays (FPGAs) have increasingly garnered attention for their integration into High Performance Computing (HPC) systems, primarily due to their notable performance and energy efficiency across various HPC applications. This trend has been propelled by several key advancements in FPGA technology and tooling. Notably, the refinement of High Level Synthesis (HLS) tools has simplified FPGA accelerator development for software engineers by enabling the utilization of familiar programming languages such as OpenCL C or C/C++. This addition to traditional hardware description languages like VHDL or Verilog has significantly streamlined the development process, reducing time-to-market for FPGA-accelerated applications. The high energy efficiency of FPGAs for various computation tasks further strenghtens the interest within the HPC domain where an increasing demand for compute resources leads to higher energy consumption and operation costs.

While HLS tooling accelerates development, it introduces additional abstraction layers that impact synthesis results. Optimization discrepancies between tool versions mean that identical source code may yield different hardware designs and performance. Mitigating this challenge involves explicitly annotating code with optimization hints to guide compilers. Although the major vendors use standardized programming models, code portability and seamless deployment across different FPGA platforms are still a challenge. In addition, the FPGA boards often come with a varying amount and type of logic resources as well as Digital Signal Processors (DSPs) that are used to implement floating-point operations. By only looking at the hardware resources, it is challenging to estimate the final performance of a specific application or to compare two different boards with each other, as it would be useful i.e. in

acquisition planning for new HPC systems. Until now, no tool exists to empirically evaluate FPGAs in the context of HPC and that supports both major vendors and their HLS toolchains. Either they only contain implementations specific to a single FPGA fabric or HLS toolchain, the contained applications are not representative of typical workloads in the HPC domain, or they are not scalable to properly utilize the available device resources.

Moreover, the integration of Quad Small Form-factor Pluggable (QSFP) ports into FPGA boards has emerged as a significant advancement. These versatile ports facilitate direct FPGA integration into networks, enabling efficient in-network data processing and inter-FPGA communication. Various communication methodologies, ranging from circuit-switched to packet-switched networks utilizing Ethernet and transport protocols like UDP or TCP, are now viable options, further enhancing flexibility and scalability in FPGA-accelerated HPC applications. These network stacks often rely on vendor Intellectual Property (IP) that has to be integrated into the user design using Hardware Description Languages (HDLs), again increasing the hurdle for application developers to make use of these features. Diverse communication frameworks have been proposed in recent years that not only simplify the accessibility of the networking features of the FPGAs but also provide higher-level communication abstractions and collective communication support to facilitate the parallelization of applications over multiple FPGAs.

A FPGA benchmark suite for HPC also needs to be able to empirically evaluate the full potential of multi-FPGA systems, their inter-FPGA connections, and HLS toolchains, to provide a solid base for performance evaluation and further developments. Such a tool is paramount in the further analysis of multi-FPGA systems and inter-FPGA communication approaches.

1.1 Contributions

This thesis makes contributions in two areas: Firstly, we propose an OpenCL-based benchmark suite specifically targeting FPGAs for the HPC domain compatible with a broad range of FPGA devices. We use the suite to empirically evaluate the performance of a range of single FPGAs boards as well as vendor software tools and go even beyond the scope of a single FPGA by leveraging and evaluating existing communication frameworks for multi-FPGA applications. Secondly, the collected results not only allow a better comparison of single boards up to multi-FPGA systems in terms of performance of power efficiency but can also be used to refine performance models of complex multi-FPGA simulations showing the impact of different inter-FPGA communication approaches on application performance. In detail, the main contributions of this work are the following:

1. We propose and implement an OpenCL benchmark suite for FPGA based on the well-known High Performance Computing Challenge (HPCC) benchmark suite to create a

cross-vendor performance measurement and evaluation tool for FPGAs. It is the first HLS benchmark suite supporting devices of both major vendors without manual code changes using the same source code and the first suite that is not only capable of scaling its benchmarks over single FPGAs but also over whole multi-FPGA systems. Therefore, we propose carefully optimized *base implementations* for each benchmark while also supporting the use of custom implementations. One of the key features of the *base implementations* is configurability such that they are capable of capturing the relevant device properties of recent FPGA boards. Therefore, we provide FPGA-adapted Open Compute Language (OpenCL) kernel implementations along with corresponding host code for setup and measurements for all HPCC benchmark applications. All benchmarks and the build system are open-source and publicly available on GitHub [2].

2. We use the benchmarks to measure the performance and power consumption on different FPGA families and boards with Intel and Xilinx FPGAs and various types of global memory including DDR and HBM2. Additionally, we evaluate varying floating-point precisions on Intel and Xilinx FPGAs and show that the benchmarks can capture significant differences in performance caused by the arithmetic units and the used DDR or HBM2 global memory. Further, we show that the suite can also be used to evaluate build tools and runtime environment behavior across tool versions and discuss the impact of the build tools on the application performance.

3. For the multi-FPGA support of the benchmark suite, we implement and analyze diverse inter-FPGA communication schemes ranging from naive approaches using PCIe and host Message Passing Interface (MPI) to FPGA-based communication frameworks making use of dedicated network infrastructure. We show the scalability of our benchmarks by executing them on production multi-FPGA systems and provide accurate performance models for each benchmark. Using our benchmarks and the provided performance models, we can show the potential impact of improved communication latency and throughput on the overall application performance.

4. Within the benchmark suite, we also implemented a state-of-the-art multi-FPGA LU decomposition within our High Performance LINPACK (HPL) implementation. The base implementation with compatibility to many Xilinx and Intel FPGAs achieves performance of more than 48 TFLOP/s for single-precision floating-point while also showing linear scalability on one of the largest multi-FPGA systems in the academic world with up to 64 FPGAs.

5. We use our benchmarks to execute an in-depth analysis of the ACCL communication framework to accelerate HPC workloads. Therefore, we identify different communication approaches that can be implemented within the framework and provide performance models for communication latency and throughput for each of the approaches.

6. We port an existing OpenCL C code of a shallow water simulation for Intel FPGAs to

Xilinx FPGAs and extend it with support for ACCL and communication via PCIe and MPI as used in HPCC FPGA. We use our benchmarks to show the impact of different configuration options in the framework and network stacks on communication latency and throughput. Our ported simulation shows the same performance as the original implementation using custom circuit-switched networks but overcomes its scalability limitations. It shows linear speedups in weak scaling scenarios with up to 48 FPGAs while completely utilizing the Noctua 2 FPGA partition – again one of the largest FPGA partitions available in academia. The limited scalability in strong scaling scenarios can be precisely explained by our improved performance model considering the communication latency and the results of our benchmark execution. With that, it is to our knowledge the first scientific multi-FPGA HPC application that makes use of a communication framework to scale over a complete FPGA partition.

1.2 Thesis Structure

The thesis is structured as follows: In Chapter 2, we will give an introduction to important concepts when using FPGAs on an HPC system. Infrastructure and toolchains are discussed and the current state of the FPGAs in HPC is described. In Chapter 3, we focus on the first four benchmarks of the suite that come with different types of memory access patterns. We describe the implementation of these benchmarks and their scalability on a single FPGA using pre-defined configuration options for the base implementations. Additionally, we evaluate the benchmarks on selected FPGAs with different kinds of global memory, floating-point precision, and tool versions.

The implementation of the remaining benchmarks of the suite is discussed in Chapter 4. In this chapter, we focus on base implementations that are scalable across multiple FPGAs and a whole system. Therefore, we propose and implement a widely usable communication approach using MPI and PCIe and execute the benchmarks on two different systems to evaluate the scaling behavior of the proposed base implementations. In addition, we provide an optimized version for each benchmark that makes use of custom circuit-switched networks for direct FPGA-to-FPGA communication based on Intel External Channel (IEC) and evaluate the impact of improved communication on the benchmarks.

In Chapter 5, we further extend the benchmarks to not only support OpenCL but also Vitis HLS and the Xilinx Runtime (XRT). Based on this extension, we evaluate the communication latency and throughput of the framework ACCL in detail and identify different communication approaches based on this framework. We then use our findings to port and optimize a shallow water simulation originally implemented for Intel FPGAs and which is showing high requirements for communication latency to Xilinx FPGAs and XRT.

2

FPGAs in HPC Systems

In recent years, FPGAs have gained significant attention as accelerator cards for HPC workloads and networking tasks. In the following, we will give an overview of the commonly used architecture of heterogeneous HPC systems equipped with FPGA boards as well as the software tools and frameworks used to develop applications for these systems and to make use of their communication capabilities.

2.1 Structure of FPGA PCIe Accelerator Boards

Within the context of HPC, FPGAs are typically integrated into systems like Graphics Processing Units (GPUs), as accelerator boards connected via PCIe to the compute node. GPUs are usually programmed based on the Single Instruction, Multiple Threads (SIMT) paradigm and achieve high compute performance through the massively parallel execution of threads over many compute cores. FPGAs on the other side support the implementation of deep, customized compute pipelines (also referred to as *dataflow* architectures) in reconfigurable hardware. Paralellism can be achieved by filling the pipeline with data such that every pipeline stage is working on different data items at a single point in time and the data items are *flowing* through the pipeline.

Most FPGA accelerator boards exhibit a similar, vendor-independent structure, as illustrated in Figure 2.1. The contained FPGA is managed and programmed via the PCIe connection to the host using runtime drivers such as OpenCL, SYCL, or custom solutions. Next to the PCIe connection, many boards also come with a USB or JTAG connection that is used for

administration purposes or to read out sensor data from the compute node. The FPGA IC includes the actual FPGA fabric, which contains logic resources, DSPs for arithmetic operations, and memory blocks. In addition to the FPGA IC itself, the board is equipped with other essential components necessary for the FPGA's basic functionality. These components include clocks and flash memory, which are required to store the FPGA program and operate the FPGA at the desired clock frequencies.

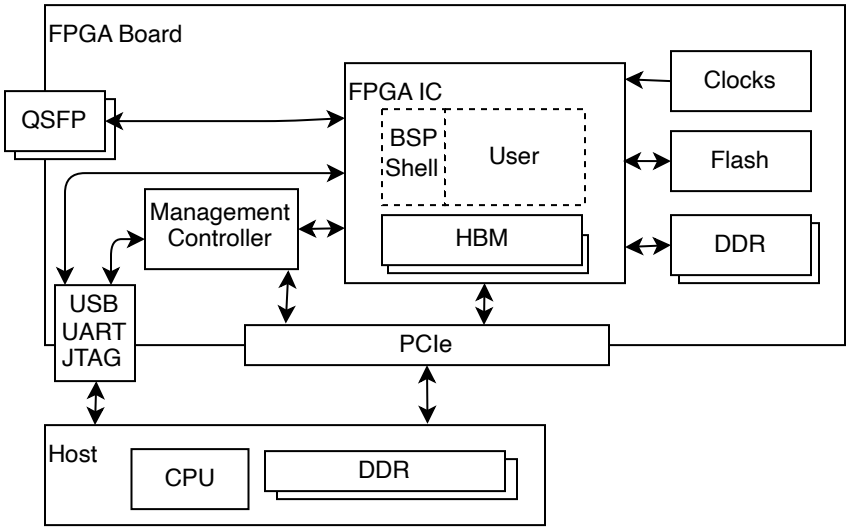


Figure 2.1: Block diagram of a FPGA PCIe board as it is used in HPC installations based on [3], [4].

The management controller provides sensor data such as voltages and temperatures to the host baseboard management controller. Additionally, it is used to manage the power-on and power-off sequences of the board.

All boards also come with a substantial amount of memory for application data. Often, this is DDR memory installed on the board. The FPGA IC may also be equipped with High Bandwidth Memory 2 (HBM2), which offers a significantly higher total bandwidth of several hundreds of GB/s compared to DDR, which typically provides only tens of GB/s. Both types of memory are usually organized in form of memory banks, where boards with DDR memory usually come with two to four banks and boards with HBM2 with 32 banks.

Recent FPGA boards support HLS toolchains, facilitating implementation from high-level languages such as C/C++ or OpenCL C. This high-level approach often provides additional logic to manage user logic execution, data transfers, and commonly used interfaces to global memory or PCIe as a *static partition*. Depending on the vendor, this *static partition* is referred to as a *Board Support Package (BSP)* or *shell*. This allows controlling the user logic from the CPU using the OpenCL or vendor-specific Application Program Interfaces (APIs) and drivers. The shell is flashed onto the FPGA, while user logic is separately flashed

using partial reconfiguration into a so-called *dynamic partition* or *user partition*. The HLS toolchains synthesize the user code into executable FPGA configurations (*bitstreams*) that are downloaded to the FPGA before usage.

In addition to these basic components, the boards frequently feature versatile QSFP ports that can be used for networking tasks. They can either be used to integrate FPGAs into networks for passive compute-in-network tasks or to make the FPGA board directly addressable via the network. Some boards – especially Intel-based FPGAs boards – include BSPs that incorporate a network link layer for point-to-point communication based on Serial Lite III, which can be directly utilized from HLS user kernels via provided APIs or OpenCL C extensions. On Xilinx FPGA boards, the connectors to these QSFP ports are routed to the user logic, allowing users to implement their network stacks independent of the used shell.

Therefore, direct FPGA-to-FPGA and host-to-FPGA communication can take place via high-speed network connections of up to 100 Gbit/s per QSFP28 port on current generation boards. Next-generation boards even offer up to 400 Gbit/s per port and come with (partially) hardened Ethernet IPs [5], [6]. These developments in the past and present further strengthen FPGAs and their networking capabilities in HPC systems.

2.2 HLS Toolchains and Their Opportunities for FPGAs

Before the widespread adoption of OpenCL C and HLS toolchains by major vendors like Intel and Xilinx, developing applications for FPGAs required extensive knowledge of HDLs and device-specific low-level details. This complexity hindered portability and slowed development, especially due to the manual integration of common interfaces in HPC such as PCIe. The introduction of HLS toolchains, along with their shells and BSPs, has significantly accelerated the development of FPGA-accelerated applications in HPC and allowed developers to focus on algorithm implementation.

2.2.1 OpenCL Terminology for FPGA Boards

OpenCL [7] is a standard for programming heterogeneous systems that include CPUs, GPUs, FPGAs, and other devices. It maintains a separation between the CPU (*host*) and the accelerator (*device*), with the device code compiled for and executed on accelerators like GPUs and FPGAs, but also itself are possible target devices. The host CPU code, implemented using the OpenCL API and a chosen programming language, manages data movements and the execution of the user logic in *command queues*. Tasks that are enqueued to the command queues are executed sequentially. The device code that is executed on the accelerators is referred to as *kernel* and is written in OpenCL C, a C-like programming language that is part of the OpenCL standard.

In OpenCL, two common programming models for kernel development are single work-item kernels (single-threaded) and ND-range kernels (multi-threaded), with the former preferred for FPGA programming due to its suitability for FPGA-specific optimizations [8]. The latter is usually preferred for the development of kernels for GPUs because the programming model better matches the hardware architecture of these devices.

For both major FPGA vendors, OpenCL terminology maps to hardware components as shown in Figure 2.2 for a single work-item kernel. The *global memory* and *constant memory* correspond to the DDR or HBM2 memory on the FPGA board, with data transferred from the *host memory* to the *global memory* via the OpenCL API before kernel execution. On the device side, global memory addresses are provided as kernel input parameters. *Local memory* buffers, declared as fixed-length arrays in the code, are implemented in on-chip memory blocks like Block RAM (BRAM), Ultra RAM (URAM), or registers, with resource mapping done automatically by HLS tools. A *compute unit* is the hardware implementation of kernel logic on the FPGA, and multiple compute units can be instantiated for a kernel to support concurrent kernel executions.

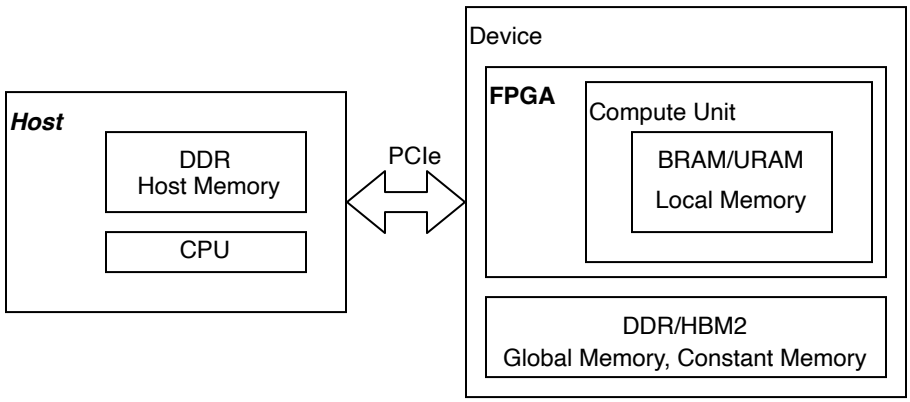


Figure 2.2: The terminology of FPGA board components as OpenCL device, highlighting the major memory regions commonly used in single work-item kernels.

2.2.2 Optimization of HLS Code for FPGAs

Several common programming patterns have emerged for implementing efficient hardware using HLS [9], [10]. They are generally applicable to HLS code and the toolchains will attempt to automatically apply many of the optimizations during synthesis. However, in many situations, the programmer has to guide the tools to synthesize performant hardware by applying modifications and compiler guidings in form of pragmas to the code. The three most important transformations *pipelining*, *unrolling*, and *memory optimizations* are briefly explained in the following.

Pipelining Pipelining is essential for increasing parallelism on an FPGA. Loops in HLS code are transformed into pipelines where iterations are processed sequentially. Depending on data dependencies, new iterations can already begin after a few clock cycles while the processing of the previous iteration is still ongoing in subsequent pipeline stages. The latency that is required between the initiation of two loop iterations is referred to as the Initiation Interval (II). The total latency C of a pipeline with N iterations and a latency L to process a single iteration is given by:

$$C = L + II \cdot (N - 1) \tag{2.1}$$

This architecture utilizes hardware efficiently and increases throughput by keeping pipeline stages occupied with different loop iterations. Strategies to optimize pipelining include loop fusion and flattening of nested loops to create deep pipelines with low IIs and high occupancies, thereby enhancing performance. As a result, the default goal of pipelining is to achieve a pipeline with an II=1, which means that in every clock cycle, the processing of a new loop iteration can be started and the overall utilization of the pipeline can be maximized.

Unrolling Operations can be vectorized through loop unrolling to further increase parallelism by initiating multiple iterations simultaneously within a wider pipeline. This reduces the overall execution latency, modeled as:

$$C = L + II \cdot \frac{(N - U)}{U} \tag{2.2}$$

where U is the unrolling factor. Unrolling can be applied to loops through pragmas, attributes, or automatically by the tools. In OpenCL C, the included vector data types can be used to generate very similar hardware compared to unrolling via pragmas or attributes.

Memory Optimization Efficient memory utilization is crucial for high performance on FPGAs. Global memory on recent FPGA boards can deliver 256-bit or 512-bit data per clock cycle and memory bank at frequencies of 250-300 MHz. To maximize this bandwidth, burst transfers must be used to aggregate data requests, requiring aligned and sequential global memory accesses. Proper pipelining and loop unrolling are used to increase the spatial locality of data accesses and ensure continuous data processing each clock cycle. Additionally, data has to be distributed across multiple global memory banks to maximize the available memory bandwidth, often necessitating compute pipeline replication to enhance the performance of memory-bound applications. This gets even more important on recent HBM2-equipped boards, offering high memory bandwidths of hundreds of GB/s distributed over up to 32 memory banks.

Local memory, leveraging on-chip memory blocks with higher bandwidth, can cache data to increase reuse. Techniques such as tiling create custom memory hierarchies from local memory, enhancing data reuse similar to CPU optimization strategies or enabling the efficient utilization of burst transfers from global memory. Next to plain data caches and memory hierarchies, specific programming patterns are detected by the tools and translated to specialized memory architectures such as shift registers or sliding windows.

2.2.3 Cross-Vendor HLS Code Compatibility

HLS toolchains can facilitate to write less device-dependent code, improving portability and simplifying library and framework development compared to HDLs. Many libraries have emerged, covering areas like linear algebra [11]–[14], emulation and usability [15], and communication frameworks [16]–[19]. However, although common optimization techniques for HLS code exist, the toolchains of the major FPGA vendors often behave differently and code annotations such as pragmas or attributes often differ across tools. However, these code annotations are cruicial to efficiently make use of specific device features and resources, ranging from the type and configuration of block memory, DSPs, and more.

With that, porting an OpenCL application from one vendor toolchain to the other remains challenging and requires careful optimization [20]. This leads to the point that, despite using C/C++ or OpenCL C, most libraries are typically compatible with only one of the two major FPGA vendors. When compatible with both, such as hlslib, separate implementations for different toolchains are usually provided.

2.3 Multi-FPGA Systems and inter-FPGA Networks

For the FPGAs of both vendors, no such standardized framework for communication similar to MPI on CPUs exists. However, several frameworks for inter-FPGA communication emerged in the previous years leveraging different kinds of infrastructure and concepts for communication. Typically, FPGA nodes are installed similarly to the schematic shown in Figure 2.3. A compute node consists of one or more CPUs that can communicate through dedicated interconnects. Communication beyond the node is realized by one or more Network Interface Controllers (NICs) connected via PCIe together with one or more FPGA boards. The QSFP ports of the FPGAs can be equipped with optical or electrical transceivers to form circuit-switched or packet-switched networks. Because of this flexibility, the depicted interconnect must not necessarily consist of a switch hierarchy. Instead, the QSFP ports can also be directly connected peer-to-peer. It is also possible to use the same interconnect for the inter-CPU and inter-FPGA communication to also allow direct communication of CPUs and FPGAs via the network.

Overall, next to single FPGA boards, we executed our benchmarks and applications on

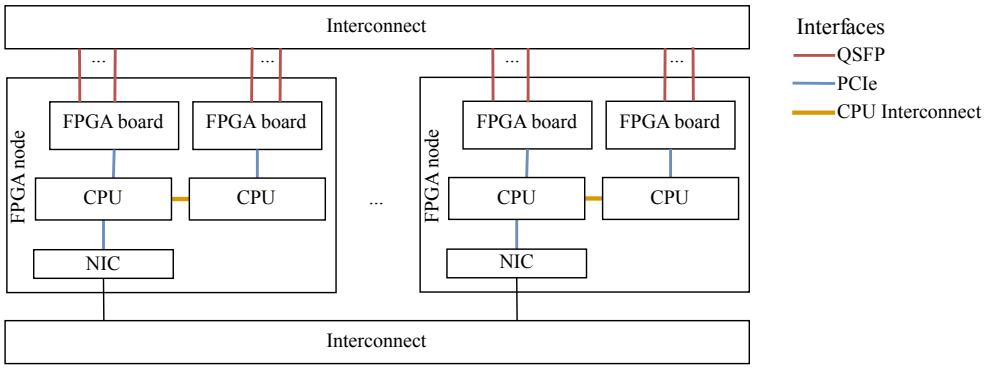


Figure 2.3: Schematic view of multi-FPGA system consisting of multiple nodes with multiple FPGA boards and CPUs per node. The CPUs communicate between nodes via one or more NICs connected via PCIe. FPGAs often form a dedicated network using their QSFP ports.

multi-FPGA systems with various configurations. These systems are described in more detail in the following.

2.3.1 Noctua 1

A visualization of the topology of the FPGA nodes in the Noctua 1 system is given in Figure 2.4. The FPGA partition of the Noctua 1 system was inaugurated at the Paderborn Center for Parallel Computing (PC²) in 2019 and consists of 32 *Nallatech/BittWare 520N* boards equipped with Intel Stratix 10 GX2800 FPGAs, where each of the 16 nodes is a two-socket system equipped with Intel Xeon Gold 6148F CPUs, 192 GiB of DDR4-2666 main memory, and two FPGAs connected via x16 PCIe 3.0. With the provided BSP version, only x8 PCIe 3.0 is available for data transfers between host and device. Moreover, the nodes in the cluster communicate over a hybrid network: The CPUs use an Intel Omni Path network with 100 Gb/s per port, whereas the FPGAs can exchange data via the four QSFP+ ports with up to 40 Gb/s per port. The serial interfaces are connected to a CALIENT S320 Optical Circuit Switch (OCS) that allows the configuration of arbitrary full-duplex point-to-point connections between the serial interfaces of the FPGAs. This functionality allows the creation of arbitrary network topologies only limited by the number of QSFP+ ports installed on the boards. The network topology is typically set up before running the application and stays unchanged during execution.

The BSP comes in two different sub-versions with and without support for the external channels. The *HPC* sub-version offers no support for communication via the QSFP ports and because of this missing functionality also a reduced resource utilization. With the *MAX* sub-version, data can direclty be passed between FPGAs via their QSFP ports using an implementation of the Serial Lite III protocol. To access the network interfaces from the OpenCL C, Intel provides an OpenCL extension covered in more detail in Section 2.4. The

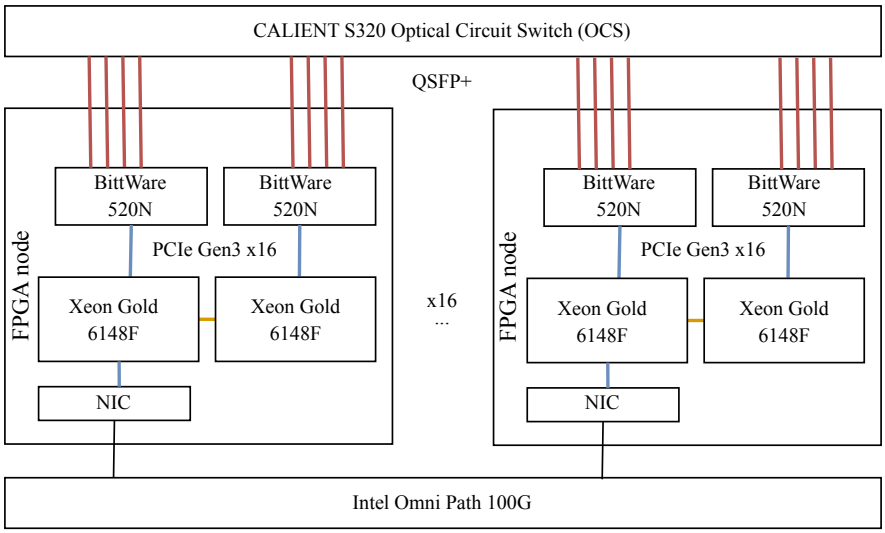


Figure 2.4: Topology of the Noctua 1 FPGA nodes. 16 nodes are connected to a single 100G Omni Path switch and every node is equipped with two CPUs and two FPGA boards. All four QSFP+ ports of each FPGA are connected to the OCS that allows the configuration of arbitrary point-to-point connections between the QSFP+ ports. Every QSFP+ port offers a maximum communication bandwidth of 40 Gbps. The OCS is typically configured once before the execution of the benchmarks.

workload manager of the system is configured in a way to allow the selection of the used BSP and SDK version during node allocation. This enables the user to define the tool version that fits best to the current workload which would otherwise be impossible for the users because the process usually requires a reboot of the node. A more detailed description of this mechanism is given by Bauer, Kenter, Lass, *et al.* [21], where the functionality was ported and further extended for the successor system Noctua 2.

2.3.2 Noctua 2

The FPGA partition of the Noctua 2 cluster [21] at the PC² contains one of the largest FPGA installations in the academic HPC domain. All BittWare 520N boards and the OCS of the Noctua 1 system were migrated to this system. Additionally, the FPGA partition was extended with a total of 48 Alveo U280 FPGAs distributed over 16 nodes totalling to 32 FPGA accelerated compute nodes. All FPGAs are connected to the CALIENT S320 OCS via their QSFP ports as given in Figure 2.5. In case of the BittWare 520N boards, these are four QSFP+ ports per board whereas for the Alveo U280 each board is equipped with two QSFP28 ports offering higher data rates of up to 100 Gb/s. The nodes are equipped with two AMD EPYC 7713 CPUs and a total of 512 GiB of main memory.

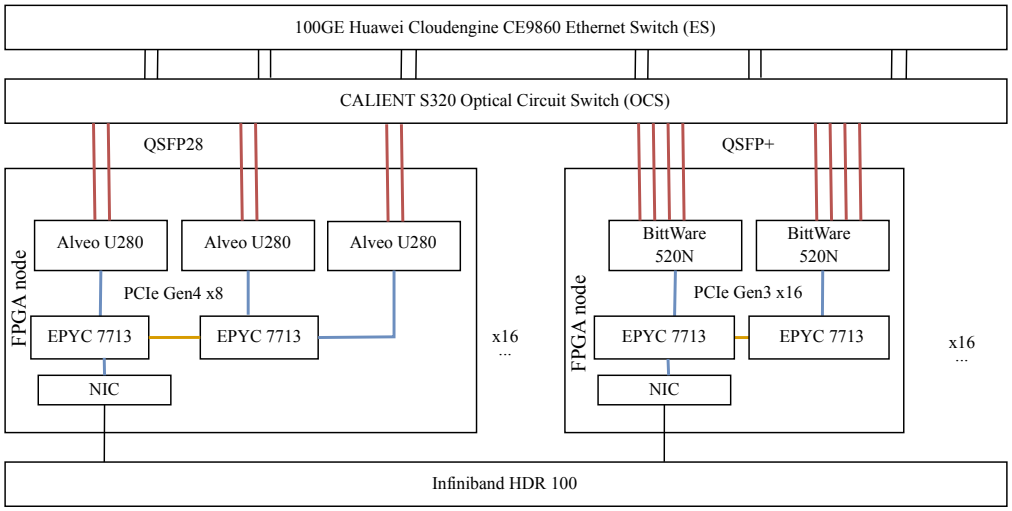


Figure 2.5: The Noctua 2 cluster consists of two node types: 16 nodes are equipped with two BittWare 520N board that are migrated from Noctua 1 together with the OCS. Additionally, 16 nodes each equipped with three Alveo U280 are added to the system and the QSFP28 ports of the FPGAs are also connected to the OCS. Another major extension of the inter-FPGA network is the ethernet switch that is also directly connected to the OCS. With that, the infrastructure supports circuit-switched as well as packet-switched inter-FPGA communication. The compute nodes communicate via a separate Infiniband network.

The OCS is protocol agnostic and can be used to physically connect arbitrary ports of the switch to form direct point-to-point connections with minimal latency overhead. In addition, a 128-port 100GE Huawei Cloudengine CE9860 Ethernet Switch (ES) is also connected to the OCS. By configuring the OCS, this setup also allows the connection of FPGA ports to the ethernet switch to form packet-switched networks. We use this setup in our evaluation to look more deeply into the communication latencies introduced by packet-switched communication. In this work, Vitis 2022.2, XRT 2.14, and the shell `xilinx_u280_gen3x16_xdma_1_202211_1` are used for synthesis and execution of all applications, but similar to Noctua 1, the system also comes with the ability to allow the users to change the runtime and shell version used with the FPGAs during node allocation.

2.3.3 ETH HACC System

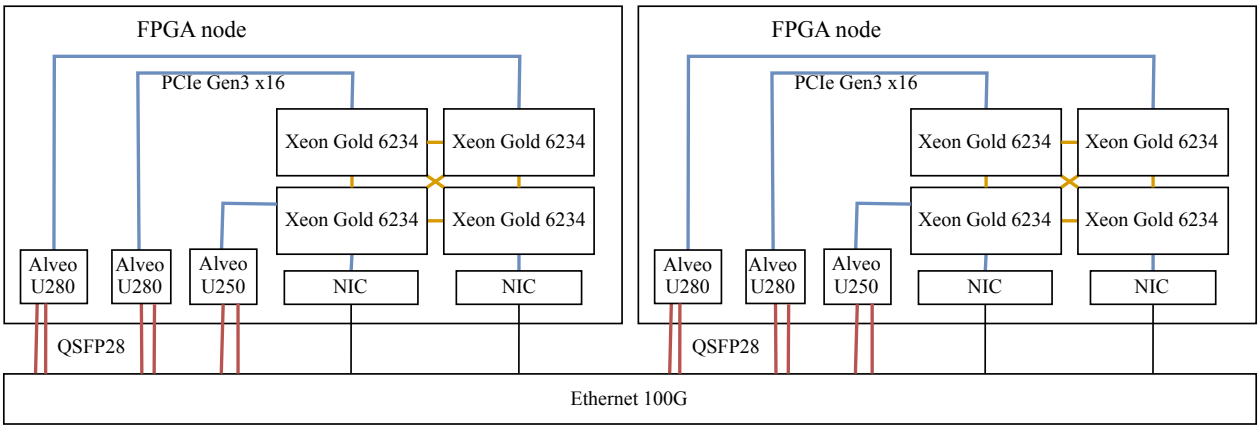


Figure 2.6: Topology of the used ETH Heterogeneous Accelerated Compute Clusters (HACC) system nodes. Each node is connected to the 100G Ethernet switch via two NICs and is equipped with four CPUs and two FPGA boards. The QSFP28 ports of the FPGA boards are connected to the same ethernet switch than the NICs.

During our experiments, the HACC system at ETH Zurich contains four heterogeneous nodes equipped with multiple types of FPGA boards [22]. The topology of the used nodes is given in Figure 2.6. The visualization is simplified to only show the accelerator boards and communication interfaces that are used for the benchmark execution. Each node also contains an additional Alveo U250 board which is not used in the experiments. Next to that, each node comes with three Alveo U280 boards and four Xeon 3234 CPUs. In contrast to the other systems presented here, all FPGAs of the system are directly connected to an ethernet switch via their QSFP ports. The NICs of the compute nodes are connected to the same ethernet switch, allowing direct communication between CPU applications and FPGAs via the network. This also emphasizes, that the networks depicted in Figure 2.3 not necessarily have to be two separate networks and may not only be limited to pure inter-FPGA communication.

2.3.4 Cygnus

The Albireo partition of the Cygnus system at the Center for Computational Sciences at the University of Tsukuba contains 32 heterogeneous compute nodes with NVidia Tesla V100 GPUs and *Nallatech/BittWare 520N* boards with Intel Stratix 10 GX2800 H-tile [23]. In contrast to the FPGAs installed in the Noctua 1 and later migrated into the Noctua 2 system, these FPGAs are equipped with QSFP28 modules supporting up to 100 Gb/s of throughput per port. Each node consists of two Intel Xeon Gold 6126, four NVidida Tesla V100, and two BittWare 520N. The topology of the nodes is given in Figure 2.7. The main memory consists of 192 GiB and a Mellanox InfiniBand HDR100 network is used for communication.

Next to the Infiniband network, the FPGA boards are also connected in a circuit-switched network forming a 8×8 2D torus using the QSFP28 ports of the boards. With that, the system is different to the other presented systems in the fixed topology of the circuit-switched inter-FPGA network. The network topology has to be taken to account when allocating compute nodes and programming FPGAs with bistreams to correclty route data through the network.

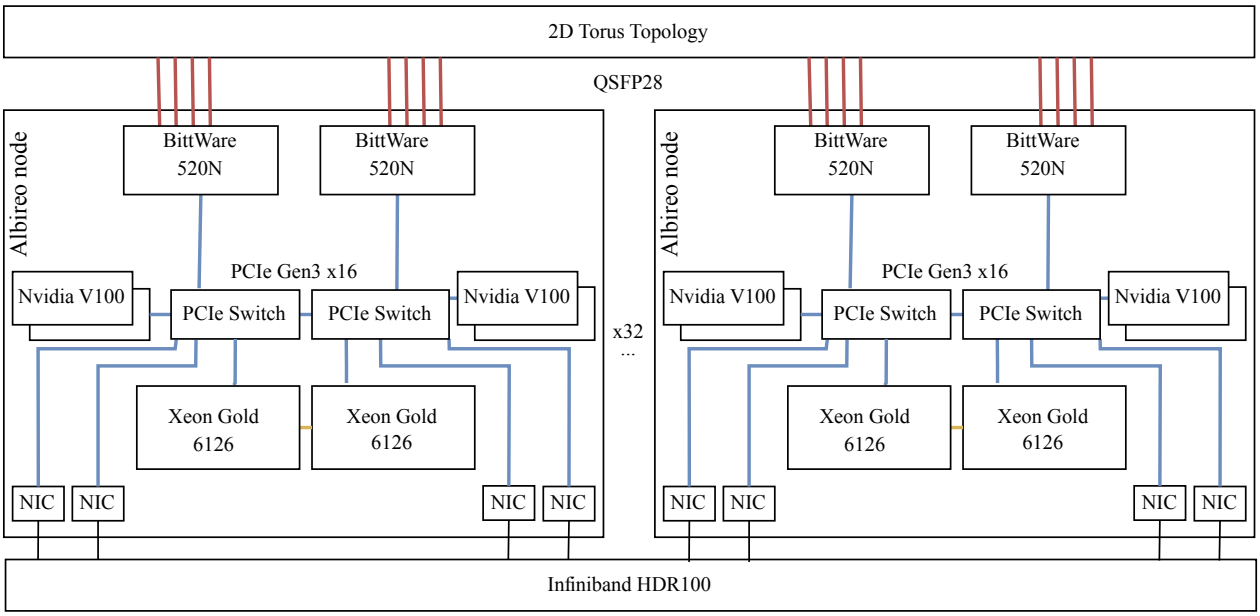


Figure 2.7: Topology of the Cygnus Albireo nodes. Each node is connected to the Infiniband network via four NICs and is equipped with two CPUs and two FPGA boards. The QSFP28 ports of the FPGA boards are used to form a fixed 8×8 2D torus.

2.4 FPGA Network Stacks and Communication Frameworks

Similar to GPUs, FPGAs are emerging from a single-device accelerator to multi-device accelerators with direct communication channels. In the GPU context, infrastructure and tools are well advanced, where popular implementations are NVLink [24] for NVIDIA as well as the Infinity Fabric (IF) for AMD GPUs. NVLink supports the usage of dedicated switches to form large networks of interconnected GPUs with a bandwidth of hundreds of GB/s whereas AMD IF comes with transport protocols to directly communicate between components using PCIe or other interfaces. This also supports direct communication of a GPU to NICs as it is done within the supercomputer Frontier [25]. The NVSwitches also come with dedicated hardware support for reductions. Implementation of the communication can be done using Compute Unified Device Architecture (CUDA) or ROCm with support for advanced MPI-like collectives provided by dedicated libraries for both vendors (Nvidia Collective Communications Library (NCCL) and ROCm Communication Collectives Library (RCCL)).

2.4.1 Network Stacks

Most existing communication frameworks for FPGA utilize dedicated networks for communication via the QSFP ports. Therefore, on the lower network layers, they utilize existing IPs or open-source network stacks. Commonly used network stacks that support easy integration into HLS projects and that are compatible with the vendor-provided FPGA shells are given in Table 2.1. Several frameworks are based on one of these network stacks and implement collective communication and other features on top. Usually, the network stacks are only compatible with either Intel or Xilinx FPGAs and their Software Development Kits (SDKs).

A dataflow architecture is often used for accelerators implemented on FPGAs where data is streamed through deep pipelines to increase hardware utilization and throughput. This makes streaming communication between FPGAs good candidates because it allows the seamless distribution of compute pipelines across multiple FPGAs. Intel provides with IEC [26] a tightly integrated network stack based on SerialLite III which is for some boards directly integrated into the BSP. Data can be sent and received from remote FPGAs by utilizing the channels/pipes API that is frequently used in HLS to represent AXI or Avalon streaming interfaces between kernels. Intel provides an OpenCL C extension to direclty use the network interfaces from openCL C code. This leads to a very similar way to implement inter-FPGA communication as it would be done for inter-kernel communication. The major limitation of this approach is that every channel is directly mapped to a physical QSFP port, thus limiting the maximum number of communication pairs across FPGAs.

For Xilinx FPGAs, mainly Ethernet-based network stacks exist with open-source implementations of UDP or TCP-IP [19], [27]. The network stacks have to be manually integrated into

the user project before they can be used. Afterwards, they can be connected to the user logic via AXI streams and using the link configuration of the HLS toolchain. Another network stack that finds use in communication frameworks is AlveoLink [28]. It uses RoCEv2 for data exchange and can be interfaced using AXI streams, similar to the UDP and TCP-IP implementation. However, at the time of writing, AlveoLink does only support Alveo U55c FPGA boards. The Aurora HLS project [29] aims to simplify the usage of the vendor-provided Aurora IP by providing a ready-to-use kernel design that can be directly intergrated into HLS projects. This enables the utilization of the QSFP ports via AXI stream interfaces from the user design, similar to IEC for Intel FPGAs. However, the network stack is not part of the shell and instead needs to be added to the project as a compute unit. This also gives the opportunity to establish circuit-switched networks among Xilinx FPGAs by only using the HLS toolchains. Currently, the network stack is only tested on Xilinx Alveo U280 FPGAs.

Table 2.1: Summary of commonly used network stacks with support for direct integration into HLS code.

Work	**Network Stack**	**Vendor Support**	**SDK**
IEC [26]	SerialLite III	Intel	OpenCL, oneAPI
VNx [27]	Ethernet, UDP	Xilinx	XRT
TCP [19]	Ethernet, TCP/IP	Xilinx	XRT
AlveoLink [28]	RoCEv2, Ethernet	Xilinx	XRT
Aurora HLS [29]	Aurora 64B/66B	Xilinx	XRT

2.4.2 Frameworks

On top of the discussed network stacks and also by implementing custom solutions, the frameworks given in Table 2.2 implement advanced communication capabilities including collective communication and collective computation. Similar to the network stacks, the framework implementations are usually restricted to specific FPGA devices and part of custom HLS toolchains or shells.

MVAPICH2-FPGA [18] makes use of the PCIe interface and the host NIC to exchange data between FPGAs. The host MPI installation is used to exchange the data via the host network which results in a wide applicability of the approach on all FPGAs with an OpenCL runtime. The data is first copied from the FPGA memory to the host memory via the OpenCL API and then transferred via MPI. This requires only a few resources on the FPGA but usually comes with a higher communication latency compared to frameworks that make use of the QSFP ports because of the additional copy operations between FPGA and host memory. The MVAPICH2 extension supports directly passing data buffers located on FPGA devices to the MPI calls. Copy operations and MPI calls are then tiled and interleaved to improve the communication throughput. Some runtimes support the direct access from the FPGA board to the host memory to reduce the number of copy operations using Shared Virtual Memory

(SVM). These approaches are good candidates to further improve the performance of this communication framework. However, even with these features, communication latency can still be a bottleneck compared to other communication approaches.

Table 2.2: Summary of MPI-like inter-FPGA Communication Frameworks.

Work	**Interface**	**Network Stack**	**SDK / Runtime**	**Vendor Support**
MVAPICH2-FPGA [18]	PCIe	Host MPI	OpenCL	All OpenCL devices
Galapagos [30], [31]	QSFP	Ethernet, TCP-IP	custom	Xilinx
EasyNet [19]	QSFP	VNx, TCP	OpenCL	Xilinx
ACCL [32]	QSFP	VNx, TCP	XRT Coyote	Xilinx
VCSN [33]	QSFP	Ethernet	custom	Intel
SMI [16]	QSFP	IEC	OpenCL	Intel
CIRCUS [17]	QSFP	custom SerialLite III	OpenCL	Intel
TAPA-CS [34]	QSFP	AlveoLink	XRT	Xilinx

All other frameworks in this overview make use of the QSFP ports for data exchange. Some of them make use of an IP core provided by Xilinx or Intel implementing Ethernet Media Access Control (MAC) and in some cases also make additional use of transport protocols like UDP or TCP. The hardware abstraction framework Galapagos for Xilinx FPGAs implements a subset of MPI in its communication layer [30], [31]. It mainly targets cloud applications and provides a complete framework including shell and host API to implement distributed FPGA applications including collective communication on top of UDP or TCP.

EasyNet [19] is another attempt to make networking functionalities available for HLS user applications. It offers point-to-point and collective communication implementations based on a TCP network stack that can be included in a user HLS design via C++ header files. In this case, the communication logic will only be instantiated in a design if the collective is used by the user kernel but the flexibility of this approach is limited because collectives can not be arbitrarily combined in the same design since they are occupying the input and output streams of the network stack. Alveo Collective Communication Library (ACCL) [32] implements dedicated hardware to execute arbitrary collectives without these limitations. The core of this hardware is the Collective Communication Library Offload (CCLO) component which is executing communication commands passed to it during runtime via an AXI stream. The firmware executed by the CCLO contains implementations for point-to-point communication and collectives that consist of a sequence of data move operations. This architecture facilitates the execution of arbitrary communication without modifications in the hardware while still supporting further optimizations through plugins and firmware updates. It comes with support for UDP, TCP, and RDMA [35] as network layer and the whole framework is designed to be highly adaptable to the intended use case. ACCL does not only allow to directly forward the

communication data to the user logic using AXI streams but also to buffer and copy received data in global memory. This flexible architecture enables versatile communication approaches targeted to the specific use case in trade-off with a potentially higher resource utilization compared to the on-demand implementation used within EasyNet. Both frameworks can be used together with the official Xilinx shells and runtimes while ACCL also comes with support for the Coyote shell. Overall, on the one hand, the implementation of complex network stacks like TCP/IP and the implementation of collective communication and computation on the FPGA requires a considerable amount of logic resources, which reduces the amount of available resources for the user application. On the other hand, the application may profit from high bandwidth and low latency communication as well as increased scalability.

TAPA-CS [34] is an extension of the TAPA framework [36]. The TAPA framework comes with a programming model that aims to simplify the implementation of dataflow architectures for Xilinx FPGAs. Another part of the used toolflow is AutoBridge [37], a tool to improve the floor planning on the FPGA to increase the clock frequencies of the synthesized bitstream. With TAPA-CS, TAPA is extended by the functionality of distributing a large design over multiple FPGAs. Therefore, the design is automatically partitioned and communication between the partitions is implemented based on RoCEv2 using AlveoLink [28]. This separates the TAPA-CS approach from the other approaches discussed in the work, since the partitioning is happening automatically and the communication between FPGAs remains implicit. With all other discussed frameworks, communication has always to be explicitly implemented by the programmer using the Single Program Multiple Data (SPMD) or other programming models.

In contrast to the packet-switched communication frameworks discussed above, also frameworks exist that make use of direct peer-to-peer communication and circuit-switched networks. SMI [16] and CIRCUS [17] are making use of the SerialLite III IP that is included in some version of the BittWare BSPs and can be directly used from OpenCL C code through vendor extensions. A circuit-switched network is formed among FPGAs by directly connecting the QSFP ports while the frameworks take care of data routing. The frameworks usually show lower resource utilization compared to the packet-switched counterparts because of the lower complexity of the network stacks, but scalability and flexibility are reduced because the network topology is often fixed and limited by the number of QSFP ports on a board. This limitation also increases the communication latency in large networks since data has to be routed over multiple FPGAs before reaching its destination. On the other side, these frameworks are well suited for the streaming nature of FPGA data processing. Thus, a slightly different approach is taken by the Virtual Circuit Switching Network (VCSN) [33] which implements peer-to-peer networks on top of Ethernet. This approach aims to combine the flexibility of a packet-switched network with the low complexity and high suitability of circuit-switched networks for FPGA-typical dataflow architectures. Using an Ethernet switch, the physical channels can be divided into multiple virtual channels to further increase the number of direct

neighbors of a node in the network. As a downside, this abstraction can lead to an increase of the communication latency [38] and individual virtual channels will show reduced bandwidth because they share the same physical channel.

This huge variety of inter-FPGA communication frameworks facilitates the use of inter-FPGA communication with various kinds of network infrastructures and multi-FPGA systems. However, there is still limited insight into how and which of the different communication concepts and network infrastructures is suited best for which kind of application. A direct comparison of the existing frameworks is often non-trivial because porting of applications between frameworks and toolchains requires major efforts and is rarely done. These problems can be reduced and the comparison of communication frameworks can be simplified with the use of standardized benchmarks that precisely define the application parameters, algorithms, and the validation process.

3

Benchmarking Single FPGAs

In HPC, benchmarks are an important tool for performance comparison across systems. They are designed to stress important system properties or generate workloads that are similar to relevant applications for the user. Especially in acquisition planning, they can be used to define the desired performance of the acquired system before it is built. Since it is a challenging task to select a set of benchmarks to cover all relevant device properties, benchmark suites can help by providing a pre-defined mix of applications and inputs, for example SPEC CPU [39] and HPCC [40].

Therefore, we propose HPCC FPGA, an FPGA OpenCL benchmark suite for HPC using the applications of the HPCC benchmark suite. The motivation for choosing HPCC is that it is well-established for CPUs and covers a small set of applications that evaluate important memory access and computing patterns that are frequently used in HPC applications. In this chapter, we will describe the implementation and four benchmarks of the suite that target different types of memory access patterns. We evaluate selected FPGA devices of the major FPGA vendors in the HPC domain using the proposed benchmarks to show how *HPCC FPGA* quantifies the impact of HBM2 high-bandwidth memory FPGAs in comparison to FPGA boards with DDR memory. With the support for half and double precision variants, the benchmark can also capture important differences in the arithmetic units of FPGA architectures.

The contents of this chapter have been presented and published in the proceedings of the *International Workshop on Heterogeneous High-performance Reconfigurable Computing* and the *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*

[41], [42]. Moreover, an in-depth evaluation was published as an article in the *Journal of Parallel and Distributed Computing* [43]. All measurements results, scripts, and reports used in this chapter are published via Zenodo [44].

3.1 HPCC Challenge for FPGAs

The HPCC benchmark suite [40] consists of seven benchmarks. Three of them are synthetic benchmarks that measure the memory performance for successive accesses (STREAM [45]) and random updates (RandomAccess) as well as the effective network bandwidth of a system (b_eff). Moreover, it contains four applications of varying complexity: Matrix transposition (PTRANS), 1D Fast Fourier Transformation (FFT), matrix multiplication (GEMM), and High Performance LINPACK (HPL).

A central concept of the HPCC benchmark suite is to characterize the performance of HPC applications by their memory access patterns as it is shown in Figure 3.1. The applications were chosen to represent border cases of spatial and temporal locality in memory accesses such that the performance for all other applications can be estimated based on these border cases. The achieved performance in CPU systems is thus highly dependent on the memory and cache architecture. In this chapter we will focus on four of the benchmarks to represent and discuss all four border cases: STREAM, RandomAccess, FFT and GEMM. The three remaining benchmarks are discussed in more detail in Chapter 4, when we move the focus from memory access patterns to inter-FPGA communication.

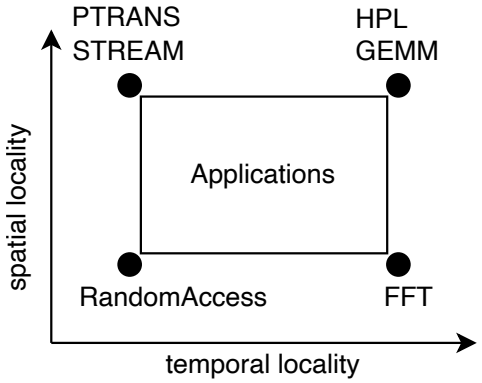


Figure 3.1: Visualization of the spatial and temporal locality of memory accesses for the HPCC benchmarks based on the illustration in [40].

When benchmarking FPGA boards with this concept, it is crucial to note that even for a given board, the memory hierarchy is not fixed. While the off-chip memory itself and typically parts of the memory controllers are fixed hardware components, local buffers or caches, address generators, pre-fetchers, and data buses inside the FPGA are provided by

the shell or generated by the SDK depending on the benchmark implementation. Thus, in contrast to the CPU version of HPCC, HPCC FPGA does not only measure hardware properties of a given memory interface but rather also the ability of the tools to optimize the memory hierarchy and compute pipelines of a kernel for the specific pattern to provide good performance of the calculation on an FPGA.

Another aspect of the HPCC benchmark suite is the distinction between two different runs:

- *Base runs:* done with the provided reference implementations.
- *Optimized runs:* done with implementations that use architecture-specific optimizations.

The goal of HPCC FPGA, as presented in this article, is to provide implementations for the base runs that are reasonably optimized for Intel and Xilinx FPGAs and their SDKs. With current FPGA execution models, such optimizations necessarily include adaptations to the available resources, in particular the number of physical memory interfaces and configurable local memory. Therefore, the base implementation exposes defined configuration parameters for such customization, without pre-empting the full flexibility that is reserved for optimized runs with manual code changes and target architecture- or SDK-specific designs. The presented adjustable parameters for the base runs are a subject of discussion and might be changed with the evolution of the FPGA architectures and toolchains.

In Table 3.1 the memory access patterns for the benchmarks contained in the first version of *HPCC FPGA* are given for CPU and the FPGA base implementations proposed in this paper. For FPGA, the memory is further divided into global and local memory representing the on-chip and off-chip memory on the FPGA, respectively. In HPCC, the type of memory accesses is categorized into spatial and temporal locality. Spatial locality is represented in the table by the *linear*, *strided*, and *random* access pattern, where *linear* corresponds to consecutive accesses and a high spacial locality whereas *strided* accesses can show medium to low and *random* very low spacial locality. Temporal locality is increased for repeated accesses to the same data, where also the time between the accesses is important for data caching.

Table 3.1: Memory Access Patterns for the first four benchmarks of HPCC.

Benchmark	**CPU**	**FPGA**	
		Global Memory	**Local Memory**
STEAM	linear	linear	linear
RandomAccess	random	random	linear
FFT	strided, repeated	linear	strided, repeated
GEMM	strided/linear, repeated	linear, repeated	strided/linear, repeated

Similar to CPU architectures, the local memory of the FPGA can be used to form caches and even cache hierarchies specifically optimized for the application. Therefore, FPGA applications follow a similar memory optimization strategy compared to CPUs by moving

the strided memory accesses with low spatial locality into the local memory and increase the temporal locality if possible using blocked algorithms. The global memory is accessed in a linear fashion similar to the access of complete cache lines in CPU architectures.

3.1.1 Common Build Setup

The HPCC FPGA benchmark suite is set up to create one host binary and FPGA bitstream per benchmark, with source code structure, build process, and benchmark execution very similar for all benchmarks. For usability, HPCC FPGA adopts the following approaches:

- Usage of the same build system for all benchmarks (CMake) offers a unified user experience during the build process and for the modification of the configuration parameters.
- Integrated tests allow checking the functional correctness of the configuration using emulation before actual synthesis.

Every benchmark can be configured using several parameters that are specified during the build configuration with CMake's `-D` flag. They directly affect the performance of the kernel and thus need to be given together with the benchmark results to allow a more meaningful comparison. These parameters are summarized in Table 3.2 for all four benchmarks. An essential parameter for all benchmarks is `NUM_REPLICATIONS` which allows to automatically create copies of the kernels. It is heavily used by the four presented implementations to scale the benchmarks to the available resources of the target boards.

During the development of the base implementation of the suite, we used a Continuous Integration (CI) setup based on GitLab and Jacamar CI [46] that makes use of GitLab's custom executor model to schedule CI jobs on an HPC system via the system workload manager. We implemented this setup on the Noctua 1 system at PC^2 and the GitLab instance of Paderborn University. This setup enabled us to easily check the functionality of the benchmark suite and the build infrastructure and simplified the optimization process for the base implementations proposed in this chapter. We continuously checked the compatibility of changes in the source code with selected target devices to prevent over-optimization for a specific FPGA board. Similarly, this infrastructure could also be used to monitor the system and tool performance over time to detect potential problems and changes in the tool behavior early on.

Table 3.2: HPCC FPGA configuration parameters.

	Parameter	**Description**
All Benchmarks	`NUM_REPLICATIONS`	Replicates the benchark kernels the given number of times.
STEAM	`DATA_TYPE`	Data type used for host and device code.

Table 3.2: HPCC FPGA configuration parameters. *(continued)*

	`VECTOR_COUNT`	If > 1, OpenCL vector types of the given size are used in the device code instead of scalar values. Can be used to increase the wdth of the pipeline to better utilize the memory interfaces.
	`GLOBAL_MEM_UNROLL`	Loop unrolling factor for the inner loops in the device code. Similar to `VECTOR_COUNT` it can be used to increase the width of the compute pipeline.
	`DEVICE_BUFFER_SIZE`	Number of values that are stored in the local memory. Implicitly affects size of global memory bursts.
RandomAccess	`DEVICE_BUFFER_SIZE`	Number of values that are stored in the local memory. Can be used to relax dependencies between subsequent accesses to global memory in trade-off with a higher error rate.
FFT	`LOG_FFT_SIZE`	Logarithm of the FFT size that is calculated. Higher FFT sizes will consume more resources but achieve higher compute performance.
GEMM	`DATA_TYPE`	Data type used for host and device code.
	`BLOCK_SIZE`	Size of the symmetric matrix block that will be stored in local memory. Should have a sufficient size to allow full utilization of global memory read and write bursts and the calculation pipeline.
	`GEMM_SIZE`	Size of the symmetric matrix block that will be stored in registers. This will affect the number of DSPs used in the implementation and the actual calculation performance.
	`GLOBAL_MEM_UNROLL`	Used to specify the amount of unrolling done in loops that access the global memory to match the width of the memory interface.

3.1.2 STREAM Benchmark

The goal of the STREAM benchmark is to measure the sustainable memory bandwidth in GB/s of a device using four different vector operations: *Copy*, *Scale*, *Add* and *Triad*. All kernels are taken from the STREAM benchmark[1] v5.10 and thus slightly differ from the kernels proposed in the HPCC article [40]. The benchmark will sequentially execute the operations given in Table 3.3. *PCIe Read* and *PCIe Write* are no OpenCL kernels but represent the read and write of the arrays from the host to the device memory. The benchmark will output the maximum, average, and minimum time measured for each of these operations. For the

[1] https://www.cs.virginia.edu/stream

calculation of the memory bandwidth, the minimum time will be used.

Table 3.3: STREAM operations reported by the STREAM implementation for FPGA.

Operation	**Kernel Logic**
PCIe Write	write arrays from host to device memory
Copy	$C[i] = A[i]$
Scale	$B[i] = j \cdot C[i]$
Add	$C[i] = A[i] + B[i]$
Triad	$A[i] = j \cdot C[i] + B[i]$
PCIe Read	read arrays from device memory to host

Listing 3.1 Simplified logic of the STREAM benchmark

```
for(uint i=0; i<array_size; i+=BUFFER_SIZE){
    DATA_TYPE buffer[BUFFER_SIZE];
    for (uint k=0; k<BUFFER_SIZE; k++)
        buffer[k] = scalar * in1[i + k];
    if (second_input)
        for (uint k=0;k<BUFFER_SIZE; k++)
            buffer[k] += in2[i + k];
    for (uint k=0;k<BUFFER_SIZE; k++)
        out[i + k] = buffer[k];
}
```

The arrays A, B, and C are initialized with a constant value over the whole array. This allows us to validate the result by only recalculating the operations with scalar values. The error is calculated for every value in the arrays and must be below the machine epsilon $\epsilon > ||d - d'||$ to pass the validation.

A simplified version of the code is given in Listing 3.1. The OpenCL kernel of the benchmark combines all four described compute kernels in a single function. Since on FPGA the source code is translated to spatial structures that take up resources on the device, a single combined kernel yields the best reuse of those resources. The computation is split into blocks of a fixed length, which requires that the array length is a multiple of the block size. In the first inner loop, the input values of the first input array `in1` are loaded into a buffer located in the local memory of the FPGA. While they are loaded, they are multiplied with a scaling factor `scalar`, implementing the required logic for the *Copy* and *Scale* operation. In the case of *Copy*, the scaling factor is set to 1.0. In the second loop, the second input `in2` is only added to the buffer, if a flag is set. Together with the first loop, this makes it possible to recreate the behavior of the *Add* and *Triad* operation. In the third loop, the content of the buffer is stored in the output array `out` located in global memory.

`DATA_TYPE` and `VECTOR_COUNT` can be used to define the data type used within the kernel. If `VECTOR_COUNT` is greater than 1, OpenCL vector types of the given length are used. Since some of the STREAM operations show an asymmetry between reads and writes from global memory, the best performance can be achieved if a single bank is used for all three arrays. Simultaneous reads and writes to DDR or HBM2 banks would potentially destroy memory bursts and thus reduce performance. By using a local memory buffer, each of the three loops can fully utilize the available bandwidth of the global memory bank and the memory bursts. This makes STREAM bandwidth bound by the global memory and also resource bound by the BRAM. The kernel has to be replicated to fully utilize all available memory banks and at the same time the local memory buffer size has to be increased to achieve larger burst sizes and to decrease the impact of memory latency.

3.1.3 RandomAccess Benchmark

The random access benchmark measures the performance for non-consecutive memory accesses expressed in the performance metric Giga Updates Per Second (GUPS). It updates values in a data array $d \in \mathbb{Z}^n$ such that $d_i = d_i \oplus a$ where $a \in \mathbb{Z}$ is a value from a pseudo random sequence. n is defined to be a power of two. As data type, 64 bit integers are used. Since only operations in a finite field are used to update the values, the correctness of the updates can easily be checked by executing a reference implementation on the host side on the resulting data using the same pseudo random sequence. The incorrect items are counted, and the error percentage is calculated with $\frac{error}{n} \cdot 100$. An error of $< 1\%$ has to be accomplished to pass the validation. Hence, update errors caused by concurrent data accesses are tolerated to some degree. The performance of the implementation is mainly bound by the memory bandwidth and latency. So the kernel should be replicated to utilize all available memory banks.

The benchmark requires $4 \cdot n$ updates on the array of length n using two pipelines. The first pipeline reads data items from global memory. Therefore, it uses a pseudo random sequence generator that is implemented in every kernel replication. The index of the value will be extracted from the random number. Since every replication of the kernel is in charge of a distinct range of addresses that is placed in their memory bank, the kernel has to check if the calculated address is actually within its range. Only then the value will be loaded from global memory and stored in a local memory buffer whose size can be configured via the `DEVICE_BUFFER_SIZE` parameter. In the second pipeline, the data in local memory is written back to global memory. With this approach, read after write dependencies will be ignored for all values that are read into the local memory in the same iteration. Without the local memory, the tools would correctly identify the read after write dependency and only start the next read after the write is completed. Because of the benchmark rules, these kind of errors are allowed to some degree, so changing the buffer size can be used as a trade-off between exptected benchmark error and performance.

Listing 3.2 Random Access simplified update loop

```
for(uint i=0; i<4*n; i+=BUFFER_SIZE){
  ulong local_address[BUFFER_SIZE];
  ulong loaded_data[BUFFER_SIZE];
  ulong update_val[BUFFER_SIZE];

  //calc next address and load to local mem.
  for (uint ld=0; ld<BUFFER_SIZE; ld++){
    ulong rand = next_rand();
    update_val[ld] = rand;
    local_address[ld] = get_address(rand);
    if (in_range(local_address[ld]))
     loaded_data[ld] = data[local_address[ld]];
  }

  //store the updated value back to global mem.
  for (uint ld=0; ld<BUFFER_SIZE; ld++){
    if (in_range(local_address[ld]))
      data[local_address[ld]] =
            loaded_data[ld]^update_val[ld];
  }
}
```

3.1.4 Fast Fourier Transformation Benchmark (FFT)

In the FFT benchmark, a batch of 1D FFTs of a predefined size – that can be specified using the `LOG_FFT_SIZE` parameter – is calculated. A batch of FFTs is used to increase the overall execution time of the benchmark, to decrease measurement errors, and to better utilize the kernel pipeline. The benchmark kernel is based on a reference implementation for the Intel OpenCL FPGA SDK included in version 19.4.0 and slightly modified to also allow execution on Xilinx devices. The calculation is done with complex single-precision floating point values. All stages of the FFT calculation are fully unrolled and the current data is stored in a shift register that is shared between the stages. This allows the implementation of the algorithm in a single pipeline but also limits the maximum size of the FFT by the available FPGA resources and SDK capabilities. Larger FFT sizes require more DSPs for the calculations, and more logic resources and BRAMs to implement the shift register. Since reads and writes from global memory are symmetric, the kernel offers the best performance if the input and output data can be stored in different memory banks. So the kernel replications should be set at most to half of the available memory banks. Also, the best trade-off between kernel replication and FFT size has to be used to achieve the best performance.

For the performance metric GFLOP/s, the floating point operations of this benchmark are calculated as $5 \cdot n \cdot ld(n)$ for an FFT of dimension n. The result of the calculation is checked by calculating the residual $\frac{||d-d'||}{\epsilon \cdot ld(n)}$ where ϵ is the machine epsilon, d' the result from the reference implementation and n the FFT size.

3.1.5 Matrix Multiplication Benchmark (GEMM)

The GEMM benchmark implements a matrix-matrix multiplication similar to the GEMM routines in the BLAS library. It calculates $C = \alpha \cdot A \cdot B + \beta \cdot C$ where $A, B, C \in \mathbb{R}^{n \times n}$ and $\alpha, \beta \in \mathbb{R}$. For the performance metric GFLOP/s, the floating point operations of this benchmark are calculated as $2 \cdot n^3$. The result is validated by calculating the normalized residual $\frac{||C-C'||}{\epsilon \cdot n \cdot ||C||_F}$ where ϵ is the machine epsilon and C' the result of the reference implementation. The implementation is based on a matrix multiplication design for Intel Stratix 10 [47] and simplified to make it compatible with a broader range of devices. The kernel creates a memory hierarchy for the matrix multiplication $A \times B$ by doing the following data movements:

- A blocked matrix multiplication in global memory
- A blocked matrix multiplication in local memory (BRAM)
- A fully unrolled matrix multiplication in registers

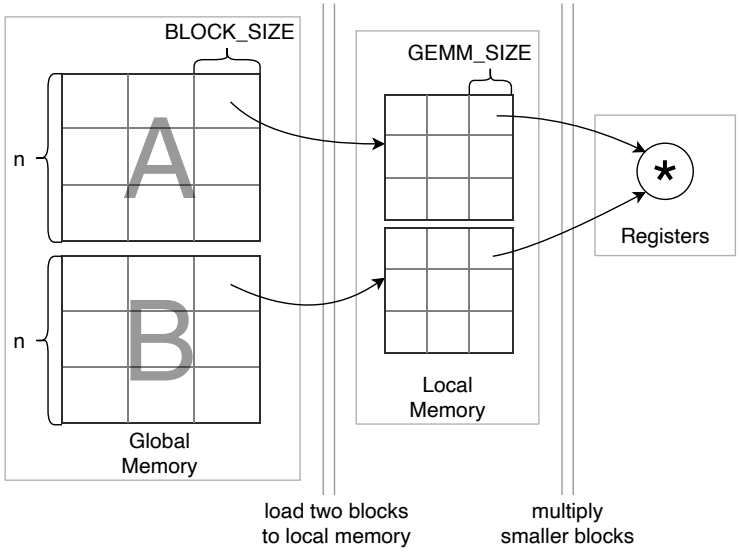


Figure 3.2: Three-level data movement from global memory to compute logic in GEMM.

A visualization of the data movements is given in Figure 3.2. The matrices are divided into blocks of a fixed size that can be specified with the `BLOCK_SIZE` parameter. The kernel calculates the result of a single block in C by sequentially executing two pipelines. In the first pipeline, a single block of A and B is loaded into a local memory buffer. In the second pipeline

the loaded blocks are used to calculate an intermediate result again with a blocked approach. This time the block size is defined by the parameter `GEMM_SIZE`. The matrix multiplication for this block is fully unrolled which allows to initiate the multiplication of a whole block every clock cycle. Both pipelines will be executed sequentially to calculate the result of a single block of $A \times B$.

After the matrix multiplication is done, a third pipeline reads the block of matrix C directly from global memory and calculates the final result for a block of the output matrix. The number of values that are read per clock cycle is configurable using the parameter `GLOBAL_MEM_UNROLL`. The final result $C = \alpha \cdot A \cdot B + \beta \cdot C$ is calculated for each value of C and the result of the matrix multiplication and direclty written back to the result matrix in global memory.

This means the data is loaded block-wise from global memory to local memory and from there in smaller blocks to registers to perform the actual matrix multiplication. Writing the data back to global memory and executing the addition and scaling in the third pipeline does not have a huge impact on the overall performance for huge matrices because this just has to be done every $\frac{n}{BLOCK\_SIZE}$ iterations.

The host code divides the output matrix into rows of blocks of size `BLOCK_SIZE` and distributes the calculation over all kernel replications. Since the kernel is compute-bound, the second pipeline should be scaled as large as possible or the kernel should be replicated to increase the parallelism of the calculation. This makes the implementation resource bound in terms of DSPs for the calculation pipeline and BRAMs for the local memory buffer. Loading and storing data from global memory just takes up a minor role in the implementation. Thus, the kernel replications should be used to fill the FPGA instead of utilizing all memory banks.

3.2 Evaluation of Single FPGA Benchmarks

In the following, we synthesize and execute all four benchmarks, collect first benchmark results for the proposed base implementations, and evaluate the results to performance models.

3.2.1 Evaluation Environment and Configuration

The benchmarks were synthesized and tested on four different boards: *Nallatech/BittWare 520N*, *BittWare 520n MX*, *Intel PAC D5005* and *Xilinx Alveo U280*. The BittWare 520N boards are connected to the host node via x8 PCIe 3.0 and are equipped with Intel Stratix 10 GX2800 with access to four banks of DDR4 SDRAM x 72 bit with 8 GiB per bank and a transfer rate of 2400 MT/s. With the BSP version 19.4.0 and SDK version 19.4.0 the most recent versions available at the time of writing are used for synthesis and execution.

The BittWare 520N MX contains the Intel Stratix 10 MX2100 using 16 GiB of HBM2 memory. For the host-device communication, x8 PCIe 3.0 is used. BSP version 19.4.0 and SDK version

20.4.0 were used to collect the measurement result.

The Intel PAC D5005 board hosts a Stratix 10 SX2800 FPGA and is connected to the host with x16 PCIe 3.0. The board contains a DDR4 memory infrastructure similar to the other boards, which is not used for these experiments. Instead, a reference design BSP (18.1.2_svm) was used, offering direct access to the host's memory using SVM by building upon the Intel Acceleration Stack (IAS) [48] version 1.2. The OpenCL kernel compilation was performed with the SDK version 19.4.0.

The Alveo U280 boards are equipped with the XCU280 FPGA. The board is connected to an Intel Xeon Gold 6234 CPU over x8 PCIe 3.0 and equipped with two banks of DDR4 SDRAM x 72 bit with 16 GiB per bank and a transfer rate of 2400 MT/s. Moreover, the FPGA is equipped with 8 GiB of HBM2 split over 32 banks. Xilinx Vitis is used in version 2019.2 and the shell version is 2019.2.3. The host code is compiled with GCC 7.4.0 and CMake 3.18 is used for the configuration and build of the benchmarks. The CPU has access to 108 GiB of main memory.

Table 3.4: Final synthesis configuration for all benchmarks.

Benchmark	Parameter	520N	520N MX	U280 DDR	U280 HBM2	PAC SVM
STREAM	NUM_REPLICATIONS	4	32	2	32	1
	DATA_TYPE	float	float	float	float	float
	GLOBAL_MEM_UNROLL	1	1	1	1	1
	VECTOR_COUNT	16	8	16	16	16
	DEVICE_BUFFER_SIZE	4,096	4,096	16,384	2,048	1
RandomAccess	NUM_REPLICATIONS	4	32	2	32	1
	DEVICE_BUFFER_SIZE	1	1	1,024	1,024	1,024
FFT	NUM_REPLICATIONS	2	16	1	15	1
	LOG_FFT_SIZE	17	7	9	5	17
GEMM	NUM_REPLICATIONS	5	4	3	3	5
	DATA_TYPE	float	float	float	float	float
	GLOBAL_MEM_UNROLL	16	8	16	16	16
	BLOCK_SIZE	512	256	256	256	512
	GEMM_SIZE	8	8	8	8	8

For the Intel devices, the host node is a two-socket system equipped with an Intel Xeon Gold 6148F CPU and 192 GiB of DDR4-2666 main memory. The host codes are compiled with GCC 8.3.0 and CMake 3.15.3 is used for the configuration and build of the benchmarks.

Table 3.5: Resource utilization of the synthesized benchmark kernels.

Benchmark	Board	LUTs	Registers	BRAM	DSPs	Frequency [MHz]
STREAM	520N	176,396 (25%)	449,231 (25%)	4,029 (34%)	128 (2%)	316.67
	520N MX	261,485 (44%)	721,211 (44%)	3,634 (53%)	512 (13%)	400.00
	U280 DDR	20,832 (1.90%)	39,002 (1.39%)	558 (34.19%)	160 (1.78%)	300.00
	U280 HBM2	331,904 (20.69%)	574,976 (27.24%)	1,408 (77.70%)	2,560 (28.38%)	370.00
	PAC SVM	103,628 (12%)	244,354 (12%)	548 (5%)	32 (<1%)	346.00
Random Access	520N	115,743 (18%)	253,578 (18%)	489 (4%)	14 (<1%)	329.17
	520N MX	212,838 (38%)	608,321 (38%)	1,970 (29%)	112 (3%)	477.78
	U280 DDR	7,256 (0.65%)	11,716 (0.50%)	38 (2.23%)	14 (0.16%s)	446.00
	U280 HBM2	116,096 (10.68%)	187,456 (8.76%)	608 (33.55%)	224 (2.48%)	450.00
	PAC SVM	103,397 (12%)	225,293 (12%)	535 (5%)	0 (0%)	322.00
FFT	520N	276,676 (36%)	724,790 (36%)	4,177 (36%)	1,414 (25%)	413.34
	520N MX	458,087 (78%)	1,408,937 (78%)	2,066 (30%)	2,944 (74%s)	395.84
	U280 DDR	83,494 (7.39%)	168,150 (7.19%)	39 (2.28%)	672 (7.46%)	248.00
	U280 HBM2	602,125 (54.13%)	941,404 (42.18%)	405 (22.35%)	5,280 (58.58%)	254.00
	PAC SVM	192,189 (22%)	480,285 (22%)	2,147 (18%)	707 (12%)	348.00
GEMM	520N	275,754 (36%)	861,277 (36%)	8,860 (76%)	3,398 (59%)	160.42
	520N MX	213,337 (37%)	600,546 (37%)	2,388 (35%)	2,654 (67%)	240.00
	U280 DDR	568,558 (51.87%)	441,602 (19.43%)	666 (43.11%)	7,683 (85.23%)	100.00
	U280 HBM2	499,002 (42.64%)	920,127 (38.70%)	666 (36.71%)	7,683 (85.18%)	236.00
	PAC SVM	299,427 (33%)	829,802 (33%)	9,041 (77%)	3,398 (59%)	225.00

The used synthesis parameters for all benchmarks are given in Table 3.4 and the resulting resource usage for each benchmark is given in Table 3.5. A straightforward comparison of the resource usage between the FPGA boards is not possible because hardware and software architectures are different. Nevertheless, in the table a general overview of the resource usage is given by looking at the basic resource elements of an FPGA: The Look Up Table (LUT), Registers, BRAM and DSP. The table only takes into account the resources directly used for the kernels. Next to the absolute value, the percentage of the used resources relative to the available resources is given. Absolute values and ratios of the resource usage are taken directly from the reports generated by the HLS tools.

3.2.2 STREAM and RandomAccess

Table 3.6: Measurement results for all benchmarks.

Benchmark	**520N**	**520N MX**	**U280 HBM2**	**U280 DDR**	**PAC SVM**
STREAM Copy [GB/s]	67.01	308.35	377.42	33.94	20.15
STREAM Scale [GB/s]	67.24	308.24	365.80	33.92	20.04
STREAM Add [GB/s]	68.90	323.92	374.03	34.58	15.04
STREAM Triad [GB/s]	68.90	323.90	378.88	34.57	15.12
STREAM PCIe read [GB/s]	6.41	6.34	6.66	5.68	–
STREAM PCIe write [GB/s]	6.32	6.21	6.03	5.47	–
RandomAccess [MUOP/s]	245.0	469.8	128.1	40.3	0.5
RandomAccess Error	0.0099%	0.0106%	0.0106%	0.0106%	0.0106%
FFT [GFLOP/s]	349.45	713.95	576.20	78.26	119.66
FFT [GB/s]	65.78	326.38	368.77	27.83	22.54
FFT Error	7.1e-01	4.6e-01	5.4e-01	3.9e-01	7.1e-01
GEMM [GFLOP/s]	708.95	765.23	603.86	266.91	739.59
GEMM norm [GFLOP/s]	88.39	79.71	85.29	88.97	65.74
GEMM Error	6.0e-07	2.4e-07	2.0e-06	2.0e-06	6.0e-07

The measured performance results for all four benchmarks are given in Table 3.6. Single-precision floating point numbers are used in the STREAM benchmark. For the size of the STREAM data array, we used 2^{29} items, which corresponds to 2 GiB of data per data array. This is the largest power of two that fits into the 8 GiB HBM2 of the Alveo U280 board.

The theoretical peak performance for DDR memory is 19.2 GB/s per bank for both the 520N and the U280 boards. That said, the STREAM benchmark achieves an efficiency between 87.2% and 90.1% of the theoretical peak performance for both devices for all four operations. The HBM2 of the U280 board offers a theoretical peak bandwidth of 460 GB/s, so the benchmark achieves an efficiency of 82.4% on this board. This is similar to the 520N MX, where the efficiency is around 79.1%. One reason for the lower efficiency is the use of smaller local memory buffers because of the high number of kernel replications. This will also lead to

shorter read and write bursts which directly affect the memory bandwidth. Another reason is the simultaneous scheduling of the kernels on the FPGA. For the execution of a single kernel on the 520N MX, up to 11.18 GB/s can be measured. This is more than 87.3% efficiency for a single HBM2 bank. Since the kernels will be scheduled sequentially over PCIe, the host can not fully prevent this effect since it will measure the execution time from starting the first kernel to finishing the last kernel. With very short execution times, this can have a visible effect on the measured performance for a high number of kernel replications.

The full-duplex PCIe connection of the PAC SVM board eliminates the need for a large local memory buffer. Instead, it is set to the same size as the unrolling (`GLOBAL_MEM_UNROLL`) which fully unrolls the three inner read- and write- pipelines and results in a single pipeline that allows to simultaneously read and write from global memory. For the *Copy* and *Scale* operation the kernel achieves more than 20 GB/s. The *Add* and *Triad* operations need two input parameters, which leads to the PCIe read channel being the bottleneck and only half of the write channel's bandwidth can be utilized.

For the RandomAccess benchmark, a data array of 2^{29} items, which is a total of 4 GiB, is equally split onto the available memory banks. All kernels have to calculate all addresses and only update the value if it is placed within the range of the data array they are assigned to. This leads to a compute-bound implementation for a high number of kernel replications since the pipeline stalls caused by memory accesses per kernel will decrease. Still, the amount of random numbers that have to be generated stays the same. So the maximum achievable updates per second are limited by the kernel frequency. The measured performance will show a higher gap to the maximum performance for designs with only a small amount of replications because the random memory accesses will consume a considerable amount of time. The RandomAccess benchmark shows huge performance differences between the used boards. One reason for that is the difference in the kernel design that allows the creation of a single pipeline in the case of the 520N and 520N MX board using the Intel-specific *ivdep* pragma. This optimization also slightly decreased the calculation error that is introduced by the buffer. Nevertheless, for the PAC SVM tests this optimization had to be disabled because with 98% the error drastically exceeds the allowed range because of the latency increase for global mmeory accesses. For Xilinx toolchains there is to our knowledge no optimization flag similar to *ivdep*.

3.2.3 FFT

The FFT benchmark calculates the 1D FFT of different sizes specified in the configuration in a batched manner. The batch size is chosen such that the total size of the data is 2 GiB. This allows the kernel to fill the pipeline and hide the latency of the calculation. Eight values are loaded and stored in global memory per clock cycle and the calculation is implemented in a single pipeline. For every value, `LOG_FFT_SIZE` complex floating-point multiplications are calculated, which corresponds to five single-precision floating-point operations per value.

Eight values are loaded from global memory per clock cycle so the expected performance of the kernel can be modeled with the Equation 3.1.

$$p_{FFT} = 5 \cdot s \cdot f \cdot r \cdot 8 \tag{3.1}$$

where s is `LOG_FFT_SIZE`, r is `NUM_REPLICATIONS` and f the minimum of the kernel frequency and the memory interface frequency.

The model shows, that the FFT size and the number of kernel replications have a similar impact on the expected performance. Since every kernel exclusively needs two memory banks, the kernel replications are chosen after the number of available memory banks on the FPGA. In the second step, the FFT size is increased to fill the FPGA. The configuration for the U280 board with DDR is an exception in this regard because it does not fully utilize the FPGA resources which again shows a weakness of the current FFT implementation: The implementation makes use of a large shift register to store and delay data that is read from global memory. With the current Xilinx Vitis tools, the largest supported shift register size is 1024, which is too small for FFT sizes for 2^{10} and beyond. Also, the achieved kernel frequencies are too low to fully utilize the memory on the U280 board which has to be considered when comparing the results to STREAM.

Besides that, the kernel shows a high efficiency of more than 85% for all FPGA which is close to the STREAM results. Also, the PAC board shows a very high memory bandwidth for the FFT kernel compared to STREAM. This indicates that large memory bursts have a positive impact on the achieved memory bandwidth when using SVM.

3.2.4 GEMM

Similar to STREAM and FFT, single-precision floating point is used as data type in GEMM since it is well supported among all used devices. The GEMM benchmark uses 4096×4096 matrices for the calculation. This is the biggest matrix size that fits into a single HBM2 bank. The actual calculation is done on 8×8 matrices for all boards defined by the `GEMM_SIZE` parameter. So the kernel can initialize the calculation of 1024 floating-point multiplications and additions per clock cycle. This is the largest value for `GEMM_SIZE` that fits on the FPGAs. The kernel replications and `BLOCK_SIZE` is chosen to utilize the FPGA resources as much as possible while still providing an acceptable kernel frequency.

Because the kernel contains three pipelines that are executed sequentially, the total execution time of the kernel can be modeled by the sum of the three pipelines as given in Equation 3.2.

$$t_{exe} = \frac{b^2}{u \cdot f_{mem}} + \frac{b^3}{g^3 \cdot f_k} + \frac{b^2}{u \cdot \frac{n}{b} \cdot f_{mem}} \tag{3.2}$$

where b equals `BLOCK_SIZE`, r equals `NUM_REPLICATIONS`, u equals `GLOBAL_MEM_UNROLL`, g equals `GEMM_SIZE`.

Moreover, n is the total matrix size, f_k is the kernel frequency and f_{mem} is the frequency of the memory interface.

For the platforms with on-board memory resources (DDR and HBM), the measured performance deviates not more than 1.8% from the model performance. Only the SVM platform performs 23% slower than the model. Simultaneous memory accesses of the different kernel replications as well as the decreased memory burst sizes caused by the blocked reads are likely bottlenecks during the global memory access phases. However, due to the higher clock frequency, the overall performance of the SVM design is still slightly higher than the very similar design on 520N DDR.

Since the GEMM benchmark is computation-bound, the performance is highly dependent on the kernel frequency and less on the memory interface. We also provide a performance result normalized to 100 MHz and a single kernel replication in the table to make the performance easier to compare between all boards. The normalized performance of the 520N and U280 is close together which shows that the tools for both boards can generate hardware from the code with a similar performance – indepenedent of the final clock speed. For the U280 with DDR memory, we had to set the target frequency to 100 MHz to achieve a successful synthesis. Reasons for that are routing problems and not satisfied timing constraints because of the additional logic needed for the DDR memory interface. In contrast, when using HBM2, the place and route of three replications worked seamlessly and resulted in a considerably higher clock frequency.

3.3 Capturing Device Characteristics using Single-FPGA Benchmarks

In the previous section, we presented benchmark measurements on different FPGA platforms, with different shells and SDK versions and created simple performance models for the benchmarks. Now, we go one step further and show how the benchmark suite allows capturing specific characteristics of the used FPGA boards and tools.

3.3.1 Impact of Kernel Frequencies

An additional synthesis for the 520N board with an increased local buffer size to 16,384 resulted in the resource usage and performance given in Table 3.7. The high BRAM usage eventually leads to a reduced kernel frequency of 280 MHz, which is 93.34% of the frequency of the memory controller. At the same time, the benchmark achieves up to 92.15% of the performance compared to the kernel running with more than 300 MHz, which correlates

with the frequency reduction. This shows that the benchmark is also capable of measuring the performance of the board and tools since the kernel frequency is not least dependent on the place and route of the kernel components by the compiler. Another insight seen in the measurement results is the difference in the performance of *Copy/Scale* and *Add/Triad*. Since the kernel has a lower frequency than the memory controller, the kernel itself becomes the performance bottleneck. Execution overheads introduced by switching the pipelines now directly affect the kernel performance and are no longer hidden by latencies of the slower memory controller.

Table 3.7: STREAM benchmark results on the 520N board with 1 MiB local memory buffer.

Synthesis		**Execution**	
LUTs	203,607 (26%)	**Copy**	63.48 GB/s
FFs	436,516 (26%)	**Scale**	63.49 GB/s
BRAM	7,409 (63%)	**Add**	58.96 GB/s
DSPs	128 (2%)	**Triad**	59.00 GB/s
Freq.	280.00 MHz	**PCIe Write**	6.40 GB/s
		PCIE Read	6.32 GB/s

3.3.2 Impact of Data Type on the Benchmark Performance

As mentioned in Section 3.2, STREAM and GEMM can be configured to use Floating-Point (FP) values of different precision. The format of these data types is defined in the OpenCL specification [7]. Depending on the application scenario, on the one hand, it can be beneficial to calculate results with higher accuracy by using 64 bit Double Precision (DP) FP. On the other hand, a lower accuracy can be sufficient in other scenarios, so 16 bit Half Precision (HP) FP numbers can be used. For FPGA, especially the latter can be beneficial since it may also come with a resource reduction for the design. The higher amount of unused resources is an advantage in the place and route phase to achieve higher clock frequencies or allow further scaling of the design. To investigate the impact of varying FP precision, we synthesized and executed STREAM and GEMM additionally in HP and DP on the 520N and U280 boards. The configuration was modified for the three data types as shown in Table 3.8. Since the OpenCL specification does not offer a vector data type for HP, the scalar type and loop unrolling had to be used to match the memory interface width by increasing the `GLOBAL_MEM_UNROLL` parameter. This change required additional changes in the STREAM kernel to enable memory bursts on the Xilinx board. For an easier comparison of the results, also the Single Precision (SP) results of the updated STREAM kernel and for GEMM are given in this section.

Table 3.8: Benchmark configurations for different floating-point types.

		520N			U280 HBM2		
Benchmark	**Parameter**	**HP**	**SP**	**DP**	**HP**	**SP**	**DP**
STREAM	NUM_REPLICATIONS	-	4	4	32	32	32
	DATA_TYPE	-	float	double	half	float	double
	GLOBAL_MEM_UNROLL	-	1	1	32	1	8
	VECTOR_COUNT	-	16	8	1	16	1
	DEVICE_BUFFER_SIZE	-	65,536	32,768	65,536	2,048	16,384
GEMM	NUM_REPLICATIONS	2	5	2	3	3	6
	DATA_TYPE	half	float	double	half	float	double
	GLOBAL_MEM_UNROLL	32	16	8	32	16	8
	BLOCK_SIZE	512	512	512	512	256	128
	GEMM_SIZE	4	8	4	8	8	4

Not all features defined in the OpenCL specification for HP and DP are implemented in the SDKs. With the used Intel SDK version it is not possible to pass a HP FP value as an argument to the kernel. Instead, only pointers to global memory are allowed for this data type. This restriction makes it impossible to synthesize the current base implementation of the STREAM benchmark with HP because the kernel requires to pass the scalar value used for the *Scale* and *Triad* kernel. A fix would require the modification of the kernel signature, which is not allowed for the base implementation. The Xilinx tools do not support vector types for DP. Instead, loop unrolling was used similar to the HP implementation.

The resource utilization of the resulting bitstreams is given in Table 3.9 and the measurement results in Table 3.10. The HP version of STREAM for the U280 shows an increase in logic resources and a decrease in DSP usage. Only the addition is implemented using DSP, where still two DSP are utilized per addition. The tool heuristics favor the implementation of HP multiplications purely in logic, which leads to an increased use of flip-flops and LUT. Overall, the measured performance for the HP and DP versions is lower than the SP performance although all versions use clock frequencies way above the 225 MHz which are necessary to fully utilize the global memory interface. The performance measurements for STREAM on the 520N only show small differences for both precisions. For the DP version, a smaller local memory buffer is used because synthesis with a buffer of the same size as the SP version failed.

In the case of GEMM, the resource-efficient implementation of floating-point operations is more important. On the U280, the resource usage with the HP version decreases as expected. Multiplications are implemented using DSPs which requires fewer logic resources. In the case of DP, the number of required DSPs would exceed the number of available DSPs. To fit the kernels on the FPGA, the `GEMM_SIZE` is reduced to 4 while increasing the number of replications to 6. The high logic utilization prevents further scaling of the number of kernel

replications for this design.

Table 3.9: Benchmark resource utilization for different floating-point precisions.

Benchmark	Board	Precision	LUTs	Registers	BRAM	DSPs	Frequency [MHz]
STREAM	520N	SP	150,827 (20%)	333,381 (20%)	7,169 (61%)	128 (2%)	312.50
		DP	182,213 (25%)	423,430 (25%)	3,841 (33%)	128 (2%)	322.50
	U280 HBM2	HP	454,848 (42.58%)	874,432 (41.43%)	1,408 (77.70%)	2,048 (22.71%)	341.00
		SP	297,472 (27.84%)	574,976 (27.24%)	1,408 (77.70%)	2,560 (28.38%)	360.00
		DP	382,848 (35.84%)	731,456 (34.66%)	1,408 (77.70%)	2,816 (31.22%)	345.00
GEMM	520N	HP	420,188 (44%)	879,834 (44%)	3,657 (31%)	286 (5%)	208.63
		SP	275,754 (36%)	861,277 (36%)	8,860 (76%)	3,398 (59%)	160.42
		DP	497,462 (55%)	1,094,926 (55%)	6,381 (54%)	495 (9%)	216.68
	U280 HBM2	HP	364,614 (31.78%)	680,277 (29.11%)	1,242 (68.47%)	6,147 (68.15%)	243.00
		SP	499,002 (42.64%)	920,127 (38.70%)	666 (36.71%)	7,683 (85.18%)	236.00
		DP	553,260 (50.10%)	957,318 (43.17%)	758 (41.72%)	4,230 (46.90%)	241.00

On the 520N, most of the floating-point operations are implemented in logic for both the HP and DP versions because the DSPs in the Intel FPGAs are optimized for SP operations. This results in similar, small configurations for HP and DP and with that to a heavily decreased performance compared to the SP configuration. However, it has to be noted here that the base implementation for single precision only makes use of 60% of the available DSP on the Intel devices whereas for the Xilinx devices, it is above 85%. More optimized implementations like [47] can achieve a higher resource utilization and also a higher performance for SP on these devices of more than 1 TFLOP/s. This shows, that the DSP in the Xilinx FPGAs are more adaptable to different precisions, but an optimized SP design has a higher potential to achieve better performance on the Intel devices.

Table 3.10: Benchmark results for different floating-point precisions.

	520N			U280 HBM2		
Benchmark	**HP**	**SP**	**DP**	**HP**	**SP**	**DP**
STREAM Copy [GB/s]	-	68.40	68.10	362.35	385.97	353.23
STREAM Scale [GB/s]	-	68.40	68.11	363.80	386.67	354.70
STREAM Add [GB/s]	-	69.80	69.58	367.19	389.24	358.22
STREAM Triad [GB/s]	-	69.80	69.58	367.46	389.61	358.74
STREAM PCIe read [GB/s]	-	6.47	6.47	7.33	7.08	6.60
STREAM PCIe write [GB/s]	-	6.30	6.30	6.39	6.67	6.23
GEMM [GFLOP/s]	51.19	708.95	52.07	684.03	603.86	146.92
GEMM norm [GFLOP/s]	12.27	88.39	12.02	93.83	85.29	10.16
GEMM Error	9.6e-8	6.0e-7	5.7e-7	1.8e-5	2.0e-6	1.1e-6

3.3.3 Kernel Scheduling on the U280 with HBM2

In initial measurements, the U280 board showed a very low efficiency of below 30% for all operations of the STREAM benchmark so we further investigated the kernel performance with additional experiments. The STREAM benchmark measures the time from starting the first kernel until the last kernel has finished execution to calculate the bandwidth. This is similar to using the slowest runtime of all kernels to calculate the bandwidth. For a more detailed look at the performance of the benchmark, we used the profiling information provided by the OpenCL API to collect the execution times separately for every kernel during the benchmark execution. The used profiling events are the kernel start and end times. The observed kernel execution times for the *Copy* operations are given in Figure 3.3 in groups of 15 kernels with regards to the chronological sequence they were enqueued for execution. For this measurement, the kernels were enqueued sequentially in two different orders, and the measurements were repeated 20 times. The second and third groups show approximately double and three times the execution time of the first 15 kernels. This leads to the hypothesis that only 15 kernel executions can be maintained simultaneously and that the reported kernel execution time also contains the wait time until the kernel is started. With a modified *Copy* kernel that uses two HBM2 banks, one for each array, we were able to measure 373 GB/s using 15 kernel replications. At the same time, we experienced a similar performance drop for the execution with 16 replications. Thus, the effect seems to be independent of the utilization of the global memory.

This insight triggered further research into the kernel scheduler of the Xilinx OpenCL runtime system, which actually contains three scheduler implementations. After disabling the two scheduler variants *ERT* and *KDS* in a configuration file that at this time don't support more than 15 concurrent kernels, execution falls back to a scheduler that does not contain these limitations. This insight illustrates the importance of a benchmark suite defined on the

OpenCL level that measures not only the raw performance potential of FPGA hardware, but also the impact of compilation and synthesis tools and, in this case, runtime environments.

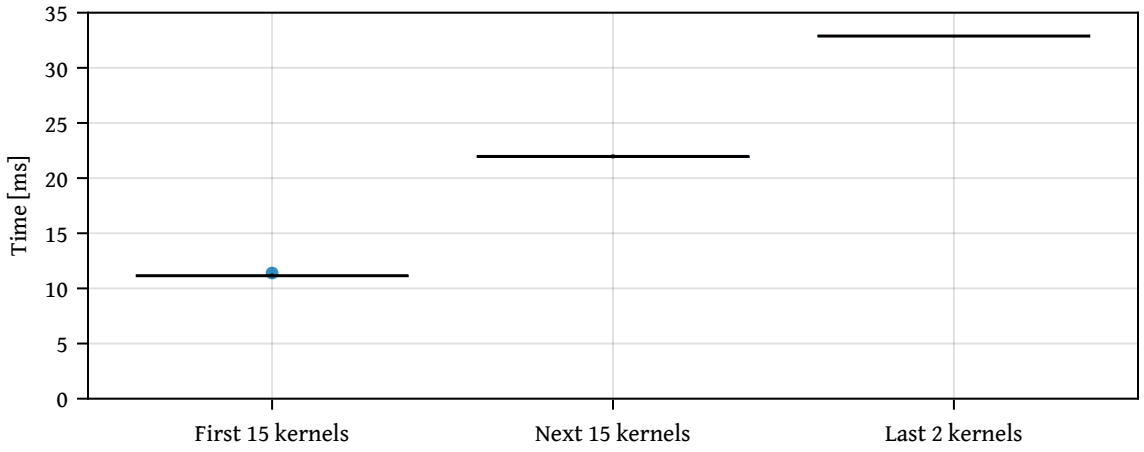


Figure 3.3: Kernel execution times for the Copy operation on the U280 board with HBM2 memory using 32 kernel replications. The execution times are combined in blocks of 15 kernels in the order they were enqueued for execution.

3.3.4 Power Measurements on FPGA and CPU

Another, increasingly important metric in HPC is power efficiency. By default, the presented HPCC FPGA benchmarks do not measure the power consumption since there is no standardized way to retrieve this data. Nevertheless, we created measurement scripts for each FPGA to measure the power consumption of STREAM and GEMM to investigate the power efficiency of recent FPGA boards during execution. The scripts measured the power consumption during the benchmark execution in intervals of 100 ms, and we calculated the average over the entire execution time. To reduce the FPGA idle time during the measurement, we increased the array sizes to the biggest power of two that fit into the device's memory and increased the number of repetitions to 100 in the case of STREAM. For GEMM, the matrix width is set to 23,040 to match the block sizes of the FPGA implementations. For the HBM2 boards, the matrix size has to be reduced to the largest matrix that fits into a single memory bank. In the case of the U280 with HBM2 this is 8,192 and for the 520N MX 11,264.

As a comparison, we executed STREAM v5.10 on the two-socket CPU system equipped with Intel Xeon Gold 6148F which is also used as the host for the BittWare 520N and PAC D5005. We measured the CPU + DDR power consumption using Likwid in a 10 s interval during execution. For the GEMM CPU implementation, we used the `sgemm` routine of Intel MKL 2020.2.254.

Table 3.11: Power consumption vs benchmark results for the STREAM benchmark.

	520N	**520N MX**	**U280 HBM2**	**U280 DDR**	**PAC SVM**	**2x Xeon**
Average [W]	68.82	54.93	41.44	36.15	54.43*	323.29
Peak [W]	76.78	73.8	59.98	47.09	55.21*	344.91
Performance per Watt [GB/(Ws)]	0.91	4.61	6.31	0.74	0.37*	0.53
Copy [GB/s]	69.09	331.70	372.27	34.06	20.43	167.15
Scale [GB/s]	69.10	331.94	374.82	34.05	20.43	173.44
Add [GB/s]	70.08	340.34	378.51	34.67	15.10	183.99
Triad [GB/s]	70.02	340.41	378.55	34.66	15.12	183.40
PCIe read [GB/s]	6.40	6.30	7.46	5.44	–	–
PCIe write [GB/s]	6.30	6.15	7.18	5.47	–	–

*Additional power consumed by using host DDR memory not included

Table 3.12: Power consumption vs benchmark results for the GEMM benchmark.

	520N	**520N MX**	**U280 HBM2**	**U280 DDR**	**PAC SVM**	**2x Xeon**
Average [W]	81.67	54.90	50.32	65.85	71.35*	281.49
Peak [W]	87.38	64.73	71.48	72.42	80.39*	362.10
Performance per Watt [GFLOP/(Ws)]	8.33	11.93	8.61	3.74	8.57*	12.57
GFLOP/s	727.58	771.99	615.14	270.87	689.31	4,552.19

*Additional power consumed by using host DDR memory not included

Table 3.11 presents the power consumption along with the benchmark performance of the FPGA targets and CPU reference for STREAM. The power consumption of the benchmark execution is given in the upper part of the table. The average power consumption is calculated for the whole benchmark execution time. It needs to be noted that for all FPGA executions except for the PAC SVM, this also includes buffer transfer times for every benchmark repetition. During this time the FPGA is nearly idle and consumes less power. Therefore, the performance per watt is calculated using the peak measured bandwidth divided by the peak power consumption. This metric can be used to compare the power efficiency of the devices during the execution of STREAM. FPGA platforms using on-board DDR memory are up to 1.7x more power efficient than the CPU reference during this memory-intensive benchmark, the FPGAs with HBM2 show a more than 8x higher performance per Watt than the CPU reference.

The measurements for GEMM in Table 3.12 show a better power efficiency of the CPU compared to all FPGAs. However, it has to be noted that this table compares a generalized OpenCL implementation for Intel and Xilinx FPGAs with the highly optimized MKL library for Intel CPUs that includes customized code paths for the AVX512 instruction set of the

benchmarked CPU type. Especially for the Intel FPGAs the base version fails to fully utilize the available DSPs. Implementations like [47] have shown that it is possible to achieve considerably better performance with optimized kernels, which most likely also results in better power efficiency.

3.4 Chapter Conclusion

In this chapter, we proposed *HPCC FPGA*, a novel FPGA OpenCL benchmark suite for HPC. Therefore, we provide configurable OpenCL base implementations and host codes of four selected benchmarks of the well-established HPCC benchmark suite. We showed that the configuration options allow the generation of efficient benchmark kernels for Xilinx and Intel FPGAs using the same source code without manual code modification. We executed the benchmarks on four FPGA boards with three different memory setups including HBM2 variants for Intel and Xilinx FPGAs and compared the results with simple performance models. The evaluation showed that, for most of the benchmarks, the measured performance is close to the modeled peak performance under resource or global memory constraints of the FPGA board. Additionally, the execution of two of the benchmarks with different floating-point precisions gives insight into their support on the tested device families. In these cases, the benchmarks can start to serve as a deployment criterion for FPGA accelerated systems.

Nevertheless, the evaluation also showed that some of the base implementations are unable to fully utilize the available resources or produce low kernel frequencies because we were not able to write code that works well on all tested devices. This can also be seen in the power measurements, where the base implementations do not always succeed in achieving a better power efficiency than the corresponding CPU implementation. However, the base implementations are always a trade-off between compatibility with as many devices as possible and performance. In these cases, it may be required to also provide optimized implementations for specific devices that adhere to the benchmark rules to increase the significance of the benchmark results. The benchmark suite comes with a code structure and build infrastructure to support these implementations.

For now, scaling of the benchmarks is limited to a single FPGA board with emphasis on the global memory systems and memory access patterns. However, to properly measure other important characteristics of multi-FPGA systems, additional benchmarks are required to allow the performance evaluation of parallel multi-FPGA applications and inter-FPGA communication.

4

Multi-FPGA Benchmarking and inter-FPGA Communication

In the previous chapter, we focused on the performance characterization of a single FPGA mainly based on memory access patterns of the applications. However, the scalability of the benchmarks is still limited to a single FPGA and some important benchmarks of the HPC Challenge are missing in the proposed suite. To fully exploit the performance potential of HPC systems, parallelization beyond a single node – or in our case FPGA board – is crucial. Because of that, our benchmark suite is the first going one step further and natively supports the parallel execution of all benchmarks on full multi-FPGA systems, utilizing inter-FPGA communication. The benchmarks *b_eff*, a synthetic network bandwidth benchmark, a parallel matrix transposition *PTRANS*, and *HPL* are good candidates to stress the communication interfaces by scaling the workload even further beyond a single FPGA.

For inter-FPGA communication, there does not exist a standard comparable to MPI on CPUs and multi-FPGA applications often implement custom network stacks specialized for the communication pattern and the used network infrastructure. The serial interfaces of recent FPGAs can be used to establish circuit-switched as well as packet-switched communication and various network stacks and full communication frameworks range from link-level networking IPs to highly complex frameworks with high-level APIs and collective communication support. This high flexibility in the network architecture and functionality enables optimization of the network stack and infrastructure for the specific application in the HPC domain. However, as discussed in Section 2.3, existing multi-FPGA systems often come with a fixed

infrastructure for inter-FPGA communication that is only compatible with a small set of existing communication frameworks. Limiting the usage of communication frameworks for our *base implementations*, this fact again shows the importance of the extensibility of our benchmark suite to support framework-specific optimized benchmark implementations to utilize the system-specific network infrastructure.

In this chapter, we will take a closer look into the landscape of inter-FPGA communication approaches and their current state of the art for FPGA accelerated HPC applications by further extending our existing benchmark suite and evaluating different communication approaches. The design of our benchmark suite simplifies the extension with support for new communication frameworks while taking advantage of the shared build pipeline and result validation. With that, our benchmark suite sets the foundation for a common baseline for the empirical evaluation of communication frameworks, independent of the FPGA vendor and networking infrastructure.

The contents of this chapter have been published as an article in the *ACM Transactions on Reconfigurable Technology and Systems* in 2023 [49]. All measurements results, scripts, and reports used in this chapter are published via Zenodo [50].

4.1 Definition and Implementation of Multi-FPGA Benchmarks

The existing benchmark kernels of HPCC FPGA are called *base implementations* and are designed to provide good performance on different FPGA architectures. On the one hand, this is achieved with configuration parameters that allow scaling the benchmark kernels and, on the other hand, with code optimizations that apply to a broad range of FPGAs. This allows the creation of efficient designs without manual code changes for different FPGA architectures. The configuration parameter `NUM_REPLICATIONS` is supported by all benchmarks of the suite and is used to replicate kernels to increase resource utilization. We also support this parameter in the newly added benchmarks. A more detailed description of the build process is given in Chapter 3 and in the online documentation.

Different hardware interfaces can be utilized for inter-FPGA communication with recent FPGA boards. Direct inter-FPGA communication via the serial interfaces requires vendor-specific extensions and libraries, which makes it impossible to create *base implementations* with this approach. However, sending the data over the host via PCIe and MPI can be implemented in vendor-independent OpenCL code. Thus, we use this communication approach in the base implementations.

Since we use the same structure for code organization and the build process as the existing benchmarks in *HPCC FPGA*, the new benchmarks come with support for custom kernels.

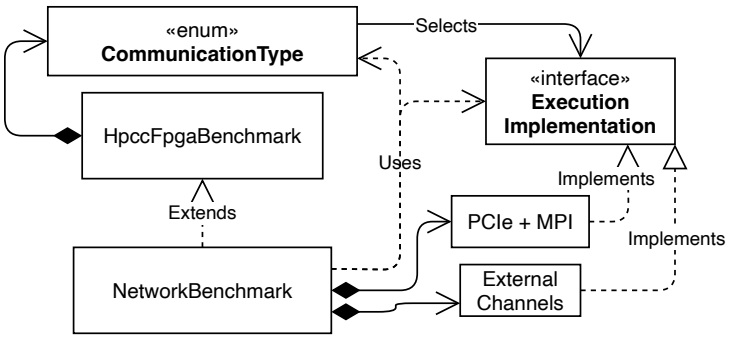


Figure 4.1: Improved architecture of the benchmark host code to increase extendability with different OpenCL kernels. `NetworkBenchmark` is the host implementation of one of the new benchmarks. It itself contains different implementations for the execution of the OpenCL kernels on the FPGA that depend on the communication scheme. To extend a benchmark for another communication scheme, only a new execution implementation needs to be added.

This allows easy extension of the benchmarks with additional OpenCL kernels. One restriction is that the OpenCL kernels need to have the same kernel signature to work with the existing host code. This means the types of input and output parameters of the kernels and their ordering need to be the same. For some communication schemes and optimized designs, it may be required to slightly change the kernel signature to pass additional information to the kernels. We extended the host code architecture as shown in Figure 4.1 to also simplify the extension of the benchmarks from the host side. The `CommunicationType` defines the different communication schemes that are supported by the benchmark suite. Each benchmark inherits from the `HpccFpgaBenchmark` class, so a substantial part of the host code is shared between all benchmark host implementations. The actual OpenCL kernel execution is done in the implementations of the `ExecutionImplementation` interface. This enables FPGA designs with different numbers of OpenCL kernels or kernel signatures. During runtime, the execution implementation is selected based on the `CommunicationType` which is determined by the name of the bitstream file. Adding support for another FPGA design with different kernels only requires an additional implementation of the execution interface as it is done for *PCIe + MPI* and *External Channels* in the class diagram. In this work, we use this feature to implement optimized versions of the benchmarks for Intel FPGAs with direct inter-FPGA communication via QSFP ports. The OpenCL extension that we use for this implementation is called IEC.

For every benchmark, there has to be one MPI rank per FPGA, so the number of MPI ranks needs to match the number of used FPGAs. Before every kernel execution on the FPGA, the hosts synchronize using an MPI barrier to reduce the measurement error. Always the slowest execution time among all FPGAs is reported for each repetition of the benchmark execution. The best repetition is used to calculate the derived performance metric of the benchmark.

In the following, we give a detailed description of the new benchmark implementations.

4.1.1 Effective Bandwidth Benchmark

In this benchmark, we use the rules of the Effective Bandwidth (b_eff) benchmark given in [51]. It is a synthetic benchmark that uses the derived metric *effective bandwidth* to combine both – the network latency and bandwidth – into a single metric. The original benchmark sends a total of 21 messages of sizes $2^0, 2^1, \ldots, 2^{20}$ byte to neighbor nodes in a ring topology. The effective bandwidth is calculated from the measured bandwidth for the different message sizes as shown in Equation 4.1.

$$b_{\mathit{eff}} = \frac{\sum_L (max_{rep}(b(L, rep)))}{21} \tag{4.1}$$

where L are the message sizes, *rep* the repetitions of the execution and $b(L, rep)$ the measured bandwith for message size L during repetition *rep*.

The base implementation of this benchmark does not require a FPGA kernel because data is transferred between the FPGAs solely by the host. The optimized version for Intel FPGAs is configurable with the parameters given in Table 4.1. Next to the number of kernel replications, it contains the width of the external channels in Bytes.

Table 4.1: Configuration parameters of the b_eff benchmark.

Parameter	Description
`CHANNEL_WIDTH`	The width of a single external channel in bytes

4.1.1.1 Base Implementation

The base implementation exchanges the messages between the global memory of neighboring FPGAs in the ring. Therefore, it reads a memory buffer representing the message using the OpenCL directive `clEnqueueReadBuffer` from FPGA to the host. In a second step, it exchanges the buffer via `MPI_Sendrecv` with the node that contains the neighboring FPGA. In the last step, the buffers are written to the global memory of the FPGAs with the `clEnqueueWriteBuffer` OpenCL directive. These steps are executed for both directions in the ring and for all message sizes.

The expected performance is limited by the required time to read ($pcie\_read_t$) and to write ($pcie\_write_t$) a message of the given size to the FPGA via PCIe, plus the time required to exchange the message between the nodes using MPI (mpi_t). All three steps need to be executed sequentially, so the expected bandwidth for a message size L can be modeled with Equation 4.2.

$$b_L = \frac{2 \cdot L}{pcie\_write_t + mpi_t + pcie\_read_t} \tag{4.2}$$

4.1.1.2 Intel External Channels Implementation

The Intel-optimized implementation requires OpenCL kernel code and consists of two different kernel types: a *send* kernel and a *receive* kernel. During execution, they continuously send or receive data over two external channels of the width specified in `CHANNEL_WIDTH`. A kernel replication always consists of both kernels since a *send* kernel always requires a counterpart. Because the message size might exceed the width of the channels, the messages are further divided into data chunks that match the channel width. Thus, a message is streamed chunk-wise over two channels to the receiver in a pipelined loop. At kernel start, the *send* kernel will generate a message chunk that is filled with bytes of the value $ld(m) \mod 256$. The message chunk will be used continuously for sending and will be stored in global memory after the last transmission. This allows verifying the correct transmission of the data chunk over the whole range of repetitions.

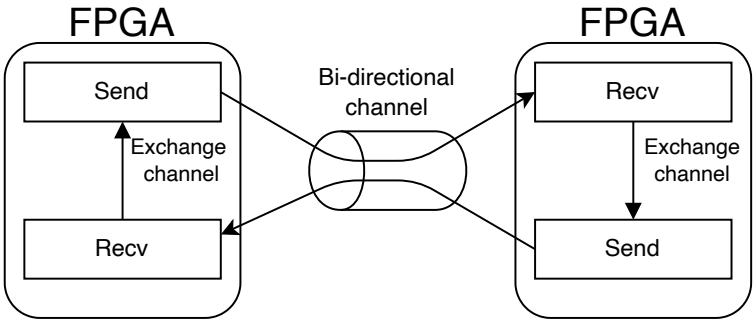


Figure 4.2: Data exchange of two kernel pairs over the external channels. The kernel pairs are executed on two different FPGAs and communicate over the bi-directional external channels. The kernel pairs are connected over an internal channel to forward the received data to the send kernel for the next iteration.

A schematic view of the channel connections for the kernel implementation is given in Figure 4.2. The kernels are connected to another kernel pair on a different FPGA. In the Figure, the kernels form a small ring over two FPGAs and the topology can be arbitrarily scaled by adding more FPGAs.

The arrows describe the path of a single data chunk through the kernels. A message chunk will be repeatedly sent over the external channel until the sum of all sent chunks matches the desired message size. Messages are sent in parallel in both directions. After a complete message is sent, the message chunk is forwarded from the receive to the send kernel over the internal channel. Only then, the next message is sent now using the message chunk received

over the internal channel. The message chunk is stored in a global memory buffer after the last message is exchanged and used for validation on the host side. A single send-receive kernel pair will use two external channels in both directions.

Table 4.2: Characteristics of the serial channel IP of the BittWare 520N board taken from the specification [52].

Parameter	**Description**	**Value**	
c_n	Number of external channels	4	
c_l	Latency of a channel	520	ns
c_f	Frequency of a channel	156.25	MHz
c_w	Width of a channel	32	B

The performance metric of the benchmark combines latency and the total bandwidth of the network. To model the performance, we need precise information about the latency and bandwidth of the external channels as they are given in Table 4.2. Every kernel replication can only make use of two external channels, which means that $c'_n = 2$ and the total number of external channels is utilized by using a replication count of 2. The execution time of a kernel pair for a given message size can then be modeled with Equation 4.3 where L is the used message size and i is the number of messages that are sent.

$$t_{L,i} = \frac{\lceil \frac{L}{c'_n \cdot c_w} \rceil \cdot i}{c_f} + i \cdot c_l \tag{4.3}$$

For the bandwidth model, we insert the values for the IP core taken from Table 4.2 which results in Equation 4.4.

$$b_L = \frac{2 \cdot L}{\lceil \frac{L}{64B} \rceil \cdot 6.4ns + 520ns} \tag{4.4}$$

This equation models the bandwidth for a single send-receive kernel pair and is expected to scale linearly with the number of kernel pairs.

4.1.2 Parallel Matrix Transposition

The parallel matrix transposition (PTRANS) benchmark computes the solution of $C = B + A^T$ where $A, B, C \in \mathbb{R}^{n \times n}$. The matrix A is transposed and added to another matrix B. The result is stored in matrix C. All matrices are divided into blocks, and the blocks are distributed over multiple FPGAs using a PQ distribution scheme shown in Figure 4.3.

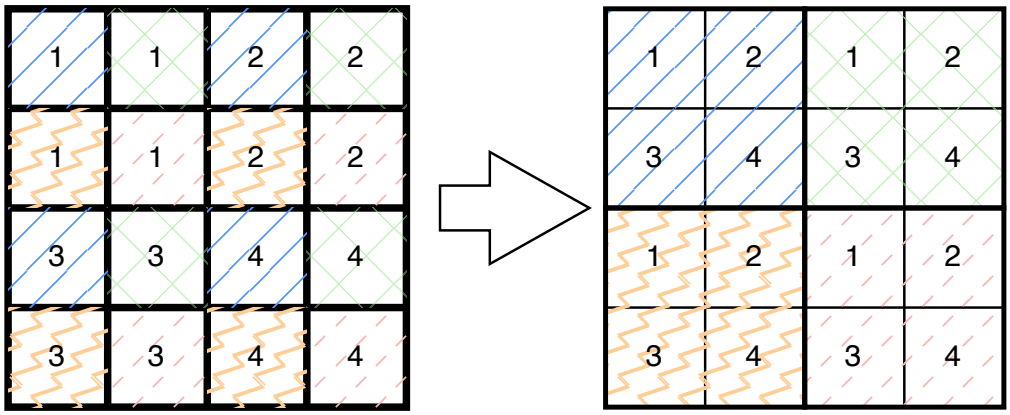


Figure 4.3: Diagonal distribution of the 16 blocks of a 4×4 block matrix on four FPGAs with $P = Q = 2$. The original matrix is shown on the left. Colors equal the FPGA in which global memory the data block of the matrix will reside at the beginning of the calculation. On the right, the placement of the data on the different FPGAs is shown. The bold lines represent the borders of the memory of a single FPGA.

Table 4.3: Configuration parameters of the PTRANS benchmark.

Parameter	Description
`BLOCK_SIZE`	Size of the matrix blocks that are buffered in local memory and also distributed between the FPGAs.
`CHANNEL_WIDTH`	Width of the channels in data items. Together with the used data type, the width of the channel in bytes can be calculated.
`DATA_TYPE`	Specifies the used data type for the calculation.

4.1.2.1 Base Implementation

The configuration parameters for the implementation are given in Table 4.3. The number of kernel pairs can be defined with the `NUM_REPLICATIONS` parameter and the block size that also defines the sizes of the local memory buffers can be set with `BLOCK_SIZE`. Moreover, the width of the channel is defined by `CHANNEL_WIDTH` and `DATA_TYPE`. It should match the width of the used communication channel. In the case of the base implementation, this is the width of the global memory interface.

The implementation consists of a single OpenCL kernel that sequentially executes three pipelines for every matrix block. In the first pipeline, a block of matrix A is read from global memory and written into a buffer. The second pipeline reads the block of A transposed from the buffer, reads a block of B from global memory, adds both blocks, and stores the result in an additional buffer. The content of this buffer is written back to global memory in the last pipeline.

Every pipeline is reading or writing a single block of data from the global memory. This is a similar approach as it is used for the STREAM benchmark in the suite and leads to efficient use of the global memory bandwidth if the design is replicated for every memory bank. Before the kernel can be executed, the matrix A needs to be exchanged by the host ranks using `MPI_Sendrecv`. After this operation, the matrix is blockwise transposed so the FPGA only has to transpose the individual blocks.

With Equation 4.5, the expected execution time for a single matrix block is given. It consists of the time required to exchange the blocks via MPI and write them into the global memory of the FPGAs (t_{MPI}) and the execution time of the OpenCL kernel. The kernel execution time is based on the three pipelines that are executed sequentially, the block size b, the channel width in number of values c_w, and the clock frequency of the used channel c_f. Depending on the number of kernel replications, the FPGA may be able to process multiple matrix blocks simultaneously without interference.

$$t_{PTRANS} = t_{MPI} + 3 \cdot \frac{b^2}{c_w} \cdot c_f \tag{4.5}$$

For the verification of the data, the non-transposed blocks of matrix A are exchanged by the hosts using MPI. Then, each host re-calculates the result using a CPU reference implementation. The reported error is the maximum residual error between the FPGA and CPU result.

4.1.2.2 Intel External Channels Implementation

The Intel-specific implementation comes with the restriction that $P \stackrel{!}{=} Q$. This allows setting up a static circuit-switched network between the pairs of FPGAs and exchanging the matrix blocks without additional routing. The FPGA logic is implemented in two kernels per external channel, similar to the b_eff benchmark. For this implementation, the width of the channel defined by `CHANNEL_WIDTH` and `DATA_TYPE` should match the width of the external channels.

One of the kernels reads a block of A into local memory. The size of this local memory buffer can be defined with `BLOCK_SIZE`. The block of matrix A is then read transposed from local memory and written into the external channel. Reading from global memory and writing to the external channel is implemented in a single pipeline, double buffering the local memory block.

The second kernel will receive chunks of a transposed block of A, add a block of B to it and store it in global memory. In consequence, no local memory is needed in this kernel. One major goal of this implementation is to continuously send and receive data over all available external channels to utilize the available network bandwidth, which is most likely the performance bottleneck. Nevertheless, the kernels may also suffer from low global memory

bandwidth because they need to concurrently read and write to three different buffers for every kernel replication.

This leads to a total required global memory bandwidth on a single FPGA given in Equation 4.6.

$$b_{global} = 3 \cdot r \cdot c_w \cdot c_f \tag{4.6}$$

where r is the number of external channels per FPGA (or the number of kernel replications), and c_f and c_w the frequency and the width of an external channel as defined in Table 4.2. This means the required global memory bandwidth is three times higher than the network bandwidth to keep the benchmark network bandwidth-bound. As a performance metric, the Floating Point Operations (FLOPs) per second are calculated. For the calculation it is assumed, that n^2 additions are required for the computation on matrices of width n. Considering the characteristics of the external channels of the used BittWare 520N boards, the maximum performance will be $p = i \cdot r \cdot 32B \cdot 156.25MHz$ for a sufficiently large matrix, where i is the number of used FPGAs. Note, that the block size is not considered in this performance model. It is used to allow larger memory bursts from global memory, which are defined by the width of the block. This will lead to a higher efficiency of the global memory accesses, but since the performance model covers the case where the network bandwidth is the bottleneck, this parameter can be neglected. However, for very small block sizes, the efficiency of the global memory may be reduced to the point that it becomes the bottleneck.

4.1.3 High Performance LINPACK

The HPL benchmark solves a large equation system $A \cdot x = b$ for x, where $A \in \mathbb{R}^{n \times n}$ and $b, x \in \mathbb{R}^n$. This is done in two steps: First, the matrix A is decomposed into a lower matrix L and an upper matrix U. In the second step, these matrices are used to first solve $L \cdot y = b$ and finally $U \cdot x = y$ to get the result for the vector x. For the implementation of the benchmark on FPGA, the rule set for the HPL-AI mixed-precision benchmark [53] was adopted, which defines A to be a diagonally dominant matrix. Thus, the LU factorization does not require pivoting. In contrast to the original benchmark, it is possible to choose between single-precision and double-precision floating-point values. Since the benchmark suite is designed to only measure the FPGA performance, no additional iterative method is used to refine the result if a lower precision is used. Only the LU decomposition, which is the most compute-intensive step in this calculation, is executed on the FPGAs. The number of FLOPs for this step is defined to be $\frac{2 \cdot n^3}{3}$ for a matrix A with width n in contrast to $2 \cdot n^2$ for solving the equation systems for the LU-decomposed matrix. Only the performance of the LU factorization on the FPGA is reported.

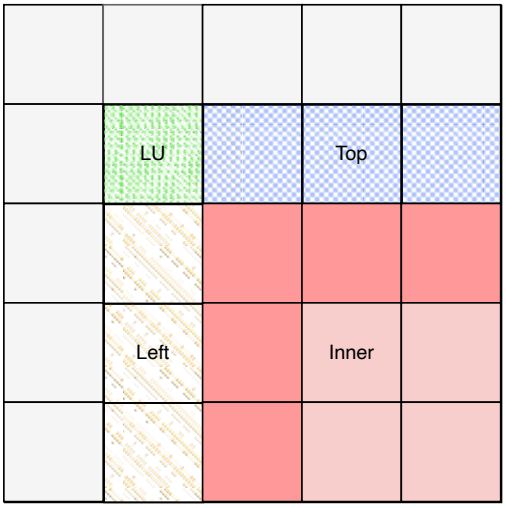


Figure 4.4: In every iteration of the algorithm, a single block in the matrix is decomposed into a lower and upper matrix (green). The lower matrix is used to update all blocks on the right of this block (blue) and the upper matrix to update all blocks below this block (orange). The updated Top and Left blocks are then used to update all inner blocks (red) and the dark red blocks need to be updated before the next communication phase can start.

4.1.3.1 Base Implementation

The base implementation uses a blocked, right-looking variant for the LU factorization as described in [54]. Therefore, the matrix will be divided into sub-blocks with a width of $2^{BLOCK\_SIZE\_LOG}$ elements. The exact size of the blocks is defined over a configuration parameter. For the update of a single row and column of blocks, we need to perform four different operations. A single iteration of the blocked LU decomposition is shown in Figure 4.4. In every iteration, the LU factorization for a diagonal block of the matrix is calculated which is marked green in the visualization. All grey-colored blocks on the left and top of this block are already updated in previous iterations and will require no further processing. This is why this approach is called right-looking since we will always update the blocks on the right of the LU block. After the LU block is decomposed, the lower matrix block L is used to update all blocks on the right of the LU block. Since they are the top-most blocks that still require an update, they are in the following called *top blocks*. The upper matrix block U is used to update all blocks below the current LU block. These are the left-most blocks that require an update, so they are referred to as *left blocks*. The left and top blocks again are used to update all inner blocks, which can efficiently be done using matrix multiplication. The design contains a separate kernel for each of the four operations.

Additionally, a single iteration of the LU decomposition is split into two subsequent steps in the design. In the *communication phase*, the LU, left, and top blocks are updated which

also involves data exchange between kernels on the same FPGA and between the FPGAs. In the *update phase*, the exchanged data is used to update all inner blocks locally using matrix multiplication kernels.

Both phases can overlap as shown by the timeline of kernel executions in Figure 4.5 based on the matrix given in Figure 4.4. The number of matrix multiplications required for a single iteration of the algorithm increases quadratically with the matrix size. No data dependency exists between the light red matrix multiplications and the operations of the next *communication phase*, which allows the overlapping of the two phases. For large matrices, this means that the performance of the implementation is limited by the aggregated performance of the matrix multiplication kernels. During the *communication phase*, matrix blocks are exchanged via the host using PCIe and MPI.

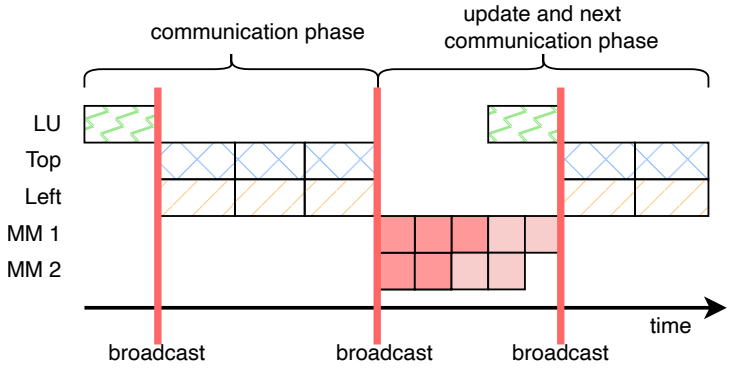


Figure 4.5: Kernel executions over time for a single iteration of the LU decomposition in the base implementation. During the communication phase, data needs to be exchanged two times between FPGAs using MPI and PCIe. The matrix multiplication kernels are executed in the update phase. Communication and update phase of subsequent iterations can overlap, so communication latency can partially be hidden.

4.1.3.2 Intel External Channels Implementation

Figure 4.6 shows connections between the kernels used in the communication phase. For the execution over multiple FPGAs, the boards are arranged into a quadratic 2D torus of variable size using point-to-point connections. Not all kernels need to be active on every FPGA within a single iteration. Instead, data can also be received over the external channels if it is computed on another FPGA. If the FPGA is in charge of calculating the LU block, the LU kernel is executed and the decomposed L and U blocks are forwarded row and column-wise to a network kernel. The network kernel forwards the data over the external channels to neighboring FPGAs in the torus. The four possible directions are used for different types of data as it is indicated by the red arrows. Moreover, the network kernel forwards the locally

computed L and U blocks to the Left and Top kernel. The top and left kernels use the data to update a block with the L or U block and forward the updated block to the next network kernels. Here, the input data is selected either from the internal or external channels and data is forwarded over the external channels if required. Besides that, incoming data is stored in global memory buffers for later use in the update phase. By splitting the network communication into three kernels, it is possible to establish a cycle-free data path through the torus during the communication phase. This reduces the impact of pipeline and channel latencies during this phase.

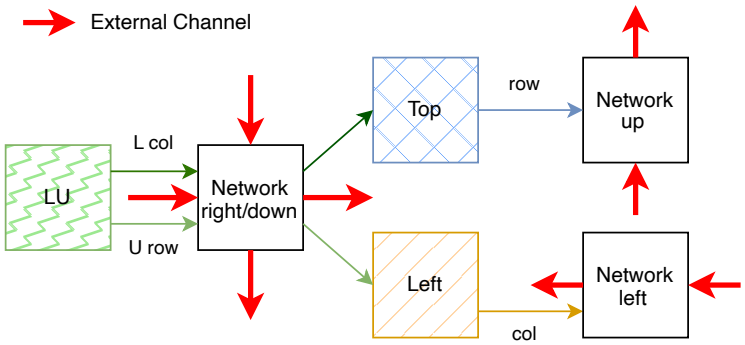


Figure 4.6: Data flow through the kernels of the communication phase on a single FPGA. The kernels are connected over internal channels. Between the calculation kernels, network kernels are used to select the correct input for the next kernel from the internal or external channels. The network kernels for the top and left direction will also store incoming matrix blocks as input for the matrix multiplication. The bold arrows represent the serial channels and their direction in the 2D torus.

During the transfer from the left kernel to the network kernel, the left blocks are transposed. This allows a simplified design of the matrix multiplication for the inner blocks since all input matrices can be processed row-wise. Figure 4.7 shows a part of the execution of the kernels over time for the iteration given in Figure 4.4. It can be seen that the LU kernel is only executed once per communication phase. The lower and upper matrices are buffered by the left and top kernels to allow the update of subsequent blocks. All network kernels are summarized under *Network* in the graph. In the example, two matrix multiplication kernels are used, and the blocks a redistributed between the two replications. The next communication phase starts as soon as the first row and column of the inner blocks are updated, which is represented by the dark red blocks.

A 2D torus is used to connect multiple FPGAs for the LU decomposition. Every FPGA is programmed with the same bitstream and the host schedules the kernels in the required order and configuration. Matrix blocks are distributed between the FPGAs using a PQ-grid of the size of the torus. This allows balancing the load between the devices more evenly since the

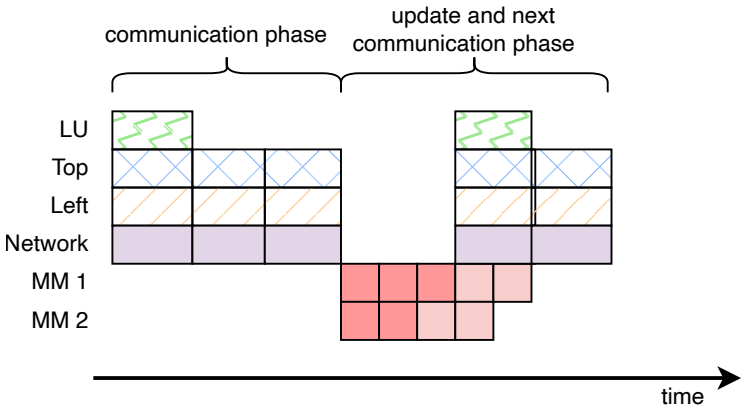


Figure 4.7: Kernel executions over time for a single iteration of the LU decomposition. During the communication phase, the network kernels are active whereas during the update phase the matrix multiplication kernel is executed. Communication and update phase of subsequent iterations overlap.

matrix will get smaller with every iteration of the algorithm. In Figure 4.8 the active kernels and the data exchange between FPGAs in a 3×3 torus is shown for a global matrix size of more than 12 blocks so the FPGA has to update more than four blocks. In this case, only the FPGA on the top left needs to execute all four compute kernels, but every FPGA will use its matrix multiplication kernel.

The base implementation of the HPL benchmark uses a similar two-leveled blocked approach to the GEMM benchmark described in Chapter 3. Thus, it uses two parameters to specify the block sizes of the local memory buffers and of the compute units as described in Table 4.4. Additionally, it is possible to specify the data type and specify the number of matrix multiplication kernels using the `NUM_REPLICATIONS` parameter. The two-leveled blocked approach is also used for the communication kernels and all kernels use the same first-level block sizes. A main difference is the parallelism in the computation between communication kernels and matrix multiplication. The matrix multiplication kernels unroll the computation in three dimensions in the second level whereas the communication only uses a two-dimensional unrolling. This means the parallel calculation increases cubically with the chosen register block size for the matrix multiplications and only quadratically for the communication kernels. It would also be possible to scale the second-level blocks differently between the two kernel types. This has to be considered in further optimization steps.

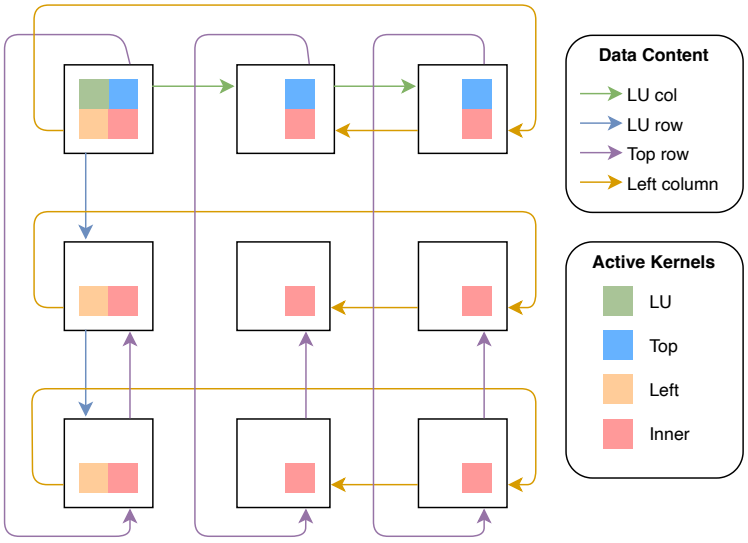


Figure 4.8: For the LU decomposition, the FPGAs use a 2D torus network topology to exchange data. This example shows the active kernels during a single iteration in a 3×3 torus. The black boxes are the FPGAs, and the colors within the boxes indicate the active kernels. The direction and type of data that is forwarded between the FPGAs is given by the arrows. In every iteration of the algorithm this communication scheme shifts one FPGA to the bottom-right in the torus.

Table 4.4: Configuration parameters of the HPL benchmark.

Parameter	Description
`LOCAL_MEM_BLOCK_LOG`	Logarithm of the size of the matrix blocks that are buffered in local memory and also distributed between the FPGAs.
`REGISTER_BLOCK_LOG`	Logarithm of the size of the second level matrix blocks. The kernels contain completely unrolled logic to start the computation of such a sub-block every clock cycle.
`DATA_TYPE`	Specifies the used data type for the calculation.

Only the LU factorization of matrix A is calculated on the FPGAs. After this step, the equation system is solved using a distributed CPU reference implementation among all MPI ranks. The input matrix was generated such that the resulting vector is a vector of all ones. The reported error is the normalized maximum residual error calculated with $\frac{||x||}{n \cdot ||b|| \cdot \epsilon}$ where n is the width of matrix A and ϵ the machine epsilon.

4.1.4 Extend Existing Benchmarks for Multi-FPGA Execution

In addition to the new benchmarks proposed in this chapter, we also extend the existing benchmarks of Chapter 3 for the execution in a multi-FPGA environment. An essential configuration parameter for all benchmarks is the specification of kernel replications `NUM_REPLICATIONS`. These kernel replications are kernels with the same or very similar functionality to allow higher utilization of the FPGA resources. The input data is then distributed between the replicated kernels such that every kernel works on its own data set. Especially for devices with HBM2 this step is crucial to make use of the high number of memory banks and the high aggregated bandwidth. As a result, the existing implementations in HPCC FPGA already handle the memory banks on a single FPGA like a distributed memory system. and changes on the OpenCL kernels are not required for most of the legacy benchmarks. The extensions focus on the host codes to trigger the distributed execution and support the validation and result collection over multiple compute nodes using MPI.

The RandomAccess benchmark was not well scalable because it could at best update a single data item per clock cycle even when scaled over multiple FPGAs. This is limited by the way the pseudo-random numbers for the address calculation are generated. We now allow the generation of multiple pseudo-random numbers per clock cycle to overcome this limitation by replicating the Random Number Generator (RNG). This also changed the configuration parameters of the benchmark as given in Table 4.5.

A single replication of the improved RandomAccess kernel is given in Figure 4.9. The RNGs are initialized with different seeds to generate a sub-part of the random-number sequence. In consequence, the same random number as with the old version is generated, only the order of

updates may vary. Every clock cycle, the RNG outputs a new random number. This number is placed into a shift register if two conditions hold:

- The buffer address derived from the random number is in range of the kernel replication.
- The shift register does not already contain a valid random number at the position where it should be inserted.

In the latter case, the RNG will stall until the random number can be placed into the shift register. So in other terms, the produced random numbers are sequentialized by the shift register for the input into the actual update logic. This approach increases the probability, that the update logic processes a valid address for high numbers of kernel replications. Since scaling over multiple FPGAs corresponds to increasing the number of kernel replications, this also improves the performance in multi-FPGA execution.

Table 4.5: Configuration parameters of the RandomAccess benchmark.

Parameter	**Description**
`HPCC_FPGA_RA_DEVICE_BUFFER_SIZE_LOG`	Logarithm of the size of the data buffer that is randomly updated in number of values.
`HPCC_FPGA_RA_RNG_COUNT_LOG`	Logarithm of the number of RNGs that are created per kernel replication.
`HPCC_FPGA_RA_RNG_DISTANCE`	Distance between RNGs in the shift register.

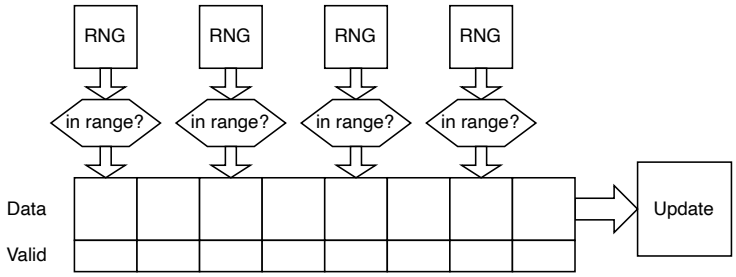


Figure 4.9: RandomAccess shift register used to connect the RNGs to the update logic. The RNG will only put the generated number in the shift register if no valid number is at the current position.

4.2 Benchmark Execution and Evaluation

In the following, we execute and evaluate the scaling behavior of the existing and the three new benchmarks of the benchmark suite on two different multi-FPGA systems containing Xilinx or Intel FPGAs.

4.2.1 Evaluation Setup and Synthesis Results

For the evaluation of the benchmarks we used version 0.5.1 of the benchmark suite [2] and we made all artifacts and code modifications for additional experiments publicly available [50].

We synthesized and executed the benchmarks on two multi-FPGA systems: The Noctua 1 system of PC² at Paderborn University and the Xilinx FPGA evaluation system of the Systems Group at ETH Zurich. The systems are described in more detail in Section 2.3. For the benchmark execution, the data is generated on the CPUs and moved to the DDR memory on the FPGA boards. Results are copied back to the CPU memory from the DDR memory for validation after the benchmark execution.

All benchmark kernels are designed to be independent of the number of used FPGAs, so only a single synthesis for every benchmark and FPGA board is required for the evaluation. With configuration parameters, it is possible to improve resource utilization and performance of the benchmark kernels for a specific FPGA board before synthesis. For the benchmarks STREAM, FFT, and GEMM, these configuration parameters were discussed in more detail in Chapter 3. Table 4.6 contains the used configurations for each benchmark. The configuration parameters are chosen to better utilize performance-relevant resources on the FPGA.

In Table 4.7 the resource usage of the synthesized benchmark kernels is given. Our updated scalable implementation of the RandomAccess benchmark now requires additional logic and BRAMs to implement the RNGs. We have chosen the number of RNGs in the configuration to be the next larger power of two of the used FPGAs. This increases the probability, that a valid number can be processed in every clock cycle by every FPGA. Further increasing the number of RNGs may lead to lower clock frequencies, which will also have an impact on performance. With the recent SDK version we were able to synthesize the GEMM benchmark with a much higher clock frequency for the BittWare 520N, which promises a large performance improvement. For the b_eff benchmark, only a single bitstream is synthesized. For the base implementations, no bitstream is required since the data transfer is solely handled by the host. Also resource consumption is not an issue with this synthetic benchmark because the performance is mainly limited by the network bandwidth and latency.

PTRANS requires BRAM buffers that are used to block-wise transpose the matrix and store intermediate results. A large block size will benefit large memory bursts but it is also important to achieve a clock frequency close to 300 MHz to make best use of the memory bandwith. For HPL it is – similar to GEMM – important to maximize the number of used DSPs for matrix multiplications. In addition, mainly some extra BRAM is required to store the matrix blocks for the kernels of the communication phase. As a result, the resource utilization is very similar to GEMM. A significant difference is visible for the Alveo U280, where only 69% of the DSPs can be utilized. We were not able to fit the communication kernels and three matrix multiplication kernels on the FPGA because this would overutilize the DSPs. One approach to make use of the remaining DSPs would be to reduce the parallelism for the third

matrix multiplication kernel by setting `REGISTER_BLOCK_LOG` to 2 only for this replication. However, the baseline version of the benchmark does not support this approach since the kernel scheduling would need to account for slow and fast matrix multiplication kernels.

Table 4.6: Synthesis configurations of all benchmarks for multi-FPGA execution.

Benchmark	**Parameter**	**520N IEC**	**520N PCIe**	**U280 PCIe**
STREAM	NUM_REPLICATIONS	4	4	2
	DATA_TYPE	float	float	float
	GLOBAL_MEM_UNROLL	1	1	1
	VECTOR_COUNT	16	16	16
	DEVICE_BUFFER_SIZE	32,768	32,768	16,384
RandomAccess	NUM_REPLICATIONS	4	4	2
	RA_DEVICE_BUFFER_SIZE_LOG	0	0	10
	HPCC_FPGA_RA_RNG_COUNT_LOG	5	5	3
	HPCC_FPGA_RA_RNG_DISTANCE	5	5	1
FFT	NUM_REPLICATIONS	2	2	1
	LOG_FFT_SIZE	17	17	9
GEMM	NUM_REPLICATIONS	5	5	3
	DATA_TYPE	float	float	float
	GLOBAL_MEM_UNROLL	8	8	8
	BLOCK_SIZE	512	512	256
	GEMM_SIZE	8	8	8
b_eff	NUM_REPLICATIONS	2	only host code required	
	CHANNEL_WIDTH	8	only host code required	
PTRANS	NUM_REPLICATIONS	4	4	2
	DATA_TYPE	float	float	float
	CHANNEL_WIDTH	8	16	16
	BLOCK_SIZE	512	512	256
HPL	NUM_REPLICATIONS	5	5	2
	DATA_TYPE	float	float	float
	LOCAL_MEM_BLOCK_LOG	9	9	8
	REGISTER_BLOCK_LOG	3	3	3

The difference in DSPs between the base and optimized implementation for the BittWare 520N is caused by the way, that the multiplication of the 8×8 matrices in registers is implemented. In case of the IEC version, only multiply-adds are used consuming 512 DSPs in total per replication. In the base implementation, the compiler created the same matrix multiplication from 64 dot-products of size 8 followed by 64 additions. This slightly increases the DSP usage

to 576 DSPs per replication but considerably reduces the logic and BRAM usage as it can be seen in the resource utilization.

Besides the two bitstreams for the baseline and one for the vendor-specific implementation, we also synthesized HPL with a block size of 256 elements for the 520N. This is the same block size that is used on the U280 and allows a better comparison of the performance results of both FPGA boards. The configuration requires considerably lesser logic and BRAM compared to the version wih 512 element block width and achieves a higher clock frequency.

Table 4.7: Resource utilization of all synthesized benchmark kernels.

Benchmark	**Board**	**Logic**	**BRAM**	**DSPs**	**Frequency [MHz]**	**Comp. Time [h]**
STREAM	520N	178,268 (19%)	3,926 (33%)	128 (2%)	341.67	2.64
	U280	188,124 (14%)	854 (42%)	170 (2%)	300.00	2.44
RandomAccess	520N	222,405 (24%)	602 (5%)	14 (<1%)	325.00	3.50
	U280	184,888 (14%)	350 (17%)	24 (<1%)	300.00	2.20
FFT	520N	280,105 (30%)	1,811 (15%)	1,560 (27%)	400.00	4.10
	U280	375,069 (29%)	342 (17%)	682 (8%)	286.00	6.94
GEMM	520N	310,564 (33%)	8,321 (71%)	3,318 (58%)	272.50	9.87
	U280	659,200 (51%)	1,139 (57%)	7,714 (85%)	186.00	12.25
b_eff	520N[2]	173,010 (19%)	512 (4%)	0 (0%)	290.63	2.45
	520N[1]	only host code required				
	U280[1]	only host code required				
PTRANS	520N[2]	242,232 (26%)	4,756 (41%)	68 (1%)	281.25	3.64
	520N[1]	233,317 (25%)	4,662 (40%)	162 (3%)	380.00	3.62
	U280[1]	283,028 (22%)	598 (30%)	96 (1%)	283.00	4.07
HPL	520N[2]	361,377 (39%)	8,326 (71%)	2,809 (49%)	244.00	10.40

Table 3.5: Resource utilization of all synthesized benchmark kernels. *(continued)*

520N[1]	303,471 (33%)	8,245 (70%)	3,185 (55%)	233.34	9.42
520N[1,3]	276,546 (30%)	2,587 (22%)	3,185 (55%)	280.00	6.08
U280[1]	663,418 (51%)	994 (49%)	6,201 (69%)	156.00	12.10

[1]communication via MPI and PCIe using the host network
[2]communication via Intel external channels (IEC) OpenCL extension
[3]version with a reduced block size of 256 elements

4.2.2 Evaluation of the Effective Bandwidth and PTRANS

The *b_eff* benchmark does not only report the derived metric *effective bandwidth* but also the achieved bandwidth for all tested message sizes, which range from a single byte to 1 MiB. A new message is only sent after the current message is received by the neighbor node.

The base implementation of the b_eff benchmark reads the data from the FPGA board to the host using an OpenCL call, exchanges the data between the host CPUs using the `MPI_Sendrecv` method, and writes the data back to the FPGA using OpenCL. This means, in contrast to the optimized IEC implementation, no OpenCL device code is required for this benchmark to work.

The measured total bandwidth over the message sizes is given in Figure 4.10 for two FPGAs or CPU nodes respectively. We do not show the MPI-only performance for the Xilinx system in this plot since it heavily overlaps with the measurements for the base implementations. The maximum theoretical bandwidths for PCIe, MPI via the Intel Omni Path 100Gbps interconnect, and IEC are given by the black dashed lines in the plot. For the base implementations, the bandwidth remains below 5 GB/s on both devices although the theoretical bandwidth of PCIe and MPI are both much higher. The bandwidth gets limited by the additional copy operations required to get the data from FPGA to CPU and back. Using Equation 4.2 and the peak MPI and PCIe bandwidths, the theoretical bandwidth of the baseline version is given in Equation 4.7:

$$b_{max} = \frac{2}{\frac{1}{12.5\text{GB/s}} + \frac{2}{8\text{GB/s}}} = 6.06\text{GB/s} \tag{4.7}$$

However, the measurements with the MPI-only implementation of the benchmark show, that the maximum message size of 1 MiB is not sufficiently large to utilize the MPI peak performance on Noctua.

The optimized IEC approach shows maximum bandwidths close to the theoretical peak for 1 MiB message sizes. Also, the measurements closely correlate to the model described in Section 4.1.1.2 in Equation 4.4.

The linear scaling behavior for the derived *effective bandwidth* metric is visualized for all implementations in Figure 4.11. All implementations show nearly perfect scaling with the available network bandwidth as it is indicated by the extrapolation lines for each device. Especially for the MPI-only and the MPI + PCIe version, the performance on a single node is considerably higher than the available network bandwidth because in these cases the data will be transferred between the ranks using shared memory. We also observe a huge difference in the effective bandwidth between the two MPI versions of Noctua and the Xilinx system. These measurements have to be kept in mind when comparing the results of the base implementation on the two systems, since the huge difference in MPI performance will also have an impact on the PCIe + MPI performance. The ability to add additional optimized implementations of the benchmark kernels allows the generation of comparable results not only limited to FPGAs but also for CPU or other accelerators. This only requires minor changes in the existing code base and large amounts of the code can be reused including handling of input parameters, input data generation and validation, calculation of derived metrics, and printing of performance summaries.

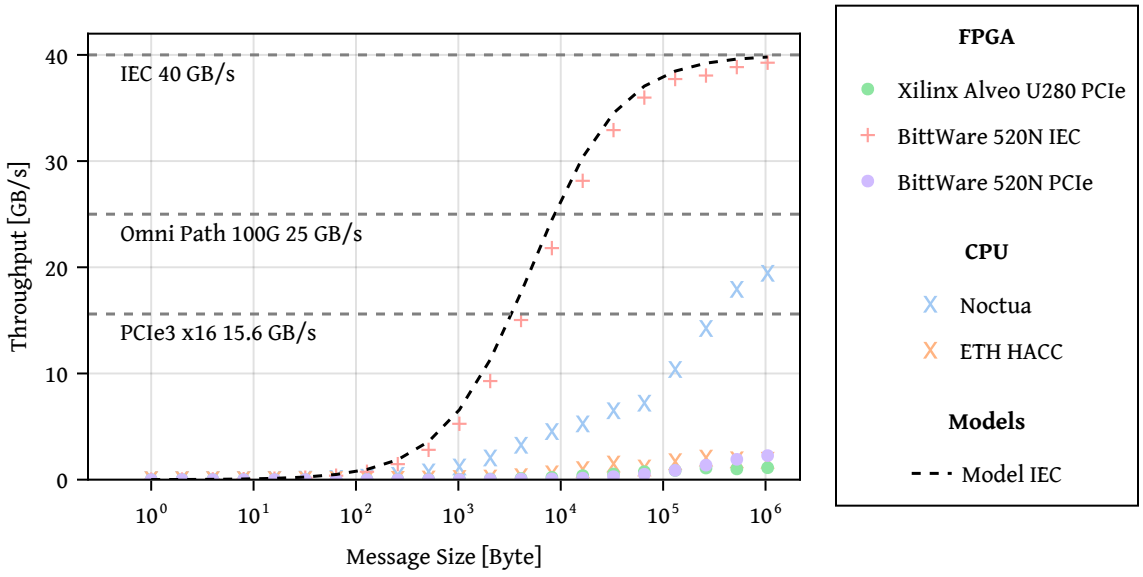


Figure 4.10: The aggregated bandwidth over different message sizes measured by the b_eff benchmark over two CPUs or FPGAs. Next to the measurements, the maximum performance for the communication between FPGAs, CPUs, and between FPGAs and CPUs via PCIe.

For the matrix transposition, the blocks of a matrix are distributed among the FPGAs in a PQ

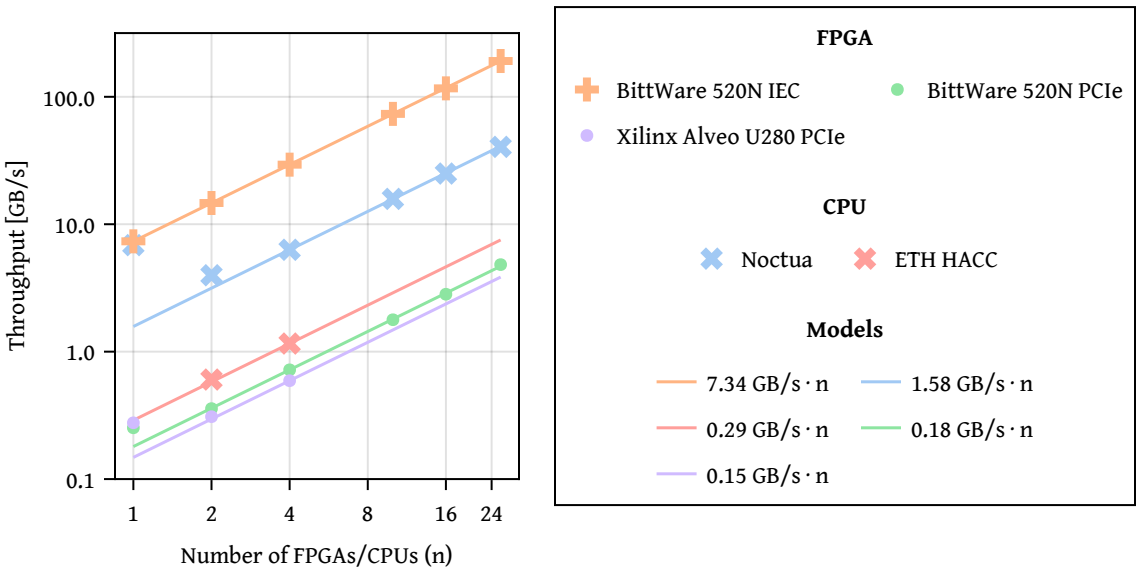


Figure 4.11: The measured effective bandwidth over the number of used FPGAs and CPUs. A logarithmic scale is used for the ring size and the measured bandwidth. The colored lines represent the perfect scaling based on measured effective bandwidth over two FPGAs or CPU nodes.

distribution where $P = Q$. A matrix of 32,768 elements is transposed using strong and weak scaling. The resulting speedups for the different FPGAs are given in Figure 4.12. In the strong scaling experiment, the base implementation on the BittWare 520N shows a better scaling behavior than the optimized version using IEC. This is because the base implementation is mainly bottlenecked by the PCIe bandwidth for the exchange of the matrices. In contrast to that, the optimized version shows a significant reduction of the speedup for a larger number of FPGAs. This is caused by the compute pipeline on the FPGAs which can not be fully utilized with the smaller matrix sizes.

In the weak scaling experiment, the matrix size per FPGA stays the same and the implementation achieves optimal speedup for up to 25 FPGAs. The base implementation shows no significant differences for both FPGAs in strong and weak scaling. On the Xilinx Alveo U280, the base implementation does not scale well. This is related to the low FPGA-to-FPGA bandwidth that we also measured in the b_eff benchmark. This means that this difference is caused by the comparably low MPI performance on the Xilinx system and not by the FPGAs.

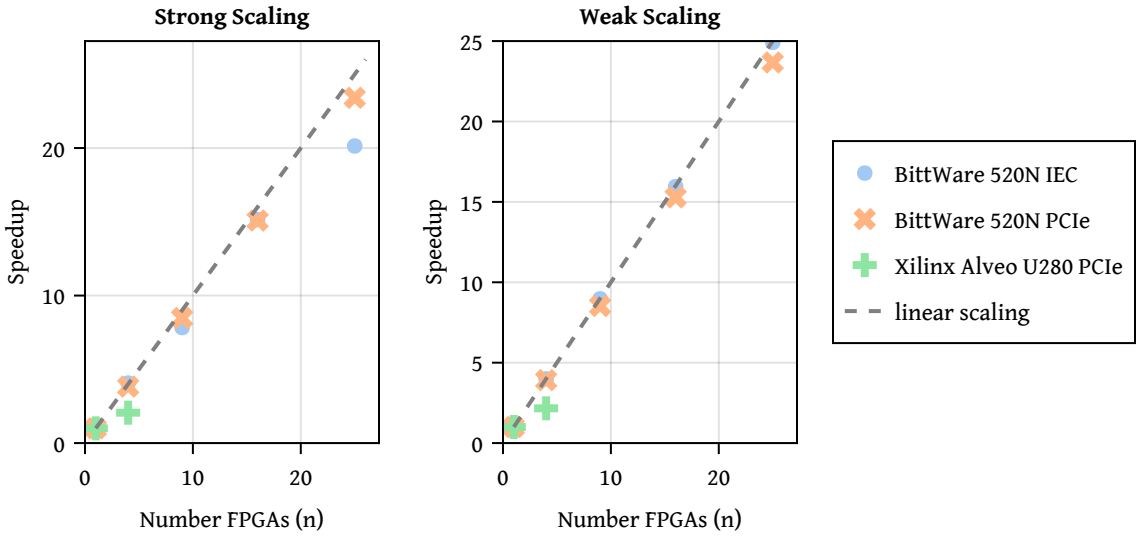


Figure 4.12: Speedup of the PTRANS benchmark executed with a quadratic matrix of 16,384 elements over up to 25 FPGAs in a weak and strong scaling scenario.

4.2.3 Evaluation of HPL

In the first experiment, we measure the performance on a single FPGA for different matrix sizes of up to 20,480 elements. The performance for four bitstreams on the two different systems is given in Figure 4.13. To allow an easier comparison of the efficiency of the design on the different platforms, the performance was normalized to a kernel frequency of 100 MHz and a single kernel replication. So the normalized performance for a given matrix size should be very similar on the different platforms. For small matrix sizes, the communication phase can not be overlapped with the computation phase. Only for larger sizes of the matrix, both phases overlap for most of the computation time and the performance converges to the matrix multiplication performance. Still, significant differences between the different bitstreams can be observed and are mainly caused by the chosen benchmark configuration parameters and compiler flags. When comparing the base version and the vendor-specific version with Intel external channels (IEC) used on the BittWare 520N board, the base version using PCIe for communication shows lower performance, although the same configuration parameters are used. The base version of the benchmark failed to synthesize with memory interleaving because additional Load Store Units (LSUs) are used for the communication and increase the complexity of the memory system. Since only a single buffer is used to store matrix A, this effectively reduces the global memory bandwidth and stalls of the matrix multiplication kernels increase.

On the Xilinx Alveo U280 board, the largest block size that fits on the device is 256 elements

in contrast to 512 element blocks for the 520N board. The reduced block size results in an overlap of communication and computation for smaller matrix sizes but also reduces the peak performance because the utilization of the matrix multiplication pipeline decreases. For comparison, we synthesized a bitstream for the 520N with a block size of 256. It shows a similar scaling behavior with regards to the matrix size but shows a slightly lower normalized peak performance. The bitstream for the 520N achieves a nearly 80% higher frequency which also increases the memory bandwidth utilization and leads to more frequent pipeline stalls, which eventually leads to a lower normalized performance.

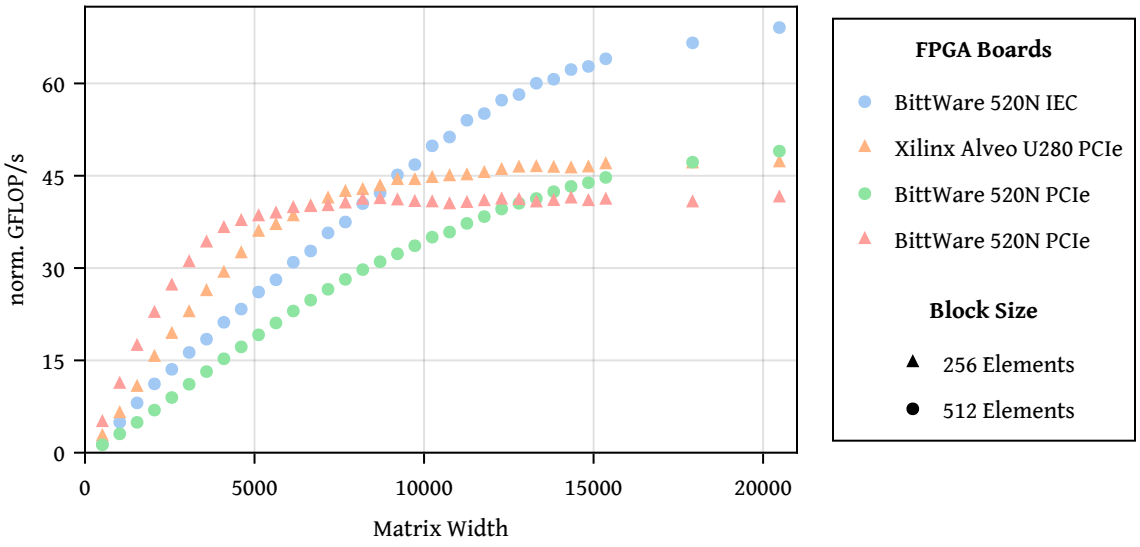


Figure 4.13: Normalized performance of the HPL bitstreams on the target FPGAs for different matrix sizes. The base versions of the benchmark are marked with PCIe, referring to the path of communication. For small matrices, the design is limited by the communication latency until it can be efficiently hidden by matrix multiplications. Moreover, an additional execution for the BittWare 520N with a block size of 256 is given for comparison with the Xilinx Alveo U280.

Based on the measurements done with a single FPGA, we set the matrix size for the multi-FPGA experiments to 24,576 elements, since all bitstreams will be close to their peak performance for this size. We use this matrix size as a base for a weak scaling experiment, where the matrix size increases with the width of the FPGA torus so the matrix size on a single FPGA remains constant. Additionally, we execute a strong scaling experiment, where the global matrix size remains constant while increasing the torus size. The measurement results for the weak scaling experiment are given in Figure 4.14. It can be observed, that all three implementations of the benchmark show a close to optimal scaling for up to 25 FPGAs. Considering the differences in the network bandwidth that affected the PTRANS results, this

also means, that the benchmark implementation is compute-bound on all FPGAs.

The results of the strong scaling experiment are given in Figure 4.15. Both benchmarks show a much lower increase in performance for larger torus sizes. Based on the data of our single FPGA scaling experiment shown in Figure 4.13, we created an extrapolation model for the strong scaling experiment. It shows that performance per FPGA is tightly coupled to the size of the local matrices on the FPGAs. The extrapolation for the Xilinx Alveo U280 shows a better speedup in this strong scaling scenario because of the smaller block sizes. With this very simple approach, it is already possible to model the performance depending on the total matrix size and the number of FPGAs with high accuracy. The strong scaling experiment shows that the overall performance in the torus is tightly coupled to the input size on a single device for all implementations.

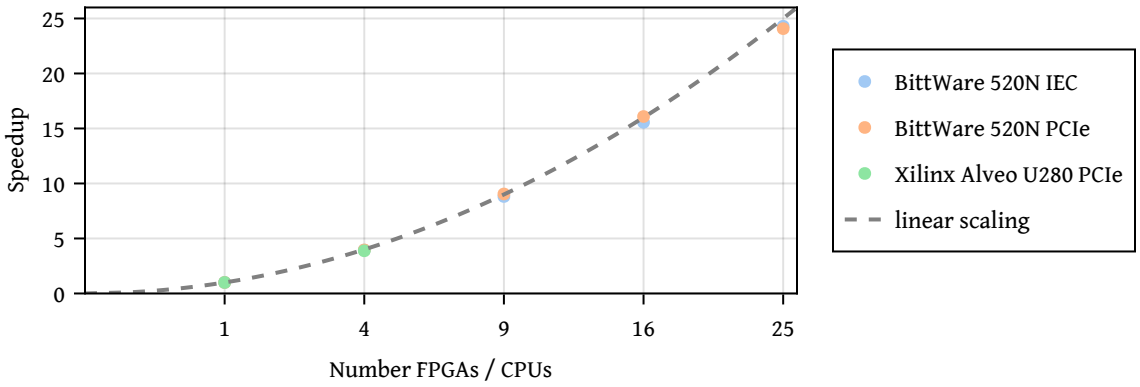


Figure 4.14: Speedup of HPL with a matrix width of 24,576 elements over multiple FPGAs in a weak scaling scenario.

The HPL implementation achieves a lower performance per FPGA than the existing GEMM benchmark, although both get their performance from matrix multiplication. Also, the configuration parameters are chosen similarly for both benchmarks, which results in a similar expected performance. The main difference in performance is caused by the different clock frequencies of the designs given in Table 4.7. The tools achieve higher clock frequencies for the base implementation of the GEMM benchmark because the kernels of the HPL communication phase consume additional resources. Also, the matrix multiplications work on smaller matrix sizes of just a single block, which reduces the reuse of data in local memory.

Our HPL implementation achieves 14.3 TFLOP/s for the base version and 20.8 TFLOP/s for the optimized version using IEC on 25 BittWare 520N FPGAs. The scaling experiments show, that the two major reasons for the performance differences rely on the achieved frequencies and a more efficient use of the global memory. Although the benchmark is computation-bound, our optimized version using IEC still achieves higher performance by reducing the number of

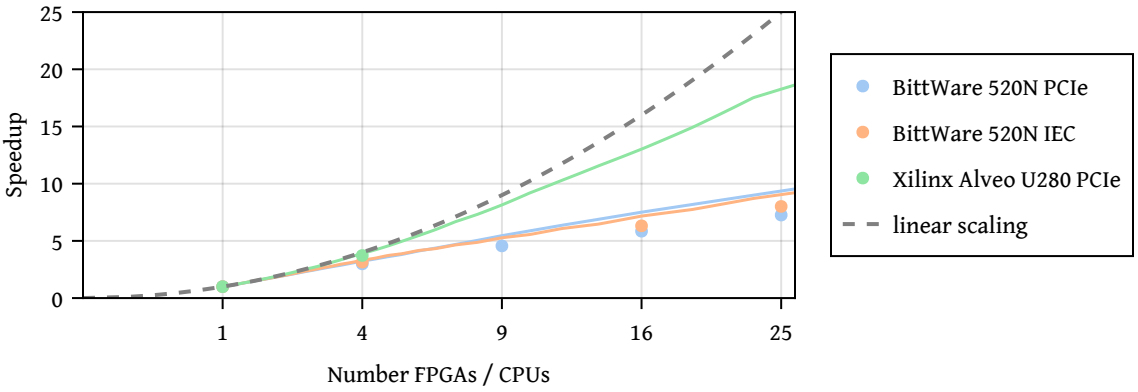


Figure 4.15: Speedup of HPL with a matrix width of 24,576 elements over multiple FPGAs in a strong scaling scenario. Extrapolation models for the three different bitstreams are given as colored lines. They are based on the measured single-FPGA performance per matrix size in Figure 4.13.

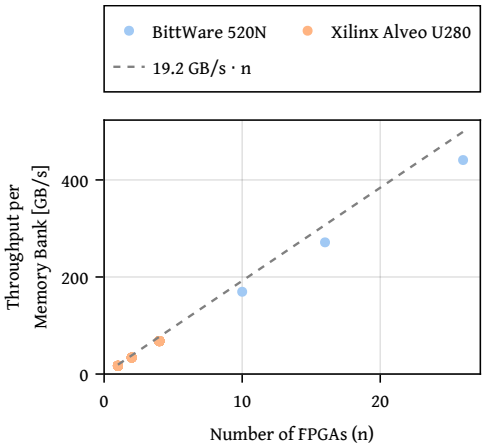
LSUs. This allows further global memory optimizations and higher kernel frequencies that improve the performance.

4.2.4 Evaluation of the Existing Benchmarks

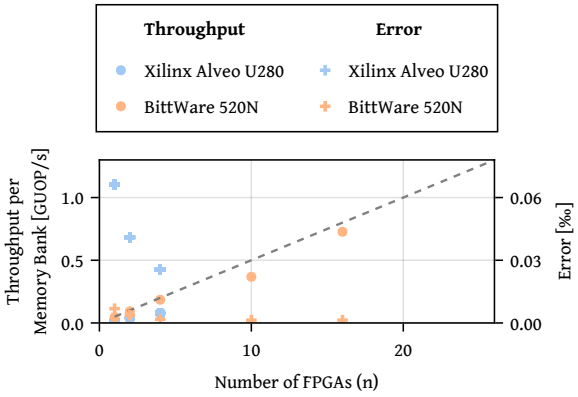
For STREAM, FFT, and GEMM, the design did not change compared to the previous work. All except RandomAccess are executed embarrassingly parallel, so every MPI rank works on a local problem. MPI is only used to exchange measurement and validation results. In the case of RandomAccess, the data array is distributed among the FPGAs. This way, only scaling to a power of two is allowed since the total size of the data array must be a power of two.

We executed the four benchmarks on up to 26 FPGAs to show their scaling performance. 4 GiB arrays are used in STREAM, FFT calculates on $4,096$ 1d FFTs of 2^{17} or 2^9 complex numbers, and GEMM uses matrices with the width of $23,040$ elements per FPGA. RandomAccess is executed in a strong scaling scenario with 8 GiB data array. The normalized measurement results are given in Figure 4.16. For STREAM, the measurements are normalized to a single memory bank with a theoretical bandwidth of 19.2 GB/s. The benchmark shows a similar scaling behavior on both devices. For GEMM, the results are normalized to a single kernel replication running at 100 MHz with with an $8 \times 8 \times 8$ matrix multiplication in registers. This leads to a maximum theoretical performance of 102.4 GFLOP/s times the number of used FPGAs. Also here, the base implementation shows a performance close to the theoretical peak on both devices. Because of the comparably high clock frequency of our synthesized design, we achieved more than 1.2 TFLOP/s per FPGA on the BittWare 520N.

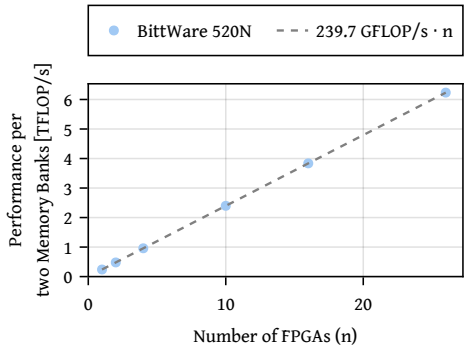
Although we used the same benchmark code on the FPGA side as in Chapter 3, we were not able to execute the FFT benchmark on the Alveo U280. The benchmark required internal channels or pipes between the kernels to forward data but support for pipes in OpenCL kernel code was removed with XRT 2.9. This still allows synthesis of the benchmark, but no execution. On the BittWare 520N, the benchmark scaled linearly. For FFT, we show the absolute measured performance.



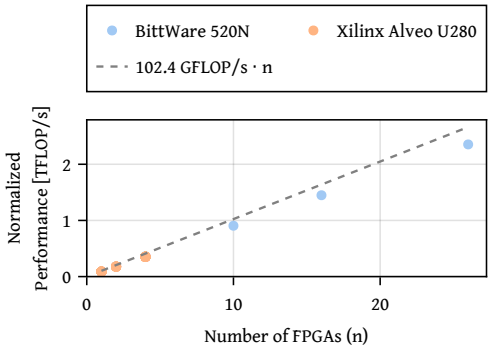
(a) STREAM Triad with 4 GiB · n.



(b) RandomAccess with 8 GiB · n of data.



(c) Batches of 4096 · n 1d FFTs of size 2^{17}.



(d) GEMM with a matrix width of 45 · n blocks.

Figure 4.16: Normalized performance of the four benchmarks without inter-FPGA communication. Data is normalized to a single memory bank and 300 MHz clock for STREAM. For GEMM and RandomAccess, the results are normalized to a single kernel replication at 100 MHz to allow a better comparison of the performance efficiency of the baseline designs on the two FPGA boards.

The RandomAccess results were also normalized to the number of memory banks and a kernel frequency of 100 MHz. Because an update of a value requires one read and one write to the memory bank, two clock cycles are required per update, which results in a theoretical peak

performance of 50 MUOP/s per FPGA. On the BittWare 520N, the base implementation gets close to this theoretical peak whereas on the Alveo U280, we only get roughly half of this performance. One reason for that is the difference in the configuration: For the Alveo U280, we need a small buffer to read and write multiple values subsequentially which partially hides the latency of memory accesses and increases performance. As a trade-off, we see an increased error rate, because we may overwrite values, that are already in the buffer. Still, this approach requires iteration between two different pipelines that fill and empty the buffer. Since these pipelines have a considerable latency because of memory accesses, this will also reduce the performance because the pipelines have to be emptied frequently. If it would be possible to ignore the dependency between reads and writes, the single-pipeline approach could be used as it is done for the BittWare 520N.

4.2.5 Overall Benchmark Results

In the previous section, we have focused on the evaluation of the performance efficiency of the proposed benchmark designs on Noctua and the HACC system. By normalizing the kernel frequency or the number of memory banks and comparing the results to simple performance models, we showed that the baseline versions of the benchmarks have a similar performance efficiency on both tested devices. The absolute performance numbers obtained with the benchmarks are given in Table 4.8. The first column contains the performance numbers obtained from the HACC system using four FPGAs, the second and third columns contain the numbers for 16 FPGAs on Noctua for the baseline version and the optimized version using IEC. The results for STREAM, RandomAccess, FFT, and GEMM are only given for the baseline version because they were not changed for the optimized runs.

Table 4.8: Multi-FPGA Benchmark results.

Benchmark	**Baseline ETH HACC**	**Baseline Noctua**	**IEC Noctua**	**Baseline CPU Noctua**
STREAM Triad [GB/s]	33.9	67.8	–	75.9
RandomAccess [GUOP/s]	0.04	0.18	–	0.06
FFT [GFLOP/s]	–	239.99	–	24.62
GEMM [GFLOP/s]	493.2	1,231.2	–	1,158.9
HPL [GFLOP/s]	568.6	9,552.3	13,326.6	5,848.7
PTRANS [GB/s]	2.55	11.48	293.45	2.43
b_eff [GB/s]	0.56	1.10	19.47	7.96
b_eff [μs]	59.56	61.18	0.34	0.75

In the last column, we executed HPCC 1.5.0 over 16 CPUs on Noctua. The benchmark suite was compiled with GCC 11.3.0, OpenMPI 4.1.1 and Intel MKL 2022.0.1. For basic optimizations, we enabled OpenMP support and set the compilation flags `-O3 -march=native`. During execution, we set the number of OpenMP and MKL threads to 20, which matches the

number of cores per CPU. Moreover, we bound the MPI ranks to sockets and configured the benchmark suite to use the same input sizes we used for the FPGA execution.

We executed the Embarrassingly Parallel (EP) versions of the HPCC benchmarks because our FPGA versions are based on their benchmarking rules. Still, there are some differences between the CPU and FPGA benchmarks: STREAM, FFT, GEMM, PTRANS, and HPL are executed in double-precision floating-point on the CPU, whereas single-precision is used on FPGA. However, for STREAM and PTRANS this should have only a minimal impact on the results since both benchmarks are very likely not compute-bound because of their low arithmetic intensity and the performance is reported in GB/s. FFT is executing one large FFT of size 2^{27} instead of a batched execution of $4,096 \times 2^{17}$ FFTs per rank. This means that the total number of FLOPs and the arithmetic intensity slightly differ between the execution of FPGA and CPU. HPL makes use of partial pivoting within the LU factorization whereas the FPGA implementation does not use pivoting. The results for all benchmarks are reported as an average over all used FPGAs/CPUs as it is done in the original HPCC benchmark suite. Only PTRANS and HPL output the total performance of the whole system.

The performance difference in STREAM for the two FPGA baseline versions is caused by the number of DDR memory banks on the FPGAs. The U280 has two DDR memory banks whereas the 520N has four which directly reflects in the measured bandwidth. The used CPU is equipped with 6 memory banks and shows the highest bandwidth in this result. However, HBM2 can be used on some FPGA boards like the Alveo U280 to considerably increase the STREAM bandwidth as shown in Chapter 3.

For RandomAccess, the FPGA implementation used on Noctua without any local memory buffers shows clear performance benefits compared to the cached version used on the ETH HACC system and the CPU.

There is a considerable performance difference between the baseline version and the optimized version of HPL on Noctua. As we have seen in Figure 4.13, the efficiency of the benchmark design is reduced for the baseline version due to differences in the memory interface. In addition, the clock frequency of the design is slightly lower than for the optimized design as given in Table 4.7.

For PTRANS and b_eff, we can observe the largest performance difference between the baseline and the optimized version on FPGA. For b_eff, the communication latency and bandwidth are reported for all systems. For the FPGA baseline versions, the communication latency is very high compared to the CPU and the optimized FPGA version. The data is first copied from the FPGA DDR memory to the CPU memory before it can be transferred via MPI. These additional copies, the latency of the data transfer via PCIe, and the additional latencies introduced by the FPGA runtimes have a huge impact on the overall latency of the baseline communication approach. As our experiments show in Figure 4.10, the b_eff benchmark included in the HPCC suite is not able to achieve a bandwidth close to the

theoretical peak of 12.5 GB/s with the 2 MB message sizes used for this measurement. With larger message sizes, bandwidths close to the theoretical peak may be achievable.

The proposed benchmarks are capable of reflecting the advantages of direct inter-FPGA communication in terms of communication bandwidth and latency. The results also show that direct inter-FPGA communication offers new opportunities for scaling FPGA applications over multiple devices.

4.2.6 Scalability of the HPL Implementation

Table 4.9: Resource usage of HPL for BittWare 520N with 100 Gbps serial interfaces.

Logic	396,597	43%
BRAM	8,828	75%
DSP	2,809	49%
Frequency [MHz]	225.00	
Compile time [h]	11.4	

To demonstrate the compatibility of the benchmark with different compute systems as well as the scalability of our LU decomposition implementation, we also executed our HPL implementation on the Albireo nodes of the Cygnus system at the University of Tsukuba. The compute nodes of the system are equipped with two Intel Xeon Gold 6126 and two *BittWare/BittWare 520N* boards. In contrast to the 520N boards used on the first evaluation system, these boards are equipped with faster serial interfaces supporting up to 100Gbps per interface. The serial interfaces of the 64 FPGAs are connected to a 8×8 torus.

For synthesis, the configuration in Table 4.6 is used and the results given in Table 4.9 are very similar to the results in Table 4.7. Logic consumption is slightly higher because of the differences in the serial interfaces. The higher logic consumption leads to a more challenging placement and routing on the FPGA, which reduces the final kernel frequency. We executed the benchmark two times with a small size of $3,072 \cdot t$ and a large size of $24,576 \cdot t$ where t is the torus width. Results of the execution are given in Table 4.10. The single FPGA performance is slightly lower compared to the previous experiment because of the reduced clock frequency. Still, the scaling factor for the large size of 56.7 is very close to linear scaling also for this large number of FPGAs. Thus, the scaling behavior is comparable to the previous experiments, which shows that the benchmark achieves good performance and scaling behavior also on this system.

Table 4.10: Execution results of HPL on all 64 FPGAs of Cygnus.

Torus Size	**Small Size [TFLOP/s]**	**Large Size [TFLOP/s]**
1×1	0.18	0.86
8×8	8.48	48.78

4.3 Chapter Conclusion

In this chapter, we extended the HPCC FPGA benchmark suite with support for multi-FPGA systems and their inter-FPGA communication interfaces. Therefore, we proposed a scalable version of the RandomAccess benchmark and extended all existing benchmarks with multi-FPGA support. Moreover, we added three new benchmarks, b_eff, PTRANS, and HPL, for Xilinx and Intel FPGAs that stress inter-FPGA communication and provided baseline implementations making use of MPI and PCIe for inter-FPGA communication. The baseline implementations show similar normalized performance and scaling behavior on our two evaluation systems with up to 26 BittWare 520N and four Xilinx Alveo U280 boards. This makes our benchmark suite the first suite supporting the scaling and parallel execution of all benchmarks over multiple-FPGAs and complete multi-FPGA systems.

To show the extendability of the benchmark suite with support for vendor-specific communication interfaces, we also provided optimized versions of the new benchmarks that make use of the IEC OpenCL extension for direct point-to-point connections between FPGAs. Evaluation of the vendor-specific and the baseline implementations revealed the advantages of direct inter-FPGA communication over communication via MPI not only for the communication-bandwidth-bound applications but also for computation-bound applications like HPL.

With HPL, we also proposed a well-scaling LU decomposition implementation in a 2D torus. The evaluation showed that the performance of the implementation is limited by the aggregated matrix multiplication performance of the used devices. With further architecture-specific optimizations to increase the clock frequency of the implementation, more than 1 TFLOP/s per FPGA on the BittWare 520N is within reach with the proposed design.

5

Case Study: Shallow Water Simulation

In the previous chapters, we showed that the benchmarks of our HPCC FPGA suite enable the analysis of performance differences across tool versions, floating-point accuracy, system configuration, and system size.

In the following, we in more detail evaluate the performance of the communication framework ACCL [32] and its network stacks for use in multi-FPGA applications and compare it to an implementation using the host-side communication approach from the baseline versions of our benchmarks in HPCC FPGA [49]. Therefore, we split this chapter into two parts.

In the first part in Section 5.1, we use synthetic benchmarks to measure the differences in resource utilization, communication latency, and throughput for different ACCL configurations and compare them to the communication approach taken by HPCC FPGA. Also, we provide models for communication latency and discuss the limitations and opportunities for different ACCL- and host-based communication approaches. Moreover, we discuss optimization options for ACCL and its network stacks for the inter-FPGA network infrastructure of the Noctua 2 supercomputer, which contains one of the largest installations of FPGAs in the academic HPC domain.

In the second part, in Section 5.2, we port a multi-FPGA shallow water simulation that operates on unstructured meshes [55] and has strong requirements for low-latency communication to Xilinx FPGAs. FPGAs in many cases show the best compute performance and power efficiency compared to other accelerators like GPUs for small to medium sized problems [56], [57]. This emphasizes the importance of low-latency communication for these devices,

especially when scaling to high number of FPGAs. We use our findings from Section 5.1 to identify the best ACCL configuration for this kind of application and communication pattern and evaluate the scaling behavior of the application over up to 48 FPGAs compared to a baseline version using host MPI for communication. In addition, we extend the existing performance models of the shallow water simulation to also reflect communication latency to show the effect of high communication latency on the application scalability.

The contents in this chapter have been presented at the *International European Conference on Parallel and Distributed Computing* in 2024 and presented in the conference proceedings [58].

5.1 Synthetic Benchmarking of ACCL Communication Approaches

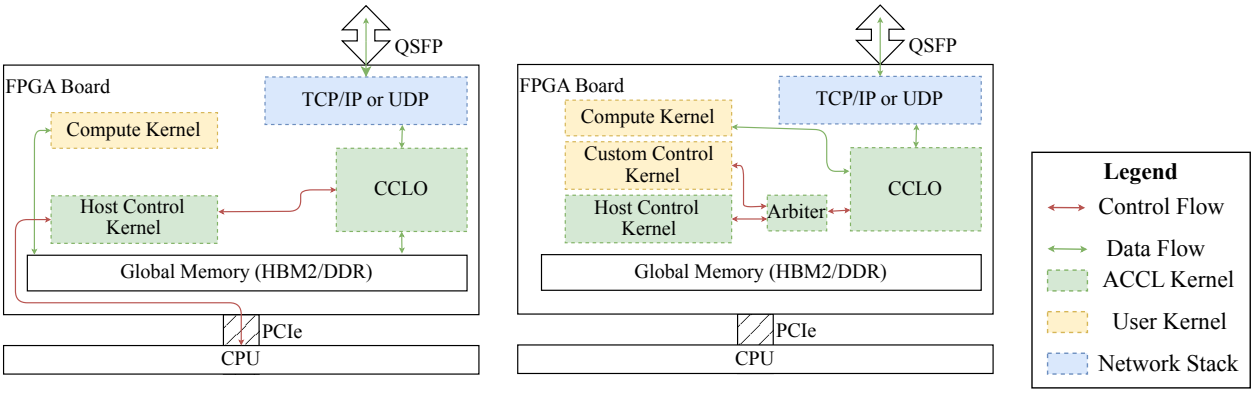
The ACCL framework implements a versatile hardware architecture on the FPGA which can be used for data exchange in various ways. In this section, we identify different communication approaches that are possible with the ACCL architecture and evaluate the performance in terms of communication latency.

5.1.1 ACCL Communication Approaches

ACCL offers two communication approaches: *streaming* and *buffered* communication. *Buffered* communication is similar to the well-known blocking MPI communication, whereas *streaming* communication supports the processing of incoming data before the transmission is complete. This allows further overlapping of communication and computation. As an additional configuration option, ACCL supports the scheduling of communication from the host side (offering more flexibility), or directly from FPGA.

All communication primitives and collectives are implemented within the CCLO, which consists of an CPU to execute the ACCL firmware. Within the firmware, all supported communication primitives and collectives are implemented as a sequence of data move operations between the ACCL components, the network stack, global memory, and user kernels. This approach offers high flexibility also for the implementation of custom collective communication and collective computation tasks. One core feature of ACCL is the plugin system, which supports the extension of ACCL with additional components for specialized computations. With that, the functionality can be extended to the application needs without major changes in the architecture. In many cases, a modification of the firmware can be sufficient to further optimize the communication for a given application. *Streamed* and *buffered* communication are implemented by either passing data directly between the *user kernels* via AXI streams or by moving the data to global memory first. These communication approaches can also be

combined within the same application.



(a) Buffered communication controlled by the host. Data is exchanged between ACCL and user kernels via global memory.

(b) Streaming communication controlled from PL. Data is exchanged between ACCL and user kernels via AXI streams.

Figure 5.1: Two examples of communication approaches using the ACCL framework.

Buffered communication with communication scheduling from the host side is visualized in Figure 5.1a. Here, ACCL will transfer data from a buffer in global memory to another buffer in the global memory of a remote FPGA. The *compute kernel* – the actual application implemented on FPGA – can read the data from this global memory buffer afterwards. The communication is controlled on the host via a C++ library which internally calls an HLS kernel which will again pass the commands to the ACCL via AXI streams. Instead of exchanging data between the *compute kernel* and the ACCL infrastructure indirectly via global memory, it can also be directly forwarded using AXI streams. This approach is indicated in Figure 5.1b as a green AXI stream between *compute kernel* and *CCLO*. A drawback of this approach is, that the order of incoming messages can not be controlled by the receiving side because received data is directly forwarded from ACCL to the AXI stream. If two FPGAs stream a message to the same recipient, the contents of the message will be forwarded in the order of arrival, which may also lead to a scattering of messages. The compute kernels need to be extended to handle these situations.

The other configuration option builds upon the AXI stream interface that is used to issue commands. In addition to the default *host control kernel* (Figure 5.1a) that requires a dedicated kernel invocation for every communication request, it is also possible to implement *custom control kernels* (Figure 5.1b). ACCL already comes with an API that can be used from HLS kernels to implement this functionality. A *custom control kernel* implements the communication pattern required by a specific *compute kernel* and thus can significantly reduce the number of required kernel invocations. *Compute kernel* and *custom control kernel* can also be combined into a single kernel.

5.1.2 Latency and Throughput of PCIe/MPI-based Communication

The measurements of the b_eff baseline version on different multi-FPGA systems in Section 4.2.2 showed that the CPU-centric communication via PCIe and the CPU network only achieves low throughput and high latency compared to CPU-only communication. This observation doesn't surprise since the CPU-only communication is only one of the multiple steps involved in data movement. In general, data first has to be produced by the sending FPGA and placed in the global memory where it can also be accessed by the host. The host moves the data from the global memory to the host memory and from there to the memory of the remote host. The host needs to be notified if the data in the global memory of the FPGA is ready for transmission. The usual way to do this is by invoking a kernel that completes execution as soon as the data is ready. Overall, the process of moving the data from one FPGA to the other as it is done in the baseline implementation of the benchmark can be divided into five sub-steps:

1. Invoke kernel and wait for completion
2. read from global memory on FPGA to host memory
3. transfer data via MPI
4. write data from host memory to global memory on FPGA
5. Invoke kernel to pass data into the compute pipeline

The measurements done in previous work accumulated the latencies for all these communication steps when reporting the results. We extended the b_eff host code with additional execution modes to only measure sub-steps of the communication. The measured latencies for each sub-step are given in Figure 5.2. The kernel invocation shows the highest latency of more than 29 μs on average. The kernel itself writes a single byte to the global memory which should take less than one microsecond. The remaining latency is introduced by PCIe data transfers, shell, and host drivers.

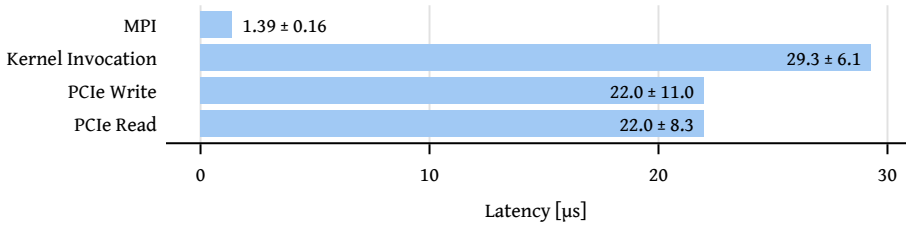


Figure 5.2: Latencies of data movement operations for a 1 Byte message measured individually.

5.1.3 Resource Utilization of the Network Stack

We used the benchmark *b_eff* from our HPCC FPGA benchmarks suite from Section 4.1.1 to evaluate the resource utilization of ACCL and the network stack on the FPGA. Therefore, we extended the benchmark suite with support for the XRT C++ API, making the use of OpenCL on the host side optional. In addition, we added support for C/C++-based kernels, fully removing the mandatory OpenCL requirements for the benchmark suite [59].

b_eff is a synthetic benchmark where the FPGAs are arranged in a (virtual) ring topology to exchange messages. The messages are sent for a given range of message sizes in a ping-ping fashion between the neighbors in the ring and can be used to measure the latency and throughput of the network. For our evaluation, we used the Xilinx Alveo U280 boards from the Noctua 2 FPGA partition described in Section 2.3.

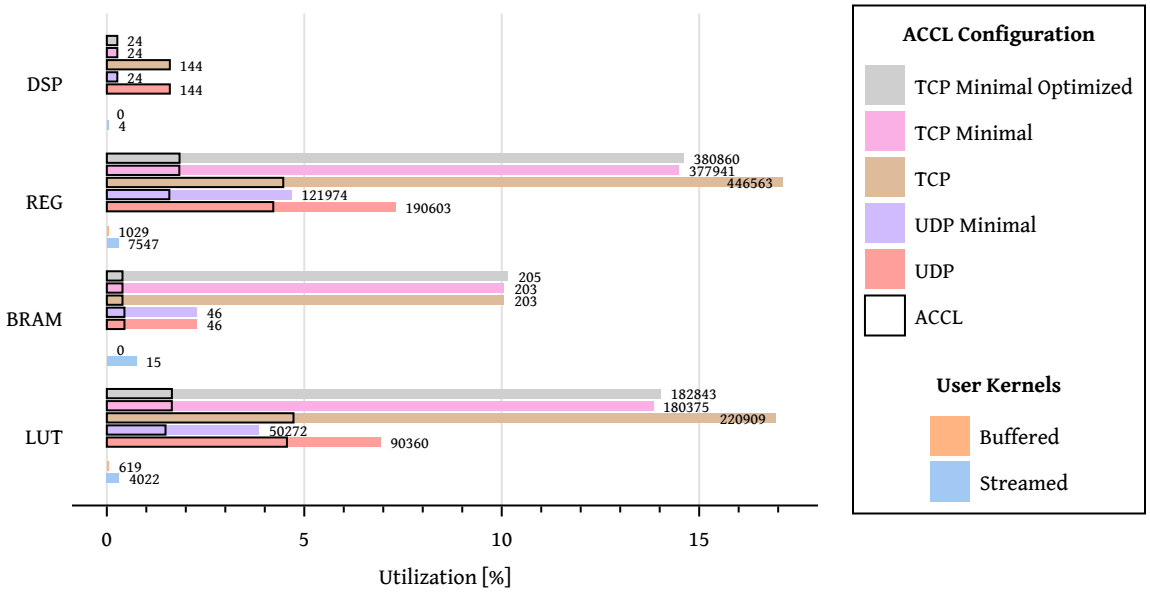


Figure 5.3: Resource utilization of the network stack and ACCL on the Alveo U280. The ACCL *Minimal* versions do not contain the compression and arithmetic plugins. The resource utilization of ACCL (green boxes in Figure 5.1) is highlighted by black boxes. All further resources are consumed by the network stack (blue boxes in Figure 5.1).

We extended the benchmark suite with support for ACCL and implemented ACCL-enabled *b_eff* designs based on the implementation presented in Figure 5.1b, now further differentiating between the TCP-IP and the UDP network stack. The resource utilization for different ACCL configurations is given in Figure 5.3. Further variants are shown, because ACCL is extendable by plugins which are included by default and provide extra functionality for data compression and arithmetic operations used in collectives, such as reductions. These plugins can be removed

from the ACCL configuration by setting build flags during synthesis. Since functionality is not needed by *b_eff*, we also synthesized a *minimal* ACCL without unused plugins to save additional resources. Moreover, to optimize the TCP throughput with the Ethernet switch, we synthesized an *optimized* TCP stack configuration. We discuss these optimization steps in more detail in Section 5.1.4. As expected, the TCP network stack consumes considerably more resources compared to the UDP stack. Our minimal ACCL version saves more than half of the logic resources and more than 83% of DSPs independent of the used network stack.

5.1.4 Modelling and Measurement of Throughput and Latency

We executed the *b_eff* benchmark for the different communication approaches discussed in Section 5.1.1 on two FPGAs. For the first experiment, we configured the optical switch to create a direct connection between the FPGAs – bypassing the ethernet switch. Furthermore, the HPCC FPGA version from the original benchmark without ACCL was used to retrieve data for a purely CPU-based baseline. The two FPGAs are located on different nodes, such that data transfers of the baseline version use the Infiniband network of the hosts via MPI.

The communication latencies over the message size are given for *streaming* and *buffered* communication in Figure 5.4, each in combination with communication scheduling from the host and with a custom control kernel from the FPGA (denoted here as *Programmable Logic (PL)*). We also integrated a performance model for the different approaches as dashed lines in the plot. As expected, the baseline communication approach shows the highest communication latency over all message sizes with latencies of more than 120μs for 64 Byte messages. Using the buffered ACCL communication scheduled from the host side, the major limitation for the latency becomes the kernel scheduling time. The ACCL host control kernel needs to be executed two times for sending and receiving a message. We measured around 30µs of latency for a single kernel invocation through the used XRT. In contrast, messages scheduled directly from *PL* reach latencies below 3µs.

We modeled the latency for buffered and streamed communication using the theoretical peak throughput of the involved links as well as our measured kernel start overheads. For buffered communication, this results in the model given in Equation 5.1, where l_k is the time required to schedule a command to ACCL, l_m the latency to copy the message from the receive buffer in global memory to the destination buffer on the receiving FPGA, and l_c is the latency of the communication link. For host-scheduled communication, l_k equals the kernel invocation latency, whereas for PL scheduling it is reduced to a fraction of microseconds since it only represents the time required by ACCL to process the command. We implemented a synthetic benchmark to empirically determine l_k for commands scheduled from custom control kernels and refine our latency model [60].

$$2 \cdot l_k + l_m + l_c \tag{5.1}$$

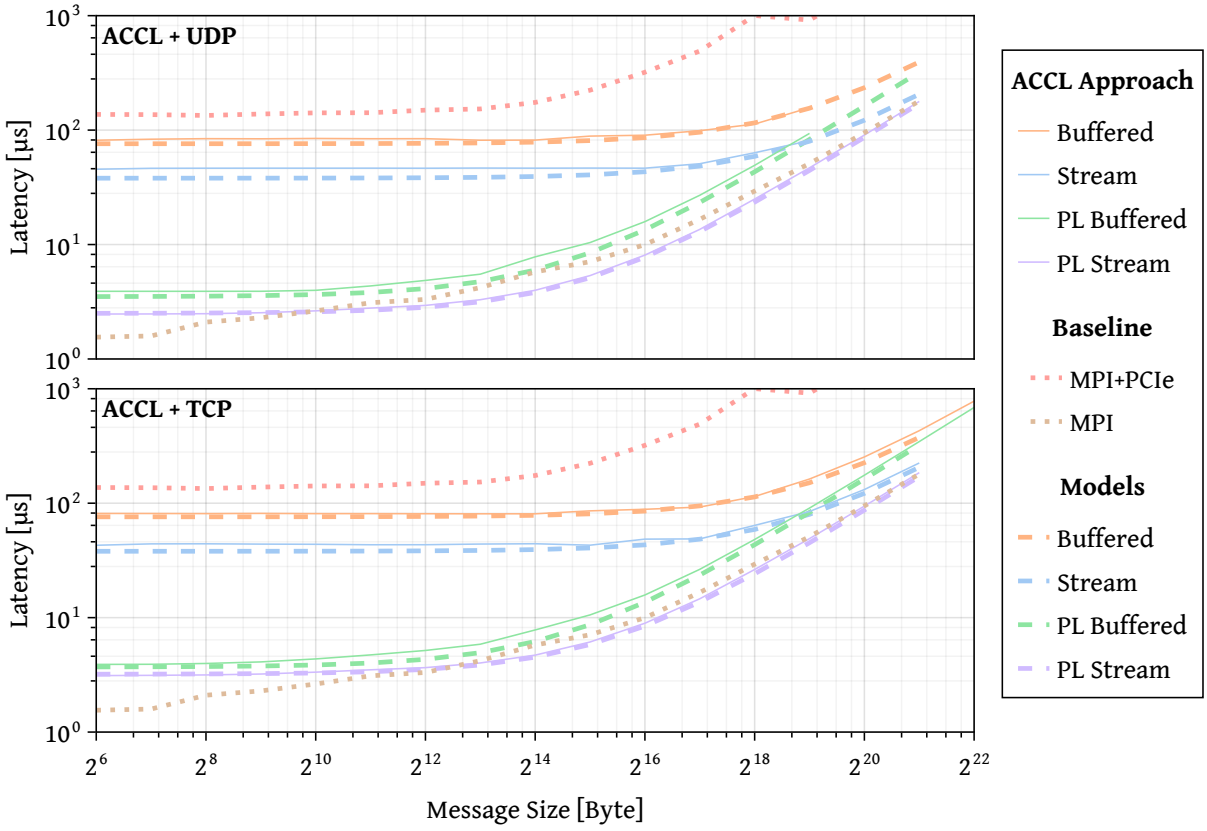


Figure 5.4: Full-duplex communication latencies using ACCL UDP and TCP network stacks and the discussed communication approaches. The latencies for host-side scheduling are modeled for buffered and streaming communication. All measurements are done with directly connected FPGAs.

For streaming, the model simplifies to $l_k + l_c$, because only a single kernel invocation is required per transmission. Also, there is no copy operation required since the data is directly passed to the AXI stream of the user kernel. While l_k is a constant overhead per ACCL command, l_c and l_m depend on the message size and overtake the equation for large message sizes. The additional copy operation required in buffered communication thus leads to a reduced theoretical peak throughput of $(14\text{GB/s}^{-1} + 12.5\text{GB/s}^{-1})^{-1} = 6.6\text{GB/s}$ – only slightly more than half of the peak throughput of the communication link.

5.1.5 Impact of Ethernet Switch and Noctua 2 Optimizations

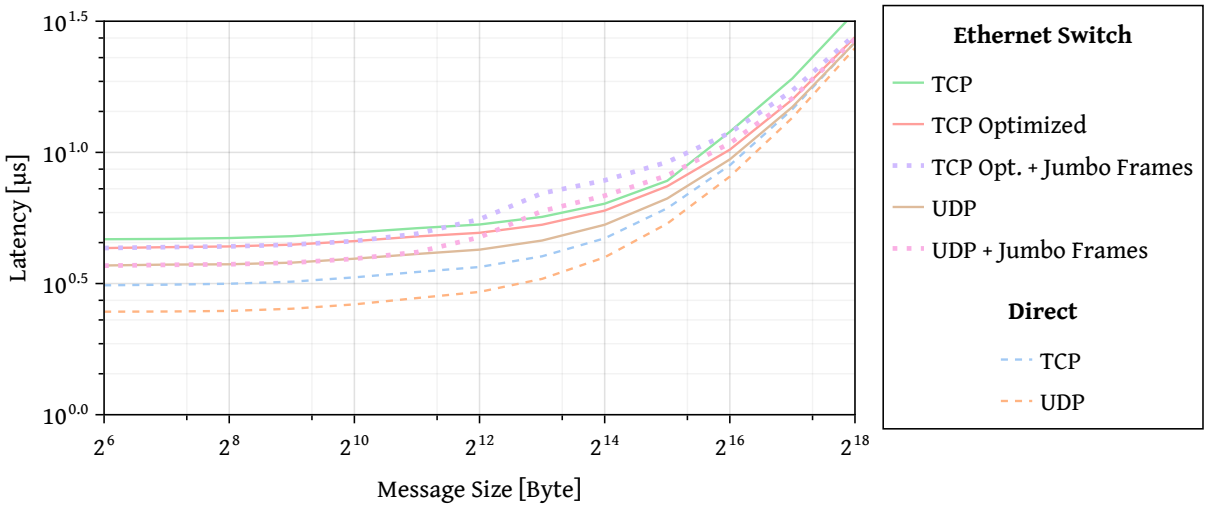


Figure 5.5: Latencies of the PL stream approach executed on two FPGAs with directly connected QSFP links and connected via the Ethernet switch.

Additionally, we compared the communication latency of the network configuration with direct optical links to the latency of connections via the ethernet switch. The measurements via the ethernet switch for the optimized TCP and UDP stack are given in Figure 5.5. Overall, the ethernet switch adds around 1μs of latency for both network stacks resulting in latencies of 2.5 to 5 µs for small 64 Byte messages. For the TCP stack, the throughput was at first considerably reduced when using the Ethernet switch. Because of the increased communication latency, the sending side has to stop transmission and wait for acknowledgments. In our optimized TCP implementation, we enabled window scaling to overcome this issue, with a minor impact on resource consumption as shown in Section 5.1.3. Moreover, we analyzed the reduction of the overall protocol overhead by enabling jumbo frames on the ethernet switch and increasing the maximum segment size for the TCP stack and the maximum packet size in the ACCL firmware accordingly. These measurements are labeled with *jumbo frames* in the plot. All changes together increased the throughput for large messages from initially

8.5 GB/s with the TCP stack to 12.3 GB/s for both network stacks. However, it can also be observed, that the use of jumbo frames harms the latency for small messages. Especially for sizes between 2 KiB and 8 KiB we see an increase in latency with both network stacks when using jumbo frames. This overlaps with the differences in frame size which is increased from 1460 Bytes to 8960 Bytes. Using the streaming communication approach, data is forwarded to the user kernel as soon as it arrives on the receiving FPGA. In consequence, smaller frame- and packet sizes can improve the overall communication latency for small messages because the processing of the incoming data can overlap with the actual transmission. For large messages with the size of hundreds of KB, the overlapping of communication and computation using streaming communication can also efficiently take place with jumbo frames. However, not using jumbo frames reduces the peak throughput in our experiments from 12.3 GB/s to below 12 GB/s. This means when optimizing the network stack for a specific communication pattern, adjusting the packet and frame size has to be considered to optimize throughput or latency.

The measured ping-ping latency for ACCL with UDP or TCP stack and 64 Byte messages is given in Figure 5.6. As discussed, there is also a considerable difference between streamed and buffered communication because of the additional copy operation. Our Ethernet switch adds an additional 1 µs latency to every configuration. Still, all configurations support communication latencies in the microsecond range.

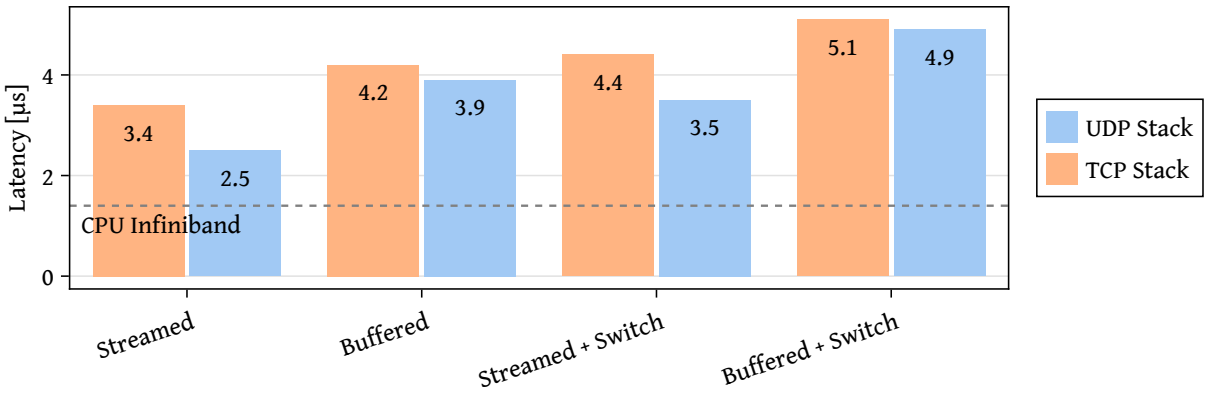


Figure 5.6: 64B Ping-Ping Latencies for the different ACCL communication approaches using the UDP or TCP stack. The latency for the host Infiniband network is given for reference.

5.2 Acceleration of Shallow Water Simulation using ACCL

Based on the results of Section 5.1, we ported a 2D shallow water simulation with high communication constraints to Xilinx FPGAs and extended the implementation to support

halo exchange via ACCL to improve the performance and scalability of the application.

5.2.1 Implementation

The evaluation of ACCL using synthetic benchmarks showed that low latency communication in the order of a few µs is possible. We applied the lessons learned to a full FPGA accelerated HPC application: A discontinuous Galerkin shallow water simulation on unstructured meshes which was originally implemented for Intel FPGAs [61] and was further extended for multi-FPGA execution in custom circuit-switched communication networks [55].

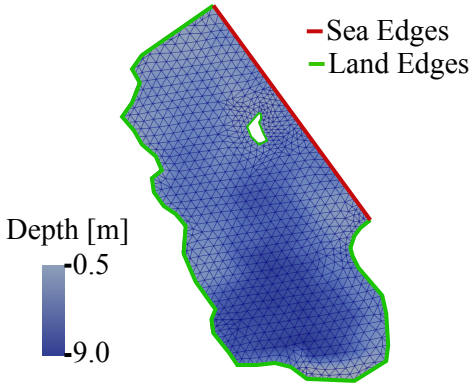


Figure 5.7: Computational mesh consisting of 1696 elements [55]. The boundary edges represent the coastline (land edges) and the border to the open sea (sea edges).

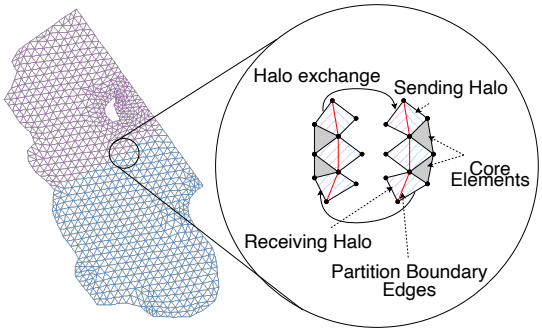


Figure 5.8: The mesh is partitioned as indicated by the colors. Data has to be exchanged between neighboring partitions in every simulation time step in the form of the partition halo. Based on Faj, Kenter, Faghih-Naini, *et al.* [55], Fig. 4

The simulation makes use of 2D shallow water equations which are derived from Navier-Strokes equations. They are a classical model to simulate tides or tsunamis but also other kinds of fluid or even atmospheric flows. The tidal flow of the bight of Abaco on Bahamas islands is used as a simulation scenario as given in Figure 5.7. The water surface of the bay is represented by an unstructured mesh. The borders of the mesh are either land or sea edges which have to be handled differently in the simulation. For the execution over multiple FPGAs, the mesh is partitioned into sub-meshes as visualized in Figure 5.8 for two partitions indicated by different colors. The mesh partitions are distributed among the FPGAs, such that every FPGA handles exactly one partition. In each simulation step, the halo around the partition edges has to be exchanged between FPGAs storing neighboring partitions using point-to-point communication. The halo consists of all mesh elements with boundary edges to the neighboring partition.

The designs in [61] and [55] support three types of polynomial discontinuous Galerkin discretizations, however, it has been shown by Faj et al. [55] that the requirements for communication latency are very similar for all three types of discretization. Thus, we will only focus on the piecewise constant discretization in our evaluation. For communication, the original implementation utilizes the QSFP ports of the FPGAs via IEC. Therefore, the FPGAs form a circuit-switched network where FPGAs holding neighboring partitions in the mesh are directly connected to exchange the halo regions via low-latency one-hop communication. This setup restricts the maximum number of neighboring partitions to the number of QSFP ports installed on the FPGA board. For a high number of FPGAs, finding an efficient partitioning of the mesh under the given restrictions is challenging, effectively limiting the scalability of the original implementation, where only at most 10 FPGAs are used.

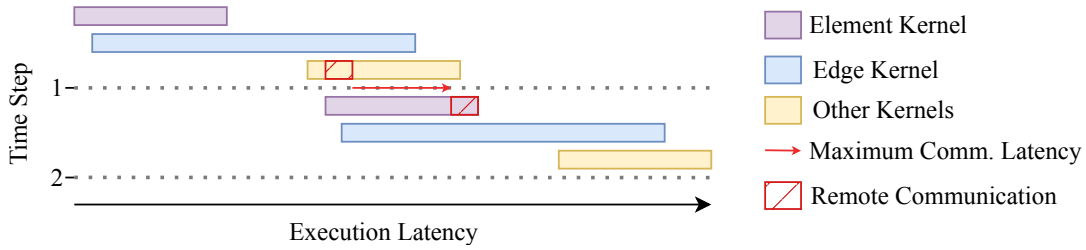


Figure 5.9: Dataflow schematic for shallow water simulation. Full overlap of compute kernels across simulation time steps. The maximum communication latency is indicated by the red arrow. If communication takes longer, the compute pipeline will stall until data is received. Simplified dataflow based on Faj, Kenter, Faghih-Naini, *et al.* [55], Fig. 5

Figure 5.9 shows the dataflow graph over two simulation time steps for the compute pipeline in a single FPGA. All simulation data is loaded into the local memory of the FPGA at the beginning of the simulation so global memory accesses are only used during simulation to write back intermediate results. The *element kernel* updates all entities of the unstructured mesh element-wise. The L^2 projection is directly forwarded to the *edge kernel* which will update all boundary and outer edges of the unstructured mesh. Afterwards, the boundary edges between partitions of the unstructured mesh will be sent within the *other kernels* to the remote FPGAs via ACCL. This data will be received at the end of the execution of the *element kernel* in the next time step. To prevent pipeline stalls, the data has to arrive at the remote FPGA within the latency indicated by the red arrow which is typically a few thousand clock cycles. The overall latency between sending and receiving boundary elements depends mainly on the number of core elements that will be updated by the *element kernel* in between. Core elements are elements that are not located on a border of the local partition and which do not require any data from remote FPGAs. This overall model is the same as in the original implementation, but the impact of high communication latencies or a low number of core elements per partition was not considered in the performance models and evaluation in [55].

The data is streamed through the kernels element-wise as given in Figure 5.10. The mesh partition will be updated on the local FPGA and all boundary elements are forwarded to a *communication kernel* which has the task of passing this data to the compute pipeline on the remote FPGA. In the baseline implementation, the *communication kernel* is invoked by the host for every simulation step. It will write all received data into a buffer in global memory and finish execution after all elements for one simulation step are written to notify the host that the data is ready for sending via MPI. The remote host calls the *communication kernel* again to read the data from global memory to the *element kernel* for the next simulation step.

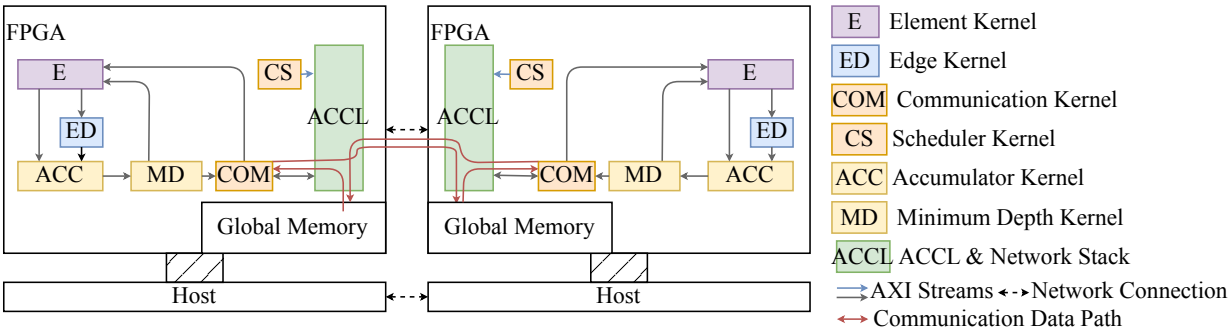


Figure 5.10: Remotely partitioned FPGA designs with two processing pipelines distributed over two FPGAs with ACCL communication via AXI streams and communication scheduling in PL.

For the ACCL-enabled version, the *communication kernel* converts the simulation data into a generic 512-bit AXI stream used to directly pass the data into the ACCL communication stack. This way, the actual simulation loop stays unchanged. In addition, we use a communication scheduler kernel to issue the send and receive commands directly from PL. This massively reduces the number of kernel invocations from the host side.

For a high number of partitions, there will be more than one neighboring partition and the *element kernel* expects the remote elements from the communication kernel in a predefined order. As a result, the incoming data has to be reordered on the receiver side before it can be passed onto the simulation pipeline. Instead of creating our own logic for this task, we make use of ACCL's buffered communication feature. Therefore, we buffer received data in global memory first and move the data from the global memory into the AXI stream using the `recv` primitive as indicated by the red arrows in the figure. In the end, the resulting communication scheme is a mixture of streaming communication on the sending side and buffered communication on the receiving side. This approach slightly increases the communication latency because data has to go through the global memory instead of passing it directly to the communication kernel. As a main advantage, we have no strict upper limit for the number of neighboring partitions since we can configure the number of receive buffers during runtime.

5.2.2 Performance Model

The port of the simulation to Vitis-compatible C++ code was possible without major changes in the dataflow characteristics. This also means, that the performance models for the simulation proposed by Faj, Kenter, Faghih-Naini, *et al.* [55] also hold for this implementation. However, their throughput model does not consider the communication latency, which is important for strong scaling scenarios and small local partition sizes, since the calculations on the core elements may not be sufficient to hide the communication latency. Because of this, we extended the existing throughput model as given in Equation 5.2 with the communication latency L_{comm} representing the latency for the FPGA with the highest number of neighbors according to the partition scheme and the largest number of sent or received elements per simulation time step.

$$throughput = f \cdot \frac{FLOP_{total}}{max(E_{core} + D_{ext}, L_{comm}) + E_{send} + E_{recv} + L_{pipe}} \tag{5.2}$$

Additionally, we model the communication latency based on our latency measurements in Section 5.1, our ACCL latency models, as well as mesh partitioning information as given in Equation 5.3. As described in the previous section, to receive the data, it has to be read from a buffer in global memory with a latency of l_m. The maximum number of neighbors N_{max} for a partition scheme has a major effect on this read latency because the read commands have to be scheduled for every neighbor. This adds l_m to the overall communication latency for every neighbor as given in Equation 5.1. The latency to process the commands in ACCL l_k has to be added for every send and receive command.

$$L_{comm} = \frac{E_{send} + E_{recv} + 2 \cdot N_{max} \cdot l_k + N_{max} \cdot l_m}{f} + L_{pingping} \tag{5.3}$$

Also, the overall latency of the communication link has to be considered, introducing another latency $L_{pingping}$, which is the ping-ping latency of the largest message exchanged among neighbors. For the total number of floating-point operations $FLOPS_{total}$ we use the simplified model $FLOPS_{total} = FLOP_{sum} \cdot E_{total}$ without the additional operations required to calculate the projection on the received elements. Instead, we calculate the FLOPs based on the total number of elements in the mesh E_{total} and the number of floating point operations per element $FLOPS_{sum}$, independent of the partitioning to make scaling experiments better comparable.

5.2.3 Evaluation

We also synthesized the base and ACCL version of our shallow water simulation for the FPGA partition on Noctua 2 as described in Section 2.3. For our application, we use the UDP minimal and TCP minimal optimized configurations of ACCL as described in Section 5.1.3.

The resulting resource utilization is given in Table 5.1 for the used configurations. The increased resource utilization compared to the base version closely reflects the resource usage of the ACCL stack. The implementation supports setting the maximum number of elements per partition, which mainly affects the BRAM and URAM utilization since the whole partition is stored in local memory. The baseline and UDP versions are synthesized with a partition size of 8192 elements where larger sizes led to routing congestion because the URAMs holding the local partition can not be easily distributed across multiple Sliced Logic Regions (SLRs). For the ACCL TCP stack, we were only able to synthesize designs with half the partition size, i.e. 4096 elements. Larger local partitions also failed because of routing congestion.

Table 5.1: Resource utilization of the shallow water simulation.

Configuration	LUTs	Registers	BRAM	URAM	DSPs	Freq. [MHz]	Synth. Time [h]
Base	126,646 (9.7%)	182,015 (7.0%)	265 (13.1%)	188 (19.6%)	1,218 (13.5%)	256	4.1
ACCL UDP	176,884 (13.6%)	305,381 (11.7%)	312 (15.5%)	188 (19.6%)	1,242 (13.8%)	274	5.2
ACCL TCP	334,225 (25.6%)	586,847 (22.5%)	344 (17.1%)	101 (10.5%)	1,242 (13.8%)	252	15.1

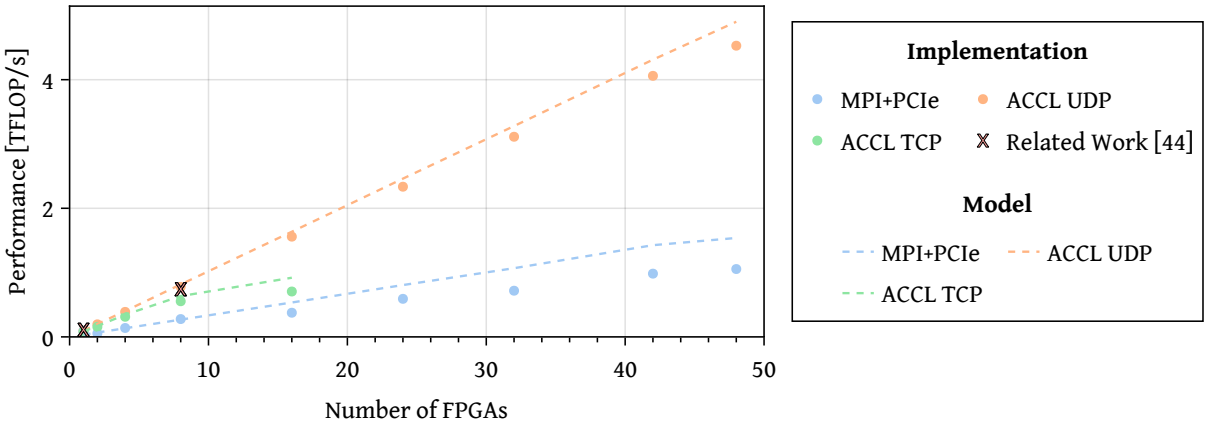


Figure 5.11: Execution times for the weak scaling scenario with ~6000 elements per FPGA. Due to a known issue in the current version of the TCP stack, we did not evaluate this configuration beyond 16 devices.

We first executed a weak scaling experiment with the three design variants of the shallow water simulation. The resulting performance compared to the modified performance model is given in Figure 5.11. We used increasing mesh sizes with up to 312,000 elements to keep the

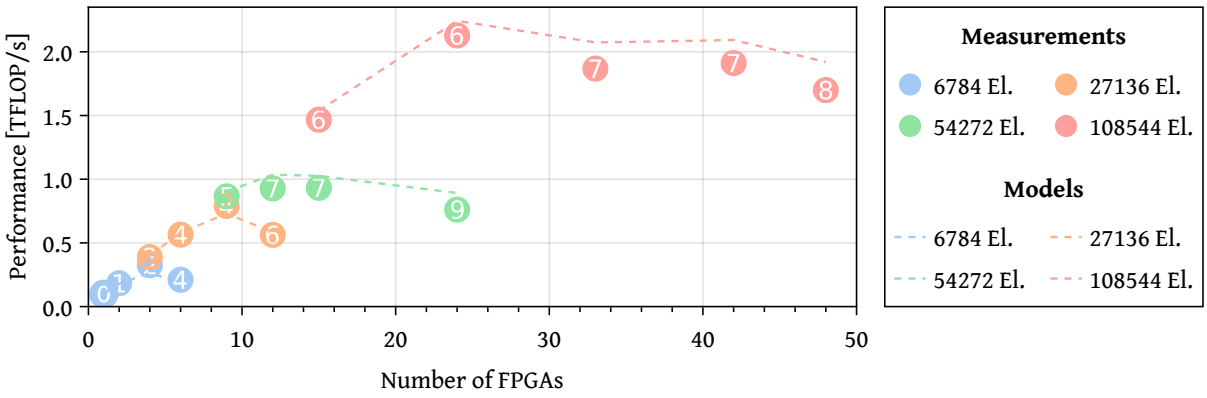


Figure 5.12: Strong scaling scenario with selected mesh sizes with ACCL and UDP stack. The numbers in the plot represent the maximum number of neighbors.

number of elements per partition between 6,000-7,000 elements. The base version annotated as *MPI+PCIe* first shows a reduced performance when scaling from one partition to two partitions. When executed only on one partition, no communication is required, eliminating compute pipeline stalls. Measurements with our synthetic benchmark given in Figure 5.4 showed an expected latency of 100-120µs for small messages of multiple KB size. This is the expected size of the halo exchange messages sent by the shallow water simulation. The simulation processes one element per clock cycle and sufficient core elements are required to hide the communication latency as expressed in the *max* term in Equation 5.2. Based on the kernel frequency of the synthesized bitstream, the pipeline could process around 25,000 to 30,000 elements in this time frame, so for the given partitions, the pipeline stalls approximately 75-80% of the execution time. With their improved communication latency, the ACCL designs with UDP and TCP stack can respectively avoid or reduce such stalls, which leads to higher performance and better scalability, to up to 4.5 measured TFLOPs on 48 FPGAs with the ACCL UDP design.

Furthermore, we executed strong scaling experiments with selected mesh sizes given in Figure 5.12, which additionally depict the maximum number of communication neighbors. The results show that this number has a high impact on the overall performance, because of the additional latency introduced by command scheduling and global memory.

To get more precise input data for our model, we measured the latency l_k and l_m using a synthetic benchmark scheduling copy operations and no-ops from the PL to ACCL [60]. In the first experiment, we repeatedly sent no-ops to the CCLO, which will decode the command and send a return status back to the user kernel. This allows us to measure the minimal ping-pong latency from the user kernel to CCLO for decoding a command and returning the status. On average, we measured 0.24 μs for a single command, which resembles 60 clock

cycles. Additionally, we executed copy operations from global memory into an AXI stream to measure the latencies for l_m for buffer sizes of 100 elements, which resulted in 1.7 μs.

When the local partitions become too small to cover the communication latency, there is no performance improvement by adding further FPGAs. Indeed, the overall performance can even degrade, because further partitioning of the mesh may introduce a higher maximum number of neighbors, which in turn further increases the communication latency. This can be observed for the 108K element mesh, where additional neighbors result in a step-wise decrease in performance. The original implementation of Faj, Kenter, Faghih-Naini, *et al.* [55] is limited to a maximum of four neighbors because of the number of QSFP ports installed on the FPGA board and thus was limited to at most 10 FPGAs for the topology used in their and our experiments. Our ACCL implementation overcomes this limitation, but we see that in strong scaling scenarios, larger local partition sizes or custom message reordering in local memory is required to hide the communication overheads introduced by the increasing number of neighbors.

For the verification of our results, we chose a similar approach to previous work. We executed the CPU reference implementation on the same mesh and compared the sea level and velocity in x and y directions over time on pre-defined positions in the mesh that we refer to as *stations*. Our shallow water simulation does report intermediate results for every simulation hour whereas the reference implementation is doing this for every simulation step. Example results of the sea level verification for a mesh with 217,088 elements executed over 48 FPGAs are given in Figure 5.13. The results collected on the FPGAs are given as dots for four different stations, and the matching CPU results are given as lines in the same color. As can be seen, the FPGA results closely overlap with the results of the CPU reference implementation.

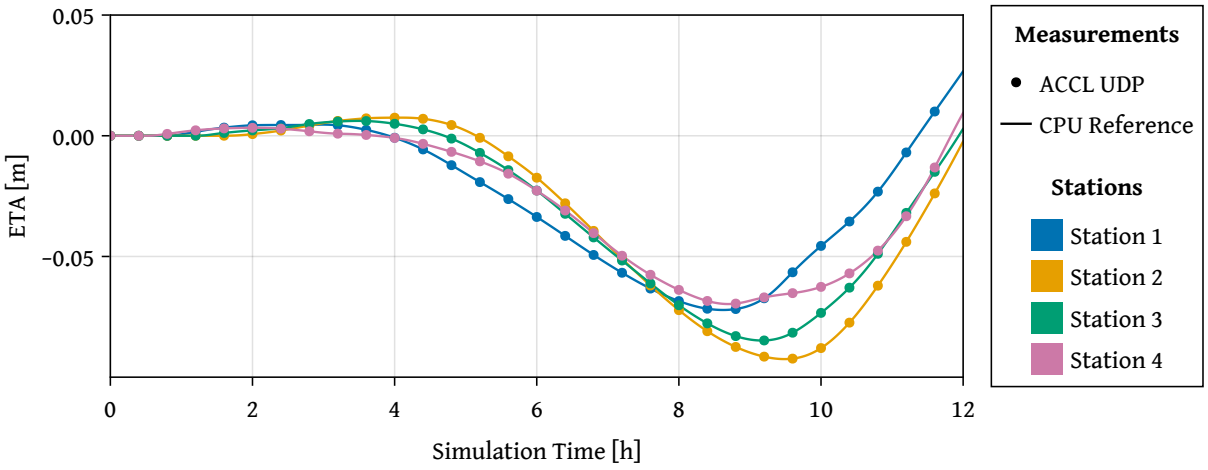


Figure 5.13: Verification of simulation results for the ACCL UDP implementation against a CPU reference implementation with 217,088 elements over 48 FPGAs.

5.3 Chapter Conclusion

In this chapter, we used our synthetic benchmark b_eff to model the communication latency of ACCL for different communication approaches. Our analysis of ACCL's network stacks showed, that fine-tuning the network stacks i.e. in terms of packet sizes can be used to either optimize the throughput or latency of the communication on a per-application basis.

Based on our ACCL evaluation, we ported a OpenCL multi-FPGA shallow water simulation to Vitis HLS and extended it with communication via ACCL. The scaling experiments showed linear speedups in weak scaling scenarios with all 48 FPGAs of the Noctua 2 partition and the same performance per partition but with much better scalability than an implementation using custom circuit-switched networks. However, by evaluating the application with UDP and TCP network stacks, we also showed, that the additional resource utilization of the network stack and communication framework can lead to reduced application performance. In the case of the TCP stack, further scaling of the application failed because of routing congestion, eventually reducing the per-FPGA performance of the application. The flexibility of recent FPGAs and multi-FPGA systems enable us to use the best-fitting network stack or communication framework for an application depending on the targetted communication patterns, throughput, latency, and resource utilization requirements. This makes it imperative for a multi-FPGA communication framework to be configurable to the specific use case to facilitate performant as well as portable multi-FPGA applications.

6

Related Work

We split the contributions of this work into three areas: HLS-/ OpenCL-based benchmarks for multi-FPGA systems, LU decomposition on multi-FPGA systems, and full FPGA-accelerated applications making use of multi-FPGA systems and inter-FPGA communication to scale workloads beyond a single FPGA.

6.1 FPGA Benchmarks for HLS toolchains and Multi-FPGA systems

There already exist several benchmark suites for FPGA and their HLS frameworks. Most OpenCL benchmark suites like Rodinia [62], OpenDwarfs [63] or SHOC [64] are originally designed with GPUs in mind. Although both GPU and FPGA can be programmed using OpenCL, the design of the compute kernels has to be changed and optimized specifically for FPGA to achieve good performance. In the case of Rodinia, this was done for a subset of the benchmark suite with a focus on different optimization patterns for the Intel FPGA (then Altera) SDK for OpenCL [65]. In contrast, to port OpenDwarfs to FPGAs, Feng, Lin, Scogland, *et al.* [63] employed a research OpenCL synthesis tool that instantiates GPU-like architectures on FPGAs. With Rosetta [66], there also exists a benchmark suite that was designed targeting FPGA using the Xilinx HLS tools from the start. It focuses on typical FPGA streaming applications from the video processing and machine learning domains and currently supports the Xilinx HLS framework. The CHO [67] benchmark targets more fundamental FPGA functionality and includes kernels from media processing and cryptography

and the low-level generation of floating-point arithmetic through OpenCL, using the Altera SDK for OpenCL.

The mentioned benchmarks often cannot easily be adjusted to the target FPGA architecture. Modifications have to be done manually in the kernel code, sometimes many different kernel variants are proposed or the kernels are not optimized at all, making it difficult to compare results for different FPGAs. A benchmark suite that takes a different approach is Spector [68]. It makes use of several optimization parameters for every benchmark, which allows modification and optimization of the kernels for a FPGA architecture. The kernel code does not have to be manually changed, and optimization options are restricted by the defined parameters. Nevertheless, the focus is more on the research of the design space than on performance characterization.

Still, in some of the benchmarks, the investigated input sizes are small enough to fit into local memory resources of a single FPGA. However, many HPC applications require more memory for their calculations or need to be scalable, so these benchmark results only have limited validity in this area.

Recent works targeting FPGAs in the HPC domain implement synthetic benchmarks in HLS or OpenCL to generate empirical roofline models [69]–[71]. Similar to the use on CPUs and GPUs, these roofline models can give an upper bound of the performance and optimization guidance for FPGA-accelerated HPC applications. However, the interpretation of such a roofline model for FPGAs is very complex because resource utilization and achieved kernel frequencies affect the overall performance. Also, caching of data through the utilization of local memory such as BRAM and ultra-RAM is usually not reflected by the benchmarks, which is often essential for the performance of real applications.

Zeni, O'Brien, Blott, *et al.* [72] implemented the High Performance Conjugate Gradients (HPCG) benchmark using OpenCL on Xilinx FPGAs, which is well a established benchmark in the HPC domain. The benchmark is currently limited to Xilinx FPGAs and was only evaluated on the Alveo U280. Similar to our work, it comes with multi-FPGA support using the host MPI installation.

6.2 LU Decomposition on FPGAs

The most complex benchmark that we added to the benchmark suite is HPL, where the most compute-intensive part is the LU decomposition. There already exist several implementations for scalable, double-precision blocked LU decomposition implemented in HDL. A multi-FPGA implementation of LU decomposition is proposed by Hauser, Dasu, Sudarsanam, *et al.* [73]. The implementation uses up to five Virtex-II FPGAs for the calculation on a single matrix arranged in a star topology. The FPGAs need to be reconfigured several times during computation and calculate the LU decomposition of an 8192×8192 matrix with double

precision complex numbers using 5 FPGAs in 1862.41 seconds and with that achieving up to 197 MFLOP/s. Wu, Dou, Sun, *et al.* [74] propose a single FPGA implementation for Virtex-5 FPGAs that reaches 8.5 GFLOP/s and that can be easily extended for multi-FPGA execution. A double-precision implementation using HLS for the Xilinx Alveo U50 was done by Kumar, Choudry, and Purini [75] and showed a peak performance of 128 GFLOP/s on a single FPGA. Jaiswal and Chandrachoodan also propose a scalable double-precision block LU decomposition implementation for Virtex-5 FPGAs [76] written in Verilog. They report achieving more than 120 GFLOP/s when scaling over 8 FPGAs.

Turkington, Masselos, Constantinides, *et al.* [77] implement the LINPACK 1000 benchmark in Handel-C and achieve more than 2.5 GFLOP/s on Stratix II. Wu, Dou, Lei, *et al.* [78] achieved more than 3.6 GFLOP/s with their HDL implementation of the same benchmark. Both implementations also include pivoting. Since we are rather taking up on the rules proposed for HPL-AI, pivoting is not part of our implementation.

Zohouri, Maruyama, Smith, *et al.* [65] implemented an OpenCL single-precision LU decomposition without pivoting within the Rodinia FPGA benchmark suite. Execution on an Intel Arria 10 FPGA with a matrix size of 8,192 elements resulted in a performance of above 366.5 GFLOP/s. Our implementation requires much larger matrix sizes to achieve its peak performance but already achieves 493.7 GFLOP/s on Stratix 10 with the given matrix size. However, since our implementation is also scalable over multiple FPGAs, the overall performance is not limited by the resources of a single FPGA.

None of the related work gets close to the measured peak performance of our implementation exceeding 48 TFLOP/s.

6.3 Multi-FPGA Applications

This work in detail analyzes ACCL as an inter-FPGA communication framework for scaling latency-sensitive applications. With that, it is the first work using ACCL within an application but several multi-FPGA applications exist, using either other existing communication frameworks or custom solutions.

Our HPCC FPGA benchmark suite [49] contains multi-FPGA implementations for LU factorization and matrix transposition. Next to the baseline versions using the naive communication approach via PCIe and MPI, it also contains optimized versions of these benchmarks directly utilizing the SerialLite network stack for communication in custom circuit-switched networks. The baseline versions in many cases suffer from high communication latency and limited throughput, making this communication approach unsuitable for most of the discussed application scenarios. For many applications, FPGAs work best on small to medium problem sizes. This, in comparison to GPUs or CPU, sets additional constraints on the communication latency and throughput since overlapping of computation and communication can not be

used as extensively. This is why most related work uses approaches that utilize the boards QSFP ports which show lower latencies and higher throughput.

Most of the related work comes with custom solutions for inter-FPGA communication tailored to the specific application. They often make use of low-level network layers which are tightly integrated into the application design based on custom shells or BSPs. In the following, we will give an incomplete selection of multi-FPGA applications from different domains to give an overview of where multi-FPGA systems and inter-FPGA communication show promising results in recent works.

Perdomo, Kropotov, Cano Ladino, *et al.* [79] implemented an emulation platform for RISC-V designs scaling over multiple FPGA. In their evaluation, they use this platform to execute an emulation using 32 Alveo U55c FPGAs making use of the QSFP28 ports for inter-FPGA communication. They make use of several approaches for inter-FPGA communication: Firstly, they use one of the QSFP ports for communication via Ethernet while the second QSFP port is used for peer-to-peer communication using Aurora. Secondly, they implemented an Ethernet-over-PCI driver solution which allows the communication among FPGAs on the same node. With this setup, they are able to simulate large-scale RISC-V designs, exceeding the logic resources of a single FPGA.

Fujita, Kobayashi, Yamaguchi, *et al.* [80] accelerate astrophysical simulations using FPGAs and the inter-FPGA communication framework CIRCUS. They achieve a high parallel efficiency with a weak scaling scenario on up to four FPGAs. Kobayashi, Fujita, Yamaguchi, *et al.* [81] extends the work utilizing GPUs and FPGAs and using MPI and CIRCUS for communication. The application showed linear speedups for weak scaling on up to two compute nodes or up to two FPGAs, respectively. However, the scaling evaluation for this application is very limited.

Huthmann, Shin, Podobas, *et al.* [82] implemented an N-Body simulation using a custom circuit-switched network. The implementation showed linear speedups in strong and weak scaling scenarios over up to 8 FPGAs. Another N-Body simulation by Menzel, Plessl, and Kenter [56] uses SerialLite via the OpenCL Intel External Channel extension to communicate within a custom circuit-switched network and archives linear scaling in a strong scaling scenario with up to 24 FPGAs. The implementation achieves sub-millisecond time step durations.

A shallow water simulation on unstructured meshes by Faj, Kenter, Faghih-Naini, *et al.* [55] similarly makes use of SerialLite within a custom circuit-switched network. The mesh is therefore partitioned and distributed over the FPGAs. FPGAs holding neighboring partitions are directly connected via the custom circuit-switched network for halo exchanges. Because of that, the number of neighboring partitions is limited to the number of available ports on the FPGA board. The implementation is scaled over up to 10 FPGAs and achieves high parallel efficiency in strong scaling scenarios with a time step duration of only several microseconds. These small time steps set high requirements on the communication latency, thus we picked

this work for our evaluation within Chapter 5.

Sheng, Tong, Jiang, *et al.* [83] perform a molecular dynamics simulation on 64 Xilinx VU13P FPGAs which are interconnected via 200 Gb/s links that form a $4 \times 4 \times 4$ 3d torus network. With this custom system architecture, they outperform multi-GPU accelerated systems over a wide range of simulation sizes. Similar to the NBody and shallow water simulation, low-latency communication is crucial for scaling this kind of simulation over large systems.

Also for deep neural network inference, FPGAs are a promising candidate for acceleration and gained more traction in recent years. However, the fast increase of neural network sizes also increases the demand for more FPGA resources to sustain or even improve throughput. Spreading the inference over multiple FPGA by utilizing direct inter-FPGA communication leads to promising results using two FPGAs as shown by Alonso, Petrica, Ruiz, *et al.* [84] and Jiang, Sha, Zhang, *et al.* [85].

7

Conclusion

This chapter provides a concise summary of the thesis and its findings, followed by a discussion on potential directions for future research based on these results.

7.1 Summary

This work proposed and evaluated a benchmark suite for FPGAs, addressing the need for performance evaluation tools for FPGA-accelerated HPC systems, their development tools, and communication infrastructure. We implemented an OpenCL benchmark suite based on the HPC Challenge for CPUs, offering carefully optimized base implementations for all benchmarks. A crucial selection criterion for the benchmarks was the memory access pattern. We evaluated the benchmarks on four boards from different vendors, utilizing DDR, HBM2, and host memory to store application data. The base implementations were written in OpenCL C, ensuring compatibility with both major FPGA vendors using the same source code without manual modifications. In our single-FPGA evaluations with four different FPGAs of the two major vendors and three different types of global memory including HBM2 and DDR, the measured performance closely resembles the modeled peak performance under resource and global memory bandwidth constraints for most of the benchmarks.

With that, we showed that the implementation of OpenCL C code resulting in performant hardware with the toolchains of both major vendors can be achieved through careful optimization. However, some of the benchmarks like GEMM were not always capable of utilizing a sufficient amount of the hardware resources or the synthesis resulted in low kernel frequencies,

leading to reduced performance. These cases showed the limitation of our *base implementation* approach, where code optimizations could only be applied reluctantly to not negatively affect the application performance with other HLS toolchains. Our evaluation showed, that our benchmarks are capable of measuring differences in the FPGA architectures of Xilinx and Intel FPGAs by configuring the benchmarks with different floating-point precisions. Moreover, we detected changes and limitations in specific versions of the FPGA tools using our benchmarks, helping us to solve performance issues caused by the kernel scheduler and document modifications of toolchain features over time. This shows the relevance of our benchmark suite not only for empirical performance evaluation but also for system maintenance.

We extended the benchmark suite's applicability beyond single FPGA boards by providing highly compatible implementations for multi-FPGA systems that leverage the host MPI installation for communication. These implementations can run on a wide range of multi-FPGA systems without being restricted to a specific inter-FPGA communication infrastructure. Additionally, we demonstrated the suite's extendability by providing optimized benchmark implementations for existing communication frameworks, dedicated inter-FPGA communication infrastructure, and other HLS runtimes like XRT. The evaluation showed that using QSFP ports for communication not only improves communication throughput and latency, and with that enhancing application performance, but can also lead to performance degradation because of the additional resource utilization of the network stacks. Our benchmark suite is the first multi-FPGA benchmark suite supporting the parallel execution of benchmarks over a complete multi-FPGA system, further establishing FPGA boards as first-class citizens in HPC.

Furthermore, we implemented and optimized two complex applications with first-class multi-FPGA support that show good scalability over complete multi-FPGA systems. Firstly, as part of the benchmark suite, we developed a parallel LU decomposition within our HPL benchmark. To our knowledge, it is the fastest LU decomposition implementation for FPGA, achieving over 48.7 TFLOP/s using single-precision floating-point values. Secondly, we ported a shallow water simulation on unstructured meshes, with high requirements for low-latency communication, to Xilinx FPGAs and extended it with support for the ACCL communication framework. Therefore, we used our benchmarks to evaluate different communication approaches based on ACCL and used our findings to select an apropiate approach to use within the application. Moreover, we refined the performance model of the application based on our benchmark results to accurately reflect the impact of communication latency on the application performance. Our implementation matches the performance per FPGA of the original implementation using SerialLite and a custom circuit-switched inter-FPGA network but overcomes the scalability limitations of the original version. In our evaluation using all 48 Xilinx FPGAs of Noctua 2, our simulation shows close to linear scaling in weak scaling scenarios and limited scalability in strong scaling scenarios, which can be precisely explained by our improved performance model considering the communication latency and number of communication peers per rank.

Communication latency plays an important role for the scaling performance of multi-FPGA applications especially for small to medium problem sizes. Our communication-latency models are application-independent, supporting the integration into other application performance models to analyze their scaling capabilities using the ACCL framework.

Our analysis of an existing communication framework for FPGA showed, that higher levels of abstraction offer improved code portability independent of the lower layers of the network stack while offering well optimized implementations for frequently used communication tasks. As shown by our implementation of the shallow water simulation, a lower level of abstraction is not always required to achieve the same performance with minimal resource overhead. This indicates that existing concepts for communication frameworks for FPGAs have the potential to accelerate the development of multi-FPGA applications in the future. However, currently communication frameworks themselves often only support a limited set of devices and network infrastructure. Also, many frameworks require the use of custom BSPs or shells instead of the default vendor shells, increasing the hurdles for a broad adoption of the frameworks. Nevertheless, because of the high compatability of our benchmark suite with various FPGA boards of the mayor vendors, it provides a promising foundation to support the further development of inter-FPGA communication frameworks.

7.2 Future Work

The benchmark suite achieved performance close to the expected bottlenecks for most benchmarks and near the theoretical peak, making it valuable for HPC acquisition planning, application optimization, and system maintenance. As discussed in Chapter 3 and Chapter 4, the benchmarks can detect changes in tool behavior. Their relatively simple nature simplifies debugging and detection of such changes because we provide performance models for every benchmark. For system maintenance, the benchmarks can be used with CI to continuously measure performance over time and across tool versions, identifying potential changes in tool behavior and device performance. This allows developers to adapt to changes with reduced debugging overhead for complex applications.

Future research should broaden the evaluation scope by assessing other communication frameworks with diverse applications featuring varying communication characteristics, particularly focusing on collective communication. Extending more benchmarks and multi-FPGA HPC applications to support these frameworks will provide a comprehensive understanding of the performance of communication approaches across a range of scenarios. Therefore the further support for additional and next-generation HLS tools, such as SYCL and Intel's oneAPI, are valuable extensions of the becnhmark suite.

Our analysis of existing communication frameworks found that both circuit-switched and packet-switched communication are advantageous depending on the scenario. However,

as summarized in Chapter 2, current efforts, especially for Xilinx/AMD-based FPGAs, are limited to packet-switched communication using complex and resource-heavy transport protocols. A lightweight framework for these devices, with support for collectives similar to SMI, would enhance their potential and allow better optimization of communication for the target application and network infrastructure. We already set the foundation for such a framework with the Aurora HLS project [29]. Another direction would be to integrate such a network stack into existing communication frameworks like ACCL to reduce the complexity for application developers to integrate new network stacks into their applications.

A current limitation of nearly all communication frameworks is the restriction to a single FPGA vendor or even a single FPGA board. Future work should also cover cross-vendor inter-FPGA communication – circuit-switched as well as packet-switched. Multi-vendor frameworks will further increase the portability of FPGA-accelerated HPC applications and enable the exploitation of architectural differences of the FPGAs like differences in DSPs, local memory, or logic resources to further optimize HPC workloads. Moreover, such a communication framework would facilitate the development of improved *base implementations* of the HPCC FPGA benchmarks that also utilize the QSFP ports of the boards.

A broadly compatible benchmark suite has huge potential to accelerate the development and adoption of these future works by offering reference implementations and alowing standardized performance comparisons. By addressing these future directions, we can further enhance the utilization and performance of FPGA-accelerated HPC systems, driving innovation and efficiency in FPGA-accelerated high-performance computing applications.

Glossary

ACCL Alveo Collective Communication Library
API Application Program Interface
BRAM Block RAM
BSP Board Support Package
CCLO Collective Communication Library Offload
CI Continuous Integration
CUDA Compute Unified Device Architecture
DP Double Precision
DSP Digital Signal Processor
EP Embarrassingly Parallel
ES Ethernet Switch
FLOP Floating Point Operation
FP Floating-Point
FPGA Field Programmable Gate Array
GPU Graphics Processing Unit
GUPS Giga Updates Per Second
HACC Heterogeneous Accelerated Compute Clusters
HBM2 High Bandwidth Memory 2
HDL Hardware Description Language
HLS High Level Synthesis
HP Half Precision
HPC High Performance Computing
HPCC High Performance Computing Challenge
HPCG High Performance Conjugate Gradients
HPL High Performance LINPACK
IAS Intel Acceleration Stack
IEC Intel External Channel

IF Infinity Fabric
II Initiation Interval
IP Intellectual Property
LSU Load Store Unit
LUT Look Up Table
MAC Media Access Control
MPI Message Passing Interface
NCCL Nvidia Collective Communications Library
NIC Network Interface Controller
OCS Optical Circuit Switch
OpenCL Open Compute Language
PC² Paderborn Center for Parallel Computing
PL Programmable Logic
QSFP Quad Small Form-factor Pluggable
RCCL ROCm Communication Collectives Library
RNG Random Number Generator
SDK Software Development Kit
SIMT Single Instruction, Multiple Threads
SLR Sliced Logic Region
SP Single Precision
SPMD Single Program Multiple Data
SVM Shared Virtual Memory
URAM Ultra RAM
VCSN Virtual Circuit Switching Network
XRT Xilinx Runtime

A

Shallow Water Simulation Mesh Data

Table A.1 and Table A.2 contain the relevant mesh properties that were used as input to our model in Section 5.2. For partitioned meshes, the values for the partition with the highest number of neighbors N_{max} and the largest number of $E_{send} + E_{recv}$ are reported.

Table A.1: Mesh properties of the meshes and partitioning used for the weak scaling experiments in Section 5.2.

#FPGAs	E_{total}	E_{core}	$E_{send} + E_{recv}$	D_{ext}	L_{pipe}	N_{max}
1	6784	6784	0	312	687	0
2	13568	6714	49	196	687	1
4	27136	6618	148	176	687	2
8	54272	6595	179	48	687	4
16	108544	6542	242	23	687	7
24	163840	6554	274	0	687	7
32	217088	6522	265	0	687	8
42	217088	4947	224	0	687	8
48	217088	4318	204	0	687	8
42	289792	6636	265	0	687	8
48	312320	6262	245	0	687	8
1	3392	3392	0	192	687	0
2	6784	3356	36	159	687	1
4	13568	3283	104	80	687	3
8	27136	3267	126	105	687	5
16	54272	3211	184	0	687	8
32	108544	3196	196	0	687	7

#FPGAs	E_{total}	E_{core}	$E_{send} + E_{recv}$	D_{ext}	L_{pipe}	N_{max}
42	163840	3687	215	0	687	9

Table A.2: Mesh properties of the meshes and partitioning used for the strong scaling experiments in Section 5.2.

#FPGAs	E_{total}	E_{core}	$E_{send} + E_{recv}$	D_{ext}	L_{pipe}	N_{max}
1	6784	6784	0	312	687	0
2	6784	3356	36	164	687	1
4	6784	1621	74	83	687	2
6	6784	1061	74	38	687	4
4	27136	6618	148	164	687	2
6	27136	4383	140	74	687	4
9	27136	2874	141	19	687	4
12	27136	2128	135	0	687	6
9	54272	5862	169	60	687	5
12	54272	4341	183	30	687	7
15	54272	3420	201	0	687	7
24	54272	2116	147	0	687	9
15	108544	7004	233	47	687	6
24	108544	4318	209	0	687	6
33	108544	3108	181	0	687	7
42	108544	2422	163	0	687	7
48	108544	2100	161	0	687	8

Author's List of Peer-reviewed Publications

[1] A. R. Tareen, M. Meyer, T. Kenter, and C. Plessl, "HiHiSpMV: Sparse matrix vector multiplication with hierarchical row reductions on FPGAs with high bandwidth memory," in *2024 IEEE 32st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2024, pp. 32–42. DOI: 10.1109/FCCM60383.2024.00014.

[21] C. Bauer, T. Kenter, M. Lass, L. Mazur, M. Meyer, H. Nitsche, H. Riebler, R. Schade, M. Schwarz, N. Winnwa, *et al.*, "Noctua2 supercomputer," *Journal of large-scale research facilities JLSRF*, vol. 9, no. 1, 2024. DOI: 10.17815/jlsrf-8-187.

[41] M. Meyer, T. Kenter, and C. Plessl, "Evaluating FPGA accelerator performance with a parameterized OpenCL adaptation of selected benchmarks of the HPCChallenge benchmark suite," in *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, GA, USA: IEEE, Nov. 2020, pp. 10–18. DOI: 10.1109/H2RC51942.2020.00007.

[42] M. Meyer, "Towards performance characterization of FPGAs in context of HPC using OpenCL benchmarks," in *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2021, pp. 1–2. DOI: 10.1145/3468044.3468058.

[43] M. Meyer, T. Kenter, and C. Plessl, "In-depth FPGA accelerator performance evaluation with single node benchmarks from the HPC challenge benchmark suite for Intel and Xilinx FPGAs using OpenCL," en, *Journal of Parallel and Distributed Computing*, vol. 160, pp. 79–89, Feb. 2022. DOI: 10.1016/j.jpdc.2021.10.007.

[49] M. Meyer, T. Kenter, and C. Plessl, "Multi-FPGA designs and scaling of HPC challenge benchmarks via MPI and circuit-switched inter-FPGA networks," en, *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 2, pp. 1–27, Jun. 2023. DOI: 10.1145/3576200.

[58] M. Meyer, T. Kenter, L. Petrica, K. O'Brien, M. Blott, and C. Plessl, "Optimizing communication for latency sensitive HPC applications on up to 48 FPGAs using ACCL,"

in *Euro-Par 2024: Parallel Processing*, J. Carretero, S. Shende, J. Garcia-Blas, I. Brandic, K. Olcoz, and M. Schreiber, Eds., Springer Nature Switzerland, 2024, pp. 121–136. DOI: 10.1007/978-3-031-69766-1_9.

Author's List of Published Software and Evaluation Artifacts

[2] M. Meyer, *HPCC_FPGA*, version v0.5.1, Oct. 2021. DOI: 10.5281/zenodo.5555971.

[44] M. Meyer, *HPCC FPGA evaluation data single-FPGA*, Jun. 2020. DOI: 10.5281/zenodo.3877164.

[50] M. Meyer, *HPCC FPGA evaluation data multi-FPGA*, Apr. 2021. DOI: 10.5281/zenodo.6226550.

[59] M. Meyer, G. Pape, T. Kenter, and D. Stansby, *pc2/HPCC_FPGA: Add XRT and ACCL support for Xilinx FPGAs*, version v0.6, Jul. 2024. DOI: 10.5281/zenodo.12655013.

[60] M. Meyer, *ACCL patch and microbenchmarks for Noctua 2*, Jul. 2024. DOI: 10.5281/zenodo.12658012.

Bibliography

[1] A. R. Tareen, M. Meyer, T. Kenter, and C. Plessl, "HiHiSpMV: Sparse matrix vector multiplication with hierarchical row reductions on FPGAs with high bandwidth memory," in *2024 IEEE 32st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2024, pp. 32–42. DOI: 10.1109/FCCM60383.2024.00014.

[2] M. Meyer, *HPCC_FPGA*, version v0.5.1, Oct. 2021. DOI: 10.5281/zenodo.5555971.

[3] *Alveo U280 data center accelerator card data sheet (ds963)*, English, version Version 1.7, AMD, Jun. 23, 2023, 2023-06-23.

[4] *Bittware 520N-MX data sheet*, English, version r3 v0, BittWare, Mar. 22, 2024. [Online]. Available: https://www.bittware.com/files/520N-MX_datasheet_r3v0.pdf, 2024-06-12.

[5] *F-tile ethernet intel® FPGA hard IP user guide*, English, version Version 24.1, Intel, Jan. 4, 2024, 2024-05-27.

[6] *Versal devices integrated 100g multirate ethernet MAC subsystem product guide (PG314)*, English, version Version 2.2, AMD, Jan. 29, 2024, 2024-03-27.

[7] Khronos OpenCL Working Group, *The OpenCL specification version 1.2*, A. Munshi, Ed., https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf, Accessed: 2021-02-09.

[8] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 409–420. DOI: 10.1109/SC.2016.34.

[9] T. Kenter, "Invited tutorial: OpenCL design flows for intel and xilinx FPGAs: Using common design patterns and dealing with vendor-specific differences," in *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*, VDE, 2019, pp. 1–8, ISBN: 978-3-8007-5045-0.

[10] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and*

Distributed Systems, vol. 32, no. 5, pp. 1014–1029, 2021. DOI: 10.1109/TPDS.2020.3039409.

[11] T. De Matteis, J. de Fine Licht, and T. Hoefler, "FBLAS: Streaming linear algebra on FPGA," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20, Atlanta, Georgia: IEEE Press, 2020, ISBN: 9781728199986. DOI: 10.1109/SC41405.2020.00063.

[12] X. Hao, M. Zhang, C. Sun, Z. Tao, H. Rong, Y. Zhang, L. He, E. Petit, W. Chen, and Y. Liang, "Lasa: Abstraction and specialization for productive and performant linear algebra on FPGAs," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, pp. 34–40. DOI: 10.1109/FCCM57271.2023.00013.

[13] *Vitis libraries*, English, version Version 2023.2, AMD, Dec. 20, 2023, 2023-12-20.

[14] "HLS libs." (), [Online]. Available: https://hlslibs.org (visited on 03/26/2024).

[15] J. de Fine Licht and T. Hoefler, "hlslib: Software engineering for hardware design," *arXiv:1910.04436*, 2019. DOI: 10.48550/arXiv.1910.04436.

[16] T. De Matteis, J. de Fine Licht, J. Beránek, and T. Hoefler, "Streaming message interface: High-performance distributed memory programming on reconfigurable hardware," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Denver, Colorado: Association for Computing Machinery, 2019, ISBN: 9781450362290. DOI: 10.1145/3295500.3356201.

[17] K. Kikuchi, N. Fujita, R. Kobayashi, and T. Boku, "Implementation and performance evaluation of collective communications using CIRCUS on multiple FPGAs," in *Proceedings of the HPC Asia 2023 Workshops*, ser. HPC Asia '23 Workshops, Raffles Blvd, Singapore: Association for Computing Machinery, 2023, pp. 15–23, ISBN: 9781450399890. DOI: 10.1145/3581576.3581602.

[18] N. Contini, B. Ramesh, K. Kandadi Suresh, T. Tran, B. Michalowicz, M. Abduljabbar, H. Subramoni, and D. Panda, "Enabling reconfigurable HPC through MPI-based inter-FPGA communication," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 477–487. DOI: 10.1145/3577193.3593720.

[19] Z. He, D. Korolija, and G. Alonso, "Easynet: 100 gbps network for HLS," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2021, pp. 197–203. DOI: 10.1109/FPL53798.2021.00040.

[20] Z. Xiao, R. D. Chamberlain, and A. M. Cabrera, "HLS portability from intel to xilinx: A case study," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–8. DOI: 10.1109/HPEC49654.2021.9622785.

[21] C. Bauer, T. Kenter, M. Lass, L. Mazur, M. Meyer, H. Nitsche, H. Riebler, R. Schade, M. Schwarz, N. Winnwa, *et al.*, "Noctua2 supercomputer," *Journal of large-scale research facilities JLSRF*, vol. 9, no. 1, 2024. DOI: 10.17815/jlsrf-8-187.

[22] "Heterogeneous accelerated compute cluster." (), [Online]. Available: https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html#infrastructure (visited on 03/26/2024).

[23] T. Boku, N. Fujita, R. Kobayashi, and O. Tatebe, "Cygnus - world first multihybrid accelerated cluster with GPU and FPGA coupling," in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP Workshops '22, , Bordeaux, France, Association for Computing Machinery, 2023, ISBN: 9781450394451. DOI: 10.1145/3547276.3548629.

[24] "NVIDIA®NVLink ™high-speed interconnect: Application performance," NVIDIA Corporation, White Paper, Nov. 2014, p. 19. [Online]. Available: https://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf.

[25] "Frontier." (), [Online]. Available: https://www.olcf.ornl.gov/frontier/ (visited on 05/28/2024).

[26] *Intel FPGA SDK for OpenCL pro edition: Programming guide*, English, version Version 22.4, Intel, Dec. 19, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683846/22-4/implementing-the-channels-extension.html, 2024-06-11.

[27] "Xup vitis network example (vnx)." (), [Online]. Available: https://github.com/Xilinx/xup_vitis_network_example (visited on 07/19/2023).

[28] "Alveolink." (), [Online]. Available: https://github.com/Xilinx/AlveoLink (visited on 08/27/2024).

[29] "Aurora HLS." (), [Online]. Available: https://github.com/pc2/Aurora-HLS (visited on 10/08/2024).

[30] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling flexible network FPGA clusters in a heterogeneous cloud data center," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 237–246. DOI: 10.1145/3020078.3021742.

[31] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, and P. Chow, "Galapagos: A full stack approach to FPGA integration in the cloud," *IEEE Micro*, vol. 38, no. 6, pp. 18–24, 2018. DOI: 10.1109/MM.2018.2877290.

[32] Z. He, D. Parravicini, L. Petrica, K. O'Brien, G. Alonso, and M. Blott, "ACCL: FPGA-accelerated collectives over 100 gbps TCP-IP," in *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2021, pp. 33–43. DOI: 10.1109/H2RC54759.2021.00009.

[33] T. Ueno and K. Sano, "VCSN: Virtual circuit-switching network for flexible and simple-to-operate communication in HPC FPGA cluster," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 2, Mar. 2023, ISSN: 1936-7406. DOI: 10.1145/3579848.

[34] N. Prakriya, Y. Chi, S. Basalama, L. Song, and J. Cong, "TAPA-CS: Enabling scalable accelerator design on distributed HBM-FPGAs," in *Proceedings of the 29th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 966–980. DOI: 10.1145/3620666.3651347.

[35] Z. He, D. Korolija, Y. Zhu, B. Ramhorst, T. Laan, L. Petrica, M. Blott, and G. Alonso, "ACCL+: An FPGA-based collective engine for distributed applications," *arXiv preprint arXiv:2312.11742*, 2023.

[36] Y. Chi, L. Guo, J. Lau, Y.-k. Choi, J. Wang, and J. Cong, "Extending high-level synthesis for task-parallel programs," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 204–213. DOI: 10.1109/FCCM51124.2021.00032.

[37] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency hls design on multi-die fpgas," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 81–92, ISBN: 9781450382182. DOI: 10.1145/3431920.3439289.

[38] T. Ueno, A. Koshiba, and K. Sano, "Virtual circuit-switching network with flexible topology for high-performance fpga cluster," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 41–48. DOI: 10.1109/ASAP52443.2021.00013.

[39] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, Sep. 2006, ISSN: 0163-5964. DOI: 10.1145/1186736.1186737. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737.

[40] J. J. Dongarra and P. Luszczek, "Introduction to the HPCChallenge benchmark suite," en, Defense Technical Information Center, Fort Belvoir, VA, Tech. Rep., Dec. 2004. DOI: 10.21236/ADA439315.

[41] M. Meyer, T. Kenter, and C. Plessl, "Evaluating FPGA accelerator performance with a parameterized OpenCL adaptation of selected benchmarks of the HPCChallenge benchmark suite," in *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, GA, USA: IEEE, Nov. 2020, pp. 10–18. DOI: 10.1109/H2RC51942.2020.00007.

[42] M. Meyer, "Towards performance characterization of FPGAs in context of HPC using OpenCL benchmarks," in *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2021, pp. 1–2. DOI: 10.1145/3468044.3468058.

[43] M. Meyer, T. Kenter, and C. Plessl, "In-depth FPGA accelerator performance evaluation with single node benchmarks from the HPC challenge benchmark suite for Intel and Xilinx FPGAs using OpenCL," en, *Journal of Parallel and Distributed Computing*, vol. 160, pp. 79–89, Feb. 2022. DOI: 10.1016/j.jpdc.2021.10.007.

[44] M. Meyer, *HPCC FPGA evaluation data single-FPGA*, Jun. 2020. DOI: 10.5281/zenodo.3877164.

[45] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," 1995.

[46] "Jacamar CI." (), [Online]. Available: https://gitlab.com/ecp-ci/jacamar-ci (visited on 09/30/2024).

[47] P. Gorlani, T. Kenter, and C. Plessl, "OpenCL implementation of cannon's matrix multiplication algorithm on intel stratix 10 FPGAs," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 99–107. DOI: 10.1109/ICFPT47387.2019.00020.

[48] E. Luebbers, S. Liu, and M. Chu, *Simplify software integration for FPGA accelerators with OPAE*, White Paper, Intel, 2019. [Online]. Available: https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf (visited on 06/05/2020).

[49] M. Meyer, T. Kenter, and C. Plessl, "Multi-FPGA designs and scaling of HPC challenge benchmarks via MPI and circuit-switched inter-FPGA networks," en, *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 2, pp. 1–27, Jun. 2023. DOI: 10.1145/3576200.

[50] M. Meyer, *HPCC FPGA evaluation data multi-FPGA*, Apr. 2021. DOI: 10.5281/zenodo.6226550.

[51] R. Rabenseifner, *Effective bandwidth (b_eff) benchmark*, G. Schulz, Ed., https://fs.hlrs.de/projects/par/mpi//b{_}eff/, Accessed: 2021-03-09.

[52] *Bittware OpenCL s10 BSP reference guide*, Rev. 1.3, Feb. 2020.

[53] J. Dongarra, P. Luszczek, and Y. Tsai, *HPL-AI mixed-precision benchmark*, https://icl.bitbucket.io/hpl-ai/, Accessed: 2021-03-10.

[54] J. J. Dongarra, S. Hammarling, and D. W. Walker, "Key concepts for parallel out-of-core LU factorization," *Parallel Computing*, vol. 23, no. 1-2, pp. 49–70, 1997.

[55] J. Faj, T. Kenter, S. Faghih-Naini, C. Plessl, and V. Aizinger, "Scalable multi-FPGA design of a discontinuous galerkin shallow-water model on unstructured meshes," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2023, pp. 1–12. DOI: 10.1145/3592979.3593407.

[56] J. Menzel, C. Plessl, and T. Kenter, "The strong scaling advantage of FPGAs in HPC for n-body simulations," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 1, Nov. 2021, ISSN: 1936-7406. DOI: 10.1145/3491235.

[57] M. Büttner, C. Alt, T. Kenter, H. Köstler, C. Plessl, and V. Aizinger, "Enabling performance portability for shallow water equations on cpus, gpus, and fpgas with sycl," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '24, Zurich, Switzerland: Association for Computing Machinery, 2024. DOI: 10.1145/3659914.3659925.

[58] M. Meyer, T. Kenter, L. Petrica, K. O'Brien, M. Blott, and C. Plessl, "Optimizing communication for latency sensitive HPC applications on up to 48 FPGAs using ACCL," in *Euro-Par 2024: Parallel Processing*, J. Carretero, S. Shende, J. Garcia-Blas, I. Brandic, K. Olcoz, and M. Schreiber, Eds., Springer Nature Switzerland, 2024, pp. 121–136. DOI: 10.1007/978-3-031-69766-1_9.

[59] M. Meyer, G. Pape, T. Kenter, and D. Stansby, *pc2/HPCC_FPGA: Add XRT and ACCL support for Xilinx FPGAs*, version v0.6, Jul. 2024. DOI: 10.5281/zenodo.12655013.

[60] M. Meyer, *ACCL patch and microbenchmarks for Noctua 2*, Jul. 2024. DOI: 10.5281/zenodo.12658012.

[61] T. Kenter, A. Shambhu, S. Faghih-Naini, and V. Aizinger, "Algorithm-hardware co-design of a discontinuous galerkin shallow-water model for a dataflow architecture on FPGA," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2021, pp. 1–11. DOI: 10.1145/3468267.3470617.

[62] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Ieee, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[63] W.-c. Feng, H. Lin, T. Scogland, and J. Zhang, "OpenCL and the 13 dwarfs: A work in progress," in *Proceedings of the International Conference on Performance Engineering (ICPE)*, ser. ICPE '12, New York, NY, USA: Association for Computing Machinery, 2012, pp. 291–294. DOI: 10.1145/2188286.2188341.

[64] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2010, pp. 63–74, ISBN: 9781605589350. DOI: 10.1145/1735688.1735702.

[65] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2016, pp. 409–420. DOI: 10.1109/SC.2016.34.

[66] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A realistic high-level synthesis benchmark suite for software-programmable FPGAs," *International Symposium on Field-Programmable Gate Arrays (FPGA)*, Feb. 2018. DOI: 10.1145/3174243.3174255.

[67] G. Ndu, J. Navaridas, and M. Luján, "CHO: Towards a benchmark suite for OpenCL FPGA accelerators," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15, Palo Alto, California: Association for Computing Machinery, 2015, ISBN: 9781450334846. DOI: 10.1145/2791321.2791331.

[68] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An OpenCL FPGA benchmark suite," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec. 2016, pp. 141–148. DOI: 10.1109/FPT.2016.7929519.

[69] E. Calore and S. F. Schifano, "Performance assessment of FPGAs as HPC accelerators using the FPGA empirical roofline," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2021, pp. 83–90. DOI: 10.1109/FPL53798.2021.00022.

[70] E. Calore and S. F. Schifano, "FER: A benchmark for the roofline analysis of FPGA based HPC accelerators," *IEEE Access*, vol. 10, pp. 94 220–94 234, 2022. DOI: 10.1109/ACCESS.2022.3203566.

[71] T. Nguyen, C. MacLean, M. Siracusa, D. Doerfler, N. J. Wright, and S. Williams, "FPGA-based HPC accelerators: An evaluation on performance and energy efficiency," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, e6570, 2022. DOI: 10.1002/cpe.6570.

[72] A. Zeni, K. O'Brien, M. Blott, and M. D. Santambrogio, "Optimized implementation of the HPCG benchmark on reconfigurable hardware," in *European Conference on Parallel Processing*, Springer, 2021, pp. 616–630. DOI: 10.1007/978-3-030-85665-6_38.

[73] T. Hauser, A. Dasu, A. Sudarsanam, and S. Young, "Performance of a LU decomposition on a multi-FPGA system compared to a low power commodity microprocessor system.," *Scalable Computing: Practice and Experience*, vol. 8, no. 4, pp. 373–385, Jan. 2007, ISSN: 1895-1767.

[74] G. Wu, Y. Dou, J. Sun, and G. D. Peterson, "A high performance and memory efficient LU decomposer on FPGAs," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 366–378, 2012. DOI: 10.1109/TC.2010.278.

[75] M. K. Kumar, Z. Choudry, and S. Purini, "Accelerating LU-decomposition of arbitrarily sized matrices on FPGAs," in *2023 International VLSI Symposium on Technology, Systems and Applications (VLSI-TSA/VLSI-DAT)*, 2023, pp. 1–4. DOI: 10.1109/VLSI-TSA/VLSI-DAT57221.2023.10134072.

[76] M. K. Jaiswal and N. Chandrachoodan, "FPGA-based high-performance and scalable block LU decomposition architecture," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 60–72, 2012. DOI: 10.1109/TC.2011.24.

[77] K. Turkington, K. Masselos, G. A. Constantinides, and P. Leong, "FPGA based acceleration of the linpack benchmark: A high level code transformation approach," in *2006 International Conference on Field Programmable Logic and Applications (FPL)*, 2006, pp. 1–6. DOI: 10.1109/FPL.2006.311240.

[78] G. Wu, Y. Dou, Y. Lei, J. Zhou, M. Wang, and J. Jiang, "A fine-grained pipelined implementation of the LINPACK benchmark on FPGAs," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 183–190. DOI: 10.1109/FCCM.2009.11.

[79] E. Perdomo, A. Kropotov, F. K. Cano Ladino, S. Zafar, T. Cervero, X. M. Bofill, and B. Salami, "Makinote: An FPGA-based HW/SW platform for pre-silicon emulation of RISC-V designs," in *Proceedings of the 16th Workshop on Rapid Simulation and Performance Evaluation for Design*, ser. RAPIDO '24, Munich, Germany: Association for Computing Machinery, 2024, pp. 29–34. DOI: 10.1145/3642921.3642928.

[80] N. Fujita, R. Kobayashi, Y. Yamaguchi, T. Boku, K. Yoshikawa, M. Abe, and M. Umemura, "OpenCL-enabled parallel raytracing for astrophysical application on multiple FPGAs with optical links," in *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2020, pp. 48–55. DOI: 10.1109/H2RC51942.2020.00011.

[81] R. Kobayashi, N. Fujita, Y. Yamaguchi, T. Boku, K. Yoshikawa, M. Abe, and M. Umemura, "GPU–FPGA-accelerated radiative transfer simulation with inter-FPGA communication," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia '23, Singapore, Singapore: Association for Computing Machinery, 2023, pp. 117–125, ISBN: 9781450398053. DOI: 10.1145/3578178.3578231.

[82] J. Huthmann, A. Shin, A. Podobas, K. Sano, and H. Takizawa, "Scaling performance for n-body stream computation with a ring of FPGAs," in *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, ser. HEART '19, Nagasaki, Japan: Association for Computing Machinery, 2019, ISBN: 9781450372558. DOI: 10.1145/3337801.3337813.

[83] N. Sheng, Z. Tong, C. Jiang, X. Ma, X. Yang, H. Li, Z. Tian, and Q. Zhang, "Microsecond simulation in a special-purpose molecular dynamics computer cluster," in *Proceedings of the International Conference on Bioinformatics and Computational Biology (ICBCB)*, 2023, pp. 151–157. DOI: 10.1109/ICBCB57893.2023.10246549.

[84] T. Alonso, L. Petrica, M. Ruiz, J. Petri-Koenig, Y. Umuroglu, I. Stamelos, E. Koromilas, M. Blott, and K. Vissers, "Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 2, Dec. 2021. DOI: 10.1145/3470567.

[85] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Achieving super-linear speedup across multi-fpga for real-time dnn inference," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. DOI: 10.1145/3358192.