

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Masterarbeit

**Performance portability analysis of
SYCL with a classical CG on CPU,
GPU, and FPGA**

Julian Franquinet

Course of Study:	Simulation Technology
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	Alexander Van Craen, M.Sc.

Commenced:	October 20, 2022
Completed:	April 20, 2023

Abstract

In this work, the capability of SYCL[™] to execute code on different hardware devices is investigated. This motivates conducting a performance portability analysis. The architectures investigated are the CPU, GPU, and FPGA. As a benchmark algorithm, the CG algorithm is used, as it is widely applicable to many fields and is more complex than simple matrix-vector multiplications. To generate reference results on the different devices, OpenMP and CUDA are used. The CG is also implemented using highly optimized libraries. These libraries are based on the BLAS standard. The results show a significant increase in performance when using the libraries on the GPU for growing problem sizes. Regarding the CPU, the optimizations are more significant for smaller problem sizes. So far, optimized libraries for the FPGA do not exist and therefore are not investigated. As a result, the performance of the FPGA is not as good as on the CPU and GPU. This is why the portability performance analysis results in rather low performance portability. However, the results show that SYCL[™] is capable of executing code on various hardware devices, making it a promising standard for future applications.

Contents

1	Introduction	15
2	Hardware	17
2.1	Central Processing Unit	17
2.2	Graphics Processing Unit	19
2.3	Field-Programmable Gate Array	20
3	Software	23
3.1	Open Multi-Processing	23
3.2	Compute Unified Device Architecture	23
3.3	SYCL™	24
3.4	Basic Linear Algebra Subprograms	25
4	Conjugate Gradient Method	27
5	Performance Portability	29
6	Implementation	31
6.1	Linear Algebra Kernels	31
6.2	Conjugate Gradient Method	36
7	Results	39
7.1	Runtimes	39
7.2	Performance Portability Analysis	46
8	Conclusion and Outlook	49
	Bibliography	51

List of Figures

2.1	Scheme of the architecture of a Central Processing Unit	18
2.2	Scheme of the architecture of a Graphics Processing Unit	20
2.3	Scheme of the architecture of a Field-Programmable Gate Array	21
6.1	Memory access of a matrix-vector multiplication kernel on the Graphics Processing Unit.	32
6.2	Memory access of a vector addition kernel on the Graphics Processing Unit.	34
6.3	Memory access of a vector reduction kernel on the Graphics Processing Unit.	35
7.1	Runtimes of representative linear algebra kernels over dimension N	40
7.2	Runtimes of representative linear algebra kernels over dimension N	43
7.3	Runtimes of the Conjugate Gradient Method implementations on the Central Processing Unit over problem size N	44
7.4	Runtimes of the Conjugate Gradient Method implementations on the Graphics Processing Unit over problem size N	45
7.5	Runtimes of Conjugate Gradient Method over problem size N	45
7.6	Pseudo-efficiencies of Conjugate Gradient Method implementations over problem size N	46

List of Tables

2.1	Key specifications of the INTEL® Xeon®Gold 6128	18
2.2	Key specifications of the NVIDIA Quadro GP100	19
2.3	Key specifications of the INTEL® Arria®10 GX 1150	21
6.1	Natively implemented linear algebra kernels	37
6.2	Used Basic Linear Algebra Subprograms functions for the Conjugate Gradient Method	37
7.1	Fastest achieved averaged runtimes of the CG implementations on each device for $N = 4,096$	46
7.2	Ideal amount of floating point operations for the Conjugate Gradient Method	47
7.3	Runtimes and efficiencies of the Conjugate Gradient Method implementations	47
7.4	Results of the performance portability analysis for the Conjugate Gradient Method with $N = 4,096$	48

List of Algorithms

6.1	Matrix-vector multiplication kernel for Central Processing Unit and Field-Programmable Gate Array	32
6.2	Matrix-vector multiplication kernel for Graphics Processing Unit	33
6.3	Vector addition kernel for Central Processing Unit and Field-Programmable Gate Array	33
6.4	Vector addition kernel for Graphics Processing Unit	34
6.5	Vector reduction kernel for Central Processing Unit and Field-Programmable Gate Array	35
6.6	Vector reduction kernel for Graphics Processing Unit	36
6.7	Conjugate Gradient Method algorithm	38

Acronyms

ALU	Arithmetic Logic Unit.	17
API	Application Programming Interface.	23
BLAS	Basic Linear Algebra Subprograms.	25
CG	Conjugate Gradient Method.	15
CLB	Configurable Logic Block.	20
CPU	Central Processing Unit.	15
CU	Control Unit.	17
cuBLAS	CUDA Basic Linear Algebra Subroutine.	16
CUDA	Compute Unified Device Architecture.	15
DRAM	Dynamic Random Access Memory.	17
FLOPS	Floating Point Operations Per Second.	17
FPGA	Field-Programmable Gate Array.	15
GPU	Graphics Processing Unit.	15
L1	Level 1 Cache.	17
L2	Level 2 Cache.	17
L3	Level 3 Cache.	17
MMU	Memory Management Unit.	17
oneMKL	INTEL® oneAPI Math Kernel Library.	16
OpenCL	Open Computing Language.	15
OpenMP	Open Multi-Processing.	23
PI	Programable Interconnect.	20

Variables

Symbol	Description
$\tilde{\mathbf{A}}$	Square, symmetric, and positive-definite matrix
\mathbf{A}	Matrix
α	Scalar
$\tilde{\alpha}$	Step size of incremental solution vector update
\mathbf{b}	Known vector
β	Scalar
$\tilde{\beta}$	Gram-Schmidt Conjugation coefficient
\mathbf{d}	Orthogonal search direction
δ	Squared residual norm
e	Efficiency
ε	Abortion criterion for the iterative solver
o	Amount of operations
P	Set of platforms
p	Platform
\mathcal{P}	Performance portability
ϕ	Theoretical peak performance
Q	Set of all programs
q	Program
\mathbf{q}	Temporary vector
$\tilde{\mathbf{r}}$	Residual
t	Runtime
\mathbf{x}	Vector
$\tilde{\mathbf{x}}$	Unknown solution vector
\mathbf{y}	Vector

1 Introduction

In recent years, the amount of computational load in many fields has increased significantly. Especially with massive progress in computational fluid dynamics, machine learning, and many other fields, the demand for more computational power is growing. Particularly in times of climate crisis, not just the computational time but also the energy consumption is important. Therefore, it is crucial to find a balance between the two. One method is to use different hardware devices according to their specific use cases in order to maximize time and energy efficiency. In the past, nearly all scientific calculations were performed on the Central Processing Unit (CPU). The focus was mainly on designing algorithms that can be executed on many CPUs in parallel. In recent years, the Graphics Processing Unit (GPU) gained more and more importance, adding a new dimension to parallel computing. The GPU is designed for highly parallel computing and is best suited for such tasks. For example, NVIDIA provides Compute Unified Device Architecture (CUDA), a parallel computing framework that also delivers an application programming interface. This architecture can be used to write code that can be executed on all NVIDIA GPUs and results in a large increase in performance and efficiency of many applications. However, there are limitations. Not all tasks are suited for the GPU. Also, since CUDA runs only on NVIDIA GPUs, it creates a huge dependency on one vendor.

Another hardware device that is becoming increasingly important is the Field-Programmable Gate Array (FPGA). The FPGA is a hardware device that can be programmed to execute a specific task. This makes it very flexible and could allow an increase in performance and efficiency for many applications. While the FPGA is not currently widespread, it is expected to gain more importance in the future, particularly with the release of SYCL™. Until the release of SYCL™, Open Computing Language (OpenCL) was the only standard that could be used to write code that can be executed on FPGAs. However, the complexity of OpenCL makes it hard to develop applications for the FPGA. SYCL™ was developed to solve this problem.

With all these different hardware devices, a new challenge arises. How can a program be designed that is executable on all of them? This is where SYCL™ could provide a solution. It not only delivers a standard for programming on the FPGA, but also for programming on the CPU and GPU. This allows the programmer to write code that can be executed on all devices without rewriting the whole code.

This work investigates the capability of SYCL™ to execute code on different hardware devices. The Conjugate Gradient Method (CG) algorithm is used as a benchmark, as it is widely applicable to many fields. Runtimes are investigated and compared to reference implementations using different methods. Additionally, a performance portability analysis is performed. This work is following up on BARATTA ET AL [1] who investigated the performance portability of SYCL™ on the CPU and GPU. Also, this thesis contains the execution of the CG algorithm on the FPGA. CALI ET AL [3] has already conducted this approach, but dealt with portability issues. Therefore, no runtimes were provided.

As this work will not focus on optimizing the CG algorithm, highly optimized libraries will be used as well. The rather simple, non-optimized native implementations will then be compared to the highly optimized libraries. This allows measurement of the impact of the libraries on the performance of algorithms. Both NVIDIA and INTEL® provide highly optimized libraries with CUDA Basic Linear Algebra Subroutine (cuBLAS) and INTEL® oneAPI Math Kernel Library (oneMKL) respectively. In the work of KHALILOV & TIMOVEEV [11] a performance analysis of cuBLAS has already been performed. However, only simple matrix-vector multiplications were used. KRAINIUK ET AL [12] did the same for oneMKL, but also for rather simple algorithms. This work will use more complex algorithms with the CG.

In the following work, the hardware devices and libraries that were used are introduced first. Then, the CG algorithm is explained and a metric for the performance portability is introduced. Afterward, the implementations are explained and the results are presented. Finally, the results are discussed and conclusions are drawn.

2 Hardware

In modern computing clusters and programs, different hardware components are used. Each component has its purpose and is designed to perform a specific task. In this chapter, the different hardware components are introduced with a representative of each architecture that is used in this work. For comparison, the key specifications of the representatives of each architecture used in this work are listed. The Floating Point Operations Per Second (FLOPS) is a measurement that describes the theoretical peak performance of each device. Since lower runtimes are not necessarily better, as efficiency is essential in many cases, the theoretical peak performance is important to take into count.

2.1 Central Processing Unit

The Central Processing Unit (CPU) is often referred to as the brain of a computer. It is the device's central processor and carries out most of the instructions of a program. It is also responsible for running the operating system. The CPU is designed to handle a wide range of tasks quickly but is limited to their concurrency. The main components of a CPU are the Control Units (CUs), Memory Management Units (MMUs), Arithmetic Logic Units (ALUs), and several levels of caches. Modern CPUs consist of multiple cores. Each core is a separate CPU consisting of a CU, MMU, ALU, and caches. The cores are connected to shared memory and can communicate with each other.

The CU is the component that is in charge of the execution of instructions. It is the component that fetches instructions from memory and decodes them. It is also responsible for the control flow of the program. The MMU is the component managing the CPU's memory. It translates virtual addresses to physical addresses, handles memory protection, and loads memory from the Dynamic Random Access Memory (DRAM). The component that takes care of the arithmetic and logical operations is the ALU. It is the component that performs the actual operations.

The caches are ultra-fast built-in memories partly shared between the cores to provide quick memory access. In Figure 2.1, a possible design of a CPU is shown. In this case, three levels of cache are used. The Level 1 Cache (L1) is the fastest and closest to the core. It is the smallest and has the least capacity. The Level 2 Cache (L2) is the next fastest, depending on the design, either exclusive or shared between the cores. In the case of Figure 2.1, it is exclusive to each core. The Level 3 Cache (L3) is the slowest and is shared between all cores. It is the largest and has the largest capacity.

Every core can only run a single process at a time. In order to increase the number of independent instructions per core, multiple virtual cores can be addressed for each physical core. This can increase the efficiency of each processor. If, for example, resources are not yet available for one process, the other process can continue if its resources are already available. In INTEL® chips, this is done by two virtual cores and is called Hyper-Threading. It is essential to mention that Hyper-Threading does not increase the computational power of a core but rather its efficiency.

When developing a high-performance program that runs on the CPU, it is crucial to be aware of multicore and Hyper-Threading. This means that the program has to be designed in a way that it can be split up into independent tasks that can be executed in parallel. Additionally, data access has to be taken into account. If data is shared between the tasks, it has to be synchronized, resulting in a performance loss.

A more detailed description of the architecture of a CPU can be found in the INTEL® developer manual [6].

The CPU used in this work is the INTEL® Xeon®Gold 6128. Specifications of it can be found on webpage [10]. The key specifications are shown in Table 2.1.

INTEL® Xeon®Gold 6128	
Frequency	3.4 GHz
Cores	6
Theoretical peak performance	0.7104 TFLOPS
DRAM	188 GB

Table 2.1: Listed key specifications of the INTEL® Xeon®Gold 6128 [10]

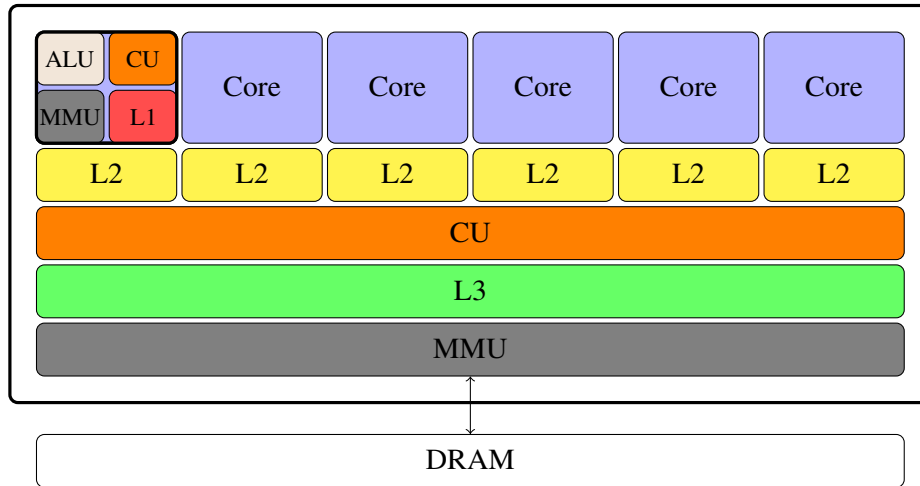


Figure 2.1: Depicted scheme of the architecture of a Central Processing Unit (CPU). This CPU consists of six cores. Each core has its own Level 2 Cache (L2) and Level 1 Cache (L1). Additionally, it contains an Arithmetic Logic Unit (ALU) for the actual calculation, a Control Unit (CU) for the orchestration of the execution of instructions, and a Memory Management Unit (MMU) for the management of the memory. All cores are connected to a shared Level 3 Cache (L3). The shared MMU communicates with the Dynamic Random Access Memory (DRAM). The shared Control Unit (CU) orchestrates the cores.

2.2 Graphics Processing Unit

First, only responsible for rendering images and videos, the Graphics Processing Unit (GPU) nowadays has a broader field of applications. Their design is based on the idea of parallel computing. Instead of processing a complex task serially like in a CPU, the idea of using a GPU is to break up the task into many subtasks and to run them in parallel. Therefore, GPUs consist of many more but smaller and less versatile cores than the CPU. This results in a smaller and more specific set of instructions with higher instruction throughput.

In general, GPUs are either an independent piece of hardware with their own memory or an integrated part that shares the memory with the CPU. Typically, integrated GPUs provide less performance since they are smaller and share resources with the CPU.

The GPU shown in Figure 2.2 consists of two levels of cache. The L2 is shared along all cores, whereas the L1 is shared along a group of cores. Such a group also consists of shared CUs.

For repetitive and highly-parallel computing tasks, such as machine learning or rendering tasks, GPUs are best suited. In order to achieve best performance, a program has to be designed in a certain way. Tasks should be split up into many independent tasks that can be executed in parallel. Like with the CPU, it is vital to consider data access, especially when using shared memory. This is because certain cores might share the same caches. Therefore, adapting the code based on awareness of the architecture of the GPU can increase performance. Nevertheless, in scientific programming not all tasks can be split into many independent tasks. Even if this is the case, another key problem with GPU programming is that all tasks in a so called warp must execute exactly the same instructions. If a warp divergence occurs, the different execution paths must be masked and sequentialized. In this case, the cores could be in idle, resulting in a performance loss. Therefore, the efficiency of the CPU can be higher than the efficiency of the GPU. Still, the GPU could be the better choice, due to its pure computational power.

In this work, the NVIDIA Quadro GP100 is used. It is a GPU that is designed for high-performance computing. The key specifications, like the theoretical peak performance, are shown in Table 2.2 and can also be found in the NVIDIA Quadro GP100 Data Sheet [15]. For a more detailed description of the architecture of a GPU, see the NVIDIA programming guide [14] and the NVIDIA whitepaper [16].

NVIDIA Quadro GP100	
Frequency	1304 MHz
Cores	3584
Theoretical peak performance	10.34 TFLOPS
DRAM	16 GB

Table 2.2: Listed key specifications of the NVIDIA Quadro GP100 [15]

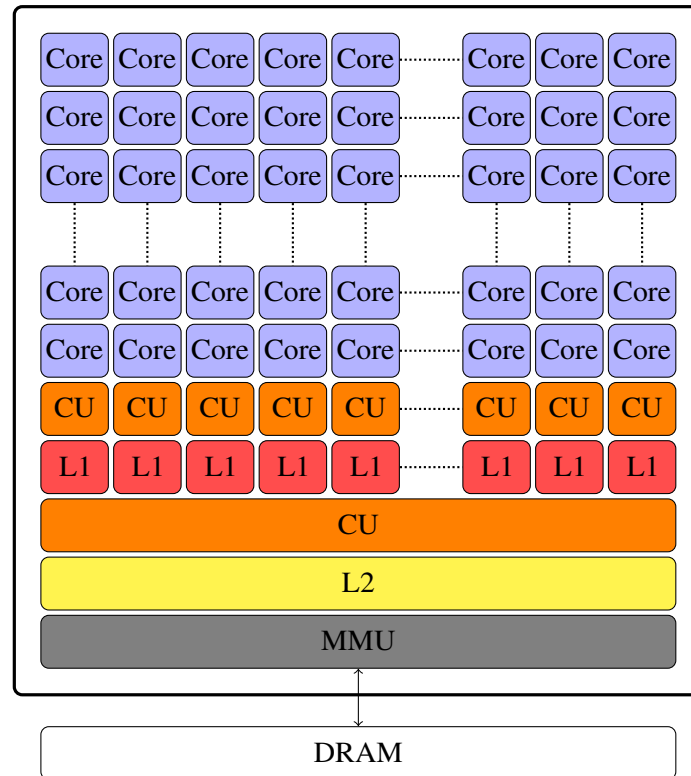


Figure 2.2: Depicted scheme of the architecture of a Graphics Processing Unit (GPU). This GPU consists of many smaller cores. A set of cores share a Level 1 Cache (L1) and a Control Unit (CU). Additionally, all sets of cores share a Level 2 Cache (L2). Also, a Control Unit (CU) is shared among all cores. The Control Unit (CU) orchestrates the cores. The Memory Management Unit (MMU) manages the memory.

2.3 Field-Programmable Gate Array

Field-Programmable Gate Arrays (FPGAs) are integrated circuits consisting of Configurable Logic Blocks (CLBs). The CLBs are connected via Programmable Interconnects (PIs). This setup provides a programmable hardware fabric that is highly flexible. High performance with low latency can be achieved by customizing the FPGA to perform specific tasks or functions. Similar to GPUs, FPGAs provide the ability to perform many calculations in parallel. Additionally, they are highly versatile, as they still offer a broad set of instructions.

Another advantage of FPGAs is their low power consumption compared to other processing units. Since FPGAs can be customized to implement only the necessary logic functions for a specific application, they can operate at lower voltages and consume less power than CPUs and GPUs, which have more general-purpose architectures.

Basically, the FPGA tries to combine the flexibility and efficiency of a CPU with the performance of a GPU. Resulting in an ideal computing unit for scientific computing. Although it is still in its early stages of development for scientific computing, the FPGA is a very promising technology. However, developing for FPGAs is not as easy as for CPUs and GPUs. The FPGA is a very complex

piece of hardware which is not easy to understand. Therefore, the development of FPGAs is more time consuming and requires more specialized knowledge. Additional compiling code can take several hours as the compiler must find the best possible hardware configuration for the given code. This is why FPGAs are not as widespread as CPUs and GPUs. Therefore, there is still a lack of documentation and example codes. Part of the motivation for this work is to address this problem.

In this work the INTEL® Arria®10 GX 1150 is used. It is a FPGA that is designed for high-performance computing. The key specifications are listed in Table 2.3. Additional specifications can be found in the INTEL® Arria®10 Product Sheet [8]. For a more detailed description of the architecture of a FPGA, refer to the INTEL® Arria®10 Datasheet [7].

INTEL® Arria®10 GX 1150	
CLBs	1150
Theoretical peak performance	1.366 TFLOPS
DRAM	157 GB

Table 2.3: Listed key specifications of the INTEL® Arria®10 GX 1150 [8]

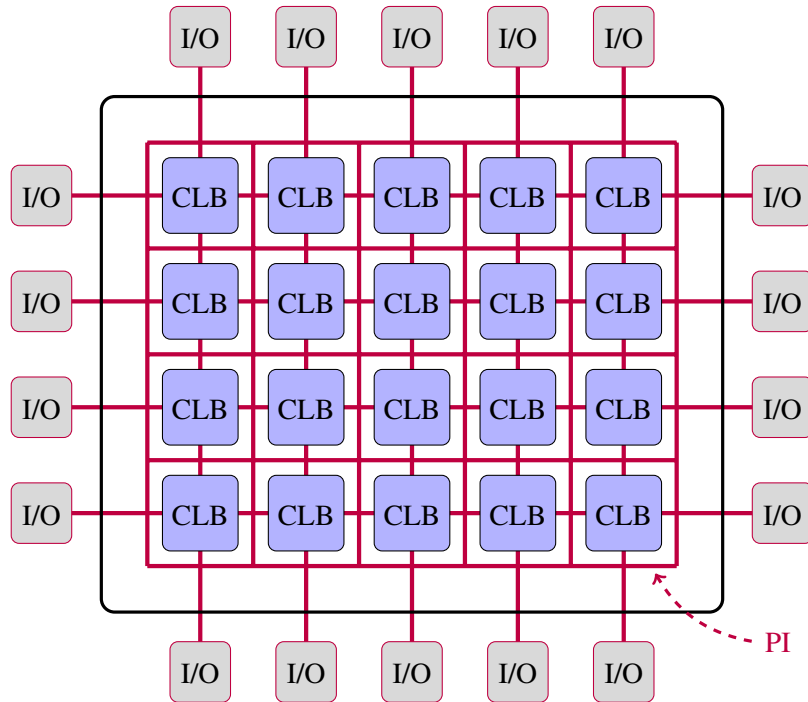


Figure 2.3: Depicted scheme of the architecture of a Field-Programmable Gate Array (FPGA). The Configurable Logic Blocks (CLBs) are connected through Programmable Interconnects (PIs). Communication and data transfer is accomplished via the Input-Output-Blocks (I/Os).

3 Software

In order to use the many different hardware devices, there are a wide range of software solutions. This chapter describes the different software solutions and explains how they can be used to develop parallel applications.

3.1 Open Multi-Processing

One of the most common ways to develop parallel applications is to use the Open Multi-Processing (OpenMP) Application Programming Interface (API). It enables parallel programming on shared memory architectures and is, therefore, suitable for running on multicore processors. By providing a set of directives, functions, and environment variables that can be used, parallelism in a program can be specified, allowing multiple threads to execute in parallel and share data. It enables incremental parallelization and is therefore an easy way to improve the performance of a program.

Compiler instructions can be used by linking the Open Multi-Processing (OpenMP) directives to the existing code, which indicates how to parallelize a specific section of code. Additionally, a header file can be included, providing a set of runtime library functions to manage threads and their synchronization.

OpenMP provides an easy and fast way to parallelize programs and increase efficiency. Algorithms can be parallelized by adding only a few lines of code. Additional command line options for the compiler are necessary to activate and allow interpretation of all OpenMP directives. Also, this parallelism can be achieved step by step, and therefore the parallelization can be done incrementally.

For further specification and a more detailed description refer to DAGUM & MENON [4] and the OpenMP reference guide [17]

3.2 Compute Unified Device Architecture

Traditionally, GPUs are used for graphics processing. However, because of their massively parallel processing ability, they can be used for many more applications. With the Compute Unified Device Architecture (CUDA), NVIDIA provided a parallel computing platform and programming model to allow the developer to use NVIDIA GPUs for general-purpose computing tasks. This program model is widely used and supported by many programming languages, including C, C++, and Fortran. Unfortunately, the usage of CUDA is limited to NVIDIA GPUs.

With just a minimal set of language extensions, the GPUs can be integrated into a workflow. The extensions consist of routines for memory management, allocating memory on the GPU and transferring data back and forth, defining and launching kernels, and synchronization.

In device programming with CUDA, the code is executed on the GPU and is therefore called a kernel. The kernel is launched by the host and is executed in parallel on the GPU. In order to organize kernels, the concept of threads and blocks is used. A thread is the smallest unit of parallel execution. Every thread is assigned to a unique identifier and can be executed in parallel along other threads, allowing computations to be performed simultaneously. A group of threads that are executed together are referred to as one block. Blocks can be organized into a one-, two- or three-dimensional grid. They also get assigned a unique identifier. The developer can specify the number of blocks and the number of threads inside a block, but these settings depend on the hardware, too. Inside a block, threads can be synchronized, and memory can be shared along threads, so the sizes should be adapted to the specific hardware to maximize performance.

The developer has to take care of the data management by himself and synchronize the GPU and the host. Memory is allocated and copied back and forth by explicit calls, meaning it is the programmer's responsibility that the data is available where it is needed. Unified shared memory can be used depending on the hardware, allowing data sharing between the GPU and the host. In this case, the data is copied automatically, meaning the programmer does not have to handle it. However, this is only possible if the GPU and the host share the same memory address space, which is only the case for some hardware. Also, the programmer still has to synchronize the GPU and the host.

A more detailed description of the CUDA API can be found in the CUDA reference guide by NVIDIA [14].

3.3 SYCL™

In order to generate high-performance and portable code that can be executed on a wide range of heterogeneous devices, the KHRONOS®GROUP developed SYCL™. It enables computational kernels to be written inside C++ source files as standard C++ code. Therefore, C++ features such as templating, generic programming, functional programming, and inheritance can be used while enabling heterogeneous multi-platform, multi-device execution. This allows the development of adaptable libraries with the capability of portable high performance. Using SYCL™ development takes place at a higher and more abstract layer than the native acceleration API. As this could limit the adaptivity of the code, SYCL™ still provides access to the lower-level code due to the seamless integration of the native acceleration API. However, this can limit the portability of the code.

Like in CUDA, kernels are defined and launched by the host. A kernel can be launched once or multiple times, depending on the use case. When launching a kernel multiple times, the amount of workers has to be specified. The SYCL™ runtime system takes care of the low-level details like the distribution of the work across the device and the synchronization of the host and the device.

In SYCL™, data access and storage are separated using buffers and accessors. This makes the manual managing and moving of storage unnecessary and removes the complexity of manually managing event dependencies between kernel instances. Runtime libraries track the movement of data and take care of correct behavior. Therefore, no explicit call is needed to move data between

the host and the device, and this is done automatically when needed. Also, SYCL™ provides the ability to use unified shared memory, which allows data sharing between the host and the device. When using this, no buffers are needed, and the data can be accessed directly. Still, this is not supported by all hardware.

In the case of kernel launches, the SYCL™ runtime system takes care of the low-level details of parallel execution, allowing developers to focus on the high-level logic of their computation. This allows the expression of parallel computations naturally and intuitively.

For a more detailed description, refer to the KHRONOS®GROUP SYCL™ specification [20].

3.4 Basic Linear Algebra Subprograms

Basic Linear Algebra Subprograms (BLAS) routines were introduced in order to perform basic linear algebra operations, such as matrix multiplication, vector addition, and reductions with high performance. There are bindings for C and Fortran and implementations to use it with Python and Matlab. This is why Basic Linear Algebra Subprograms (BLAS) routines are widely spread and used in scientific and engineering programming as well as fields like data analysis, machine learning, and computational fluid dynamics.

The routines are divided into three levels. Level one routines perform scalar, vector, and vector-vector operations. Level two provides implementations to perform matrix-vector multiplication, and consequently, level three provides results for matrix-matrix calculations. Refer to BLACKFORD ET AL. [2] for a more detailed description.

Both NVIDIA and INTEL® provide implementations of the BLAS routines for CUDA and SYCL™, respectively. cuBLAS is a library already shipped with CUDA and, therefore, easy to use. Only certain flags have to be set, and the respective header must be included. oneMKL is part of the INTEL® oneAPI project but can also be installed separately as an external library. For both implementations, it is referred to their respective reference guides [13] and [9].

4 Conjugate Gradient Method

For iteratively solving large systems of linear equations like

$$\tilde{\mathbf{A}} \tilde{\mathbf{x}} = \mathbf{b} \quad (4.1)$$

with $\tilde{\mathbf{A}}$ being a square, symmetric, and positive-definite matrix, \mathbf{b} any known vector, and $\tilde{\mathbf{x}}$ the unknown solution vector, the Conjugate Gradient Method (CG) is a very effective method. These systems can be found in a wide field of applications like machine learning, computational fluid dynamics, and more. This makes the CG an interesting and ideal example for this work to run on different hardware. A brief explanation of the iterative method can be found in the following. For a mathematical derivation as well as the convergence analysis of the method, refer to HESTENES & STIEFEL [5] and SHEWCHUK [19].

In every iterative step $\tilde{\mathbf{r}}_i$ of a problem like in Equation (4.1), the residual can be determined by

$$\tilde{\mathbf{r}}_i = \mathbf{b} - \tilde{\mathbf{A}} \tilde{\mathbf{x}}_i. \quad (4.2)$$

As first $\tilde{\mathbf{A}}$ -orthogonal search direction \mathbf{d}_0 the initial residual $\tilde{\mathbf{r}}_0$ can be used, resulting in

$$\mathbf{d}_0 = \tilde{\mathbf{r}}_0. \quad (4.3)$$

The incremental update to determine the solution vector $\tilde{\mathbf{x}}_{i+1}$ is performed as

$$\tilde{\mathbf{x}}_{i+1} = \tilde{\mathbf{x}}_i + \tilde{\alpha}_i \mathbf{d}_i, \quad (4.4)$$

with the step size $\tilde{\alpha}_i$ defined as

$$\tilde{\alpha}_i = \frac{\tilde{\mathbf{r}}_i^\top \tilde{\mathbf{r}}_i}{\mathbf{d}_i^\top \tilde{\mathbf{A}} \mathbf{d}_i}. \quad (4.5)$$

As the matrix-vector multiplication of $\tilde{\mathbf{A}}$ and \mathbf{d}_i is already performed in Equation (4.5), the result can be stored in a temporary vector \mathbf{q} . With that, the updated residual $\tilde{\mathbf{r}}_{i+1}$ can also be calculated as

$$\tilde{\mathbf{r}}_{i+1} = \tilde{\mathbf{r}}_i - \tilde{\alpha}_i \tilde{\mathbf{A}} \mathbf{d}_i = \tilde{\mathbf{r}}_i - \tilde{\alpha}_i \mathbf{q}, \quad (4.6)$$

with the advantage of needing less computational power. In order to find another $\tilde{\mathbf{A}}$ -orthogonal search direction \mathbf{d}_{i+1} , the Gram-Schmidt Conjugation can be used. The Gram-Schmidt Conjugation coefficient $\tilde{\beta}_{i+1}$ is defined as

$$\tilde{\beta}_{i+1} = \frac{\tilde{\mathbf{r}}_{i+1}^\top \tilde{\mathbf{r}}_{i+1}}{\tilde{\mathbf{r}}_i^\top \tilde{\mathbf{r}}_i}. \quad (4.7)$$

A new $\tilde{\mathbf{A}}$ -orthogonal search direction can be determined as

$$\mathbf{d}_{i+1} = \tilde{\mathbf{r}}_{i+1} - \tilde{\beta}_{i+1} \mathbf{d}_i. \quad (4.8)$$

With this new search direction \mathbf{d}_{i+1} , the next iteration can be started.

Often, it is not the exact solution that is needed, but a solution with a certain accuracy. Therefore, the loop can be aborted if the residual $\tilde{\mathbf{r}}_i$ is small enough or another abortion criterion is met.

5 Performance Portability

With the variety of different architectures, the performance of the algorithms can vary greatly. In this chapter, a metric is defined to determine the portability of the performance of programs. This metric was first introduced by PENNYCOOK ET AL. [18].

In the first step, a set of devices P is defined. This set contains all hardware platforms p that are considered for the performance portability analysis. Additionally, a set of programs Q is defined. This set contains all programs q that are considered for the performance portability analysis. With that, the performance will be measured on each platform and each program in this set. Using the runtimes and the number of operations, two efficiencies can be calculated for each combination of platform and program. The first efficiency is the application efficiency $e_{\text{App. Eff.}}$, which is the ratio of the program's runtime on a specific platform $t(p, q)$ and the best-observed runtime on that architecture. The best-observed runtime is the minimum of all runtimes of all programs on the platform.

$$e_{\text{App. Eff.}}(p, q) = \frac{\min_{q \in Q} t(p, q)}{t(p, q)} \quad (5.1)$$

The second efficiency is the architectural efficiency $e_{\text{Arch. Eff.}}$, calculated as the ratio of achieved FLOPS to the theoretical peak performance of the device $\phi(p)$. The achieved FLOPSs are calculated as the ideal number of operations o_{ideal} divided by the program's runtime on the device.

$$e_{\text{Arch. Eff.}}(p, q) = \frac{o_{\text{ideal}}}{t(p, q) \cdot \phi(p)} \quad (5.2)$$

Using the ideal number of operations ensures that every hardware-specific operation is penalized and that the hardware-specific optimizations do not benefit architectural efficiency.

The overall performance portability is then calculated as the harmonic mean of the respective efficiency resulting in the following equations:

$$\mathcal{P}_{\text{App. Eff.}}(P, q) = \begin{cases} \frac{|P|}{\sum_{p \in P} \frac{1}{e_{\text{App. Eff.}}(p, q)}}, & \text{if } q \text{ is supported } \forall p \in P \\ 0, & \text{otherwise} \end{cases} \quad (5.3)$$

and

$$\mathcal{P}_{\text{Arch. Eff.}}(P, q) = \begin{cases} \frac{|P|}{\sum_{p \in P} \frac{1}{e_{\text{Arch. Eff.}}(p, q)}}, & \text{if } q \text{ is supported } \forall p \in P \\ 0, & \text{otherwise} \end{cases} \quad (5.4)$$

6 Implementation

This chapter describes the implementation of the CG on the hardware devices of Chapter 2. Before native implementing a CG, different kernels were implemented, performing various basic linear algebra operations. These kernels were then used to implement the CG on the different devices. The implementation of the CG on the GPU was done with the CUDA programming language and SYCL™. For the CPU, the OpenMP programming language was used, as well as SYCL™ and a non-optimized sequential implementation. For running on FPGAs, only SYCL™ provided the capability. C++ was used as the underlying programming language for all implementations. Data management and synchronization between the different devices is not shown in the following pseudo-codes. They depend on the APIs used and are not relevant for this chapter.

6.1 Linear Algebra Kernels

The basic linear algebra operation needed for a CG can be divided into three underlying ideas: Matrix-Vector operation, vector addition, and vector reduction by the dot product. In the following, a sequential implementation of the kernels is described, that will be used as a basis for the parallel implementations. Extending the sequential implementation by OpenMP compiler instructions provides a parallel implementation for the CPU. Additionally, a second kernel is described, splitting the task of the first kernel into subtasks, which can be assigned to different threads. This kernel is used for the GPU implementation with CUDA and SYCL™, as well as the implementation on the CPU with SYCL™. With the FPGA optimized by the compilation process, the sequential implementation is used, letting the FPGA compiler decide on the parallelization.

In all cases, optimization strategies are ignored on purpose, as this is not the purpose of this work and can lead to hardware specific implementations. With cuBLAS and oneMKL highly optimized implementations already exist. Therefore, these libraries are used as well, so efficiency of such libraries can be compared to the simple implementations without optimizations.

6.1.1 Matrix-Vector Operation

A universal matrix-vector operation can be defined as

$$\alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y} = \mathbf{r}, \quad (6.1)$$

where \mathbf{A} represents a matrix of size $M \times N$, \mathbf{x} and \mathbf{y} are vectors of size N , and α and β are scalars. The result, vector \mathbf{r} , is of size N .

In a sequential setting, the kernel for any matrix-vector operation, like in Equation (6.1), consists of two for-loops to perform the operation. This is shown in Algorithm 6.1. In order to reduce the number of access operations, separate kernels can be implemented depending on whether scaling with scalars or additional vector adding is needed. This task can be dynamically shared along the CPU threads by including the OpenMP compiler instruction above the first for-loop.

Algorithm 6.1 Here the resulting kernel for a matrix-vector multiplication for the CPU and FPGA is displayed. The vectors are of size N and the matrix of size $M \times N$. The OpenMP compiler instruction can be included optionally when running on the CPU, to dynamically subdivide the loop into subtasks assigned to different threads.

```

procedure MatrixVectorOperation(A, x, y,  $\alpha$ ,  $\beta$ , r,  $N$ ,  $M$ )
  (#pragma omp parallel for)                                     // OpenMP parallelization
  for  $i = 1, 2, \dots, N$  do                                       // Iterate over rows of A
     $temp \leftarrow 0$ 
    for  $j = 1, 2, \dots, M$  do                                     // Iterate over columns of A
       $temp \leftarrow temp + \alpha \cdot A[i][j] \cdot x[j]$ 
    end for
     $r[i] \leftarrow temp + \beta \cdot y[i]$ 
  end for
  return r
end procedure

```

When designing for a high-multiprocessing device like a GPU, a fixed task decomposition can be achieved by assigning one row to each thread. Data access is shown in Figure 6.1, where different colors represent different threads. The resulting kernel is shown in Algorithm 6.2 and has to be launched with at least N amount of threads.

$$\begin{array}{c} \alpha \end{array} \begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,n} \\ A_{2,1} & A_{2,2} & \dots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \dots & A_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{array}{c} \beta \end{array} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{pmatrix}$$

Figure 6.1: Depicted here is a memory access of a matrix-vector multiplication kernel on the GPU. The matrix is of size $M \times N$ and the vectors of size N . Each thread is represented by one color. It accesses one row of the matrix, the whole \mathbf{x} vector, and one element of the vector \mathbf{y} . The scalars are shared among all threads.

Algorithm 6.2 The resulting kernel for a matrix-vector multiplication for the GPU is displayed in this pseudo-code. The vectors are of size N and the matrix of size $M \times N$. The kernel is launched with at least N amount of threads.

```
procedure MatrixVectorOperation( $\mathbf{A}, \mathbf{x}, \mathbf{y}, \alpha, \beta, \mathbf{r}, N, M$ )  
   $row \leftarrow \text{getGlobalID}()$   
  if  $row < N$  then                                // Check if current thread is within the bounds of the matrix  
     $temp \leftarrow 0$   
    for  $i = 1, 2, \dots, M$  do                          // Iterate over columns of  $\mathbf{A}$   
       $temp \leftarrow temp + \alpha \cdot A[row][i] \cdot x[i]$   
    end for  
     $r[row] \leftarrow temp + \beta \cdot y[row]$   
  end if  
  return  $\mathbf{r}$   
end procedure
```

6.1.2 Vector Addition

A general vector addition can be written as

$$\alpha \mathbf{x} + \beta \mathbf{y} = \mathbf{r}, \quad (6.2)$$

with \mathbf{x} and \mathbf{y} being vectors of size N , α and β being scalars, and \mathbf{r} the resulting vector of size N .

A vector operation, as in Equation (6.2), only needs one for-loop in a sequential setting. Algorithm 6.3 can be again extended by an OpenMP compiler instruction to dynamically subdivide the loop into subtasks assigned to different threads taking care of a part of the loop.

Algorithm 6.3 Here the resulting kernel for a vector addition for the CPU and FPGA is displayed. The vectors are of size N . The OpenMP compiler instruction is optional when running on the CPU, to dynamically subdivide the loop into subtasks assigned to different threads.

```
procedure VectorVectorOperation( $\mathbf{x}, \mathbf{y}, \alpha, \beta, \mathbf{r}, N$ )  
  (#pragma omp parallel for)                                // OpenMP parallelization  
  for  $i = 1, 2, \dots, N$  do                                    // Iterate over rows  
     $r[i] \leftarrow \alpha \cdot x[i] + \beta \cdot y[i]$   
  end for  
  return  $\mathbf{r}$   
end procedure
```

Like in the matrix-vector setup, the sequential code can be divided into subtasks by assigning one row to each thread. The data access, as shown in Figure 6.2, and the number of operations, can be again reduced by defining multiple kernels with the template of Algorithm 6.4 but leaving out operations that are not needed. As before, this kernel has to be launched with at least N threads.

$$\alpha \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \beta \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{pmatrix}$$

Figure 6.2: Here, memory access of a vector addition kernel on the GPU is depicted. The vectors are of size N . Each thread is represented by one color and accesses one element of each vector while the scalars are shared.

Algorithm 6.4 The resulting kernel for a vector addition for the GPU is displayed in this pseudo-code. The vectors are of size N . The kernel is launched with at least N amount of threads.

```

procedure VectorVectorOperation( $x, y, \alpha, \beta, r, N$ )
   $row \leftarrow \text{getGlobalID}()$ 
  if  $row < N$  then           // Check if current thread is within the bounds of the vectors
     $r[row] \leftarrow \alpha \cdot x[row] + \beta \cdot y[row]$ 
  end if
  return  $r$ 
end procedure

```

6.1.3 Vector Reduction

With two vectors x and y of size N , the dot product can be defined as

$$x^T y = r, \tag{6.3}$$

resulting in a scalar r .

Like in the vector addition, only a single for-loop is needed to achieve the dot product of two vectors. The sequential code is shown in Algorithm 6.5. As the result is incremented by each thread with its sub-result, setting the result to zero before starting the kernel is essential. In the case of multi-threading, all threads have to access the same memory location. Therefore, it must be ensured that no two threads are writing to the identical memory location at the same time. This can be achieved by extending the added OpenMP compiler instruction above the loop.

In the case of native subtask implementation, attention must be paid to data access, too. Again, access is first assigned by row to each thread. After the multiplication, the first thread of a block sums up the results of the other threads of the same block. Afterward, `atomicAdd` is used to add the result of the block to the global result, which is a blocking operation. The resulting kernel is shown in Algorithm 6.6. The complete data access is shown in Figure 6.3, where thread 1 to l represent one block, and every color represents a different thread. In this case, the result has to be set to zero before starting the kernel to avoid multiple access to the same memory location. In this

Algorithm 6.5 Here, the resulting kernel for a vector reduction for the CPU and FPGA is displayed. The OpenMP compiler instruction can be included optionally when running on the CPU to dynamically subdivide the loop into subtasks assigned to different threads.

```

procedure VectorReduction( $\mathbf{x}, \mathbf{y}, r, N$ )
   $rz \leftarrow 0$ 
  (#pragma omp parallel for reduction(+:r))           // OpenMP parallelization
  for  $i = 1, 2, \dots, N$  do                          // Iterate over rows
     $r \leftarrow r + x[i] \cdot y[i]$ 
  end for
  return  $r$ 
end procedure

```

implementation, the block size is of crucial importance. If the block size is too small, the overhead of the many launched blocking operations will be too high. If the block size is too large, all threads, except one of a block, have a lot of idle time.

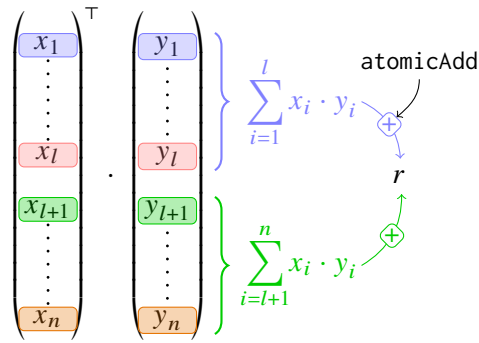


Figure 6.3: Depicted memory access of a vector reduction kernel on the GPU. The vectors are of size N . Each thread is represented by one color and accesses one element of each vector. Threads inside one block share the same memory location for the multiplication of the entries. In each block, the first thread sums up the partial results of the other threads and adds the result to the global result with a blocking operation.

Algorithm 6.6 Here, the resulting kernel for a vector reduction for the GPU is displayed. All threads of a block write to a shared memory location. The first thread of a block sums up the results of the other threads of the same block and adds the result to the global result with a blocking operation.

```

procedure VectorReduction( $\mathbf{x}, \mathbf{y}, r, N$ )
     $row \leftarrow \text{getGlobalID}()$ 
     $localID \leftarrow \text{getLocalID}()$ 
     $blockSize \leftarrow \text{getBlockSize}()$ 
    if  $row < N$  then                                // Check if current thread is within the bounds of the vectors
         $temp[localID] \leftarrow x[row] \cdot y[row]$ 
    else
         $temp[localID] \leftarrow 0$ 
    end if
    syncThreads()                                    // Wait for all threads within same block to finish
    if  $localID = 0$  then
        for  $i = 1, 2, \dots, blockSize$  do                // Sum up all values in temp
             $temp[0] \leftarrow temp[0] + temp[i]$ 
        end for
        atomicAdd( $temp[0], r$ )                        // Add the partial sum to the global result
    end if
    return  $r$ 
end procedure

```

6.2 Conjugate Gradient Method

The CG, as described in Chapter 4, consists of multiple basic linear operations. In order to fulfill the predefined abortion criterion ε , the squared norm of the residual δ can be defined as

$$\delta = \|\tilde{\mathbf{r}}\|^2 = \tilde{\mathbf{r}}^\top \tilde{\mathbf{r}}. \quad (6.4)$$

Together with the maximum amount of steps, this determines the abortion criterion for the while-loop of the iterative solver. Because of the limited precision of floating point numbers, propagation of errors occurs. Therefore, error correction can be added by occasionally calculating the residual by Equation (4.2) instead of Equation (4.6). In this implementation, this is done every 50th step. Nevertheless, this operation is more expensive and should be avoided, if possible.

All required operations are listed in Table 6.1. In order to reduce data access and calculations, every operation is implemented separately.

In the case of using the BLAS library, only certain operations are needed. Nevertheless, a BLAS function is only available in some cases, resulting in possible unnecessary data accesses and additional operations. It is also important to mention that these functions mostly perform in-place operations, meaning the result is written to the same memory location as the input. This can be problematic if the input is still needed in the following steps leading to additional operations by first copying into temporary memory. All used BLAS functions are listed in Table 6.2.

The resulting CG functions are presented in Algorithm 6.7. On the left, the implementation using native functions is shown. In the middle, the implementation using BLAS functions is written down. The comments on the right side indicate the matching equation. In some cases, additional functions

Equation	Native Function	Template
$\mathbf{A} \mathbf{x}$	$\mathbf{r} \leftarrow \text{AX}(\mathbf{A}, \mathbf{x}, N, N)$	Algorithm 6.1 & Algorithm 6.2
$\mathbf{y} - \mathbf{A} \mathbf{x}$	$\mathbf{r} \leftarrow \text{YMAX}(\mathbf{A}, \mathbf{x}, \mathbf{y}, \alpha, N, N)$	Algorithm 6.1 & Algorithm 6.2
$\alpha \mathbf{x} + \mathbf{y}$	$\mathbf{r} \leftarrow \text{AXPY}(\mathbf{x}, \mathbf{y}, \alpha, N)$	Algorithm 6.3 & Algorithm 6.4
$\mathbf{x} + \alpha \mathbf{y}$	$\mathbf{r} \leftarrow \text{XPAY}(\mathbf{x}, \mathbf{y}, \alpha, N)$	Algorithm 6.3 & Algorithm 6.4
$\mathbf{x}^\top \mathbf{y}$	$r \leftarrow \text{VECREduc}(\mathbf{x}, \mathbf{y}, N)$	Algorithm 6.5 & Algorithm 6.6

Table 6.1: All natively implemented linear algebra kernels are listed here. The first two columns show the equation and the corresponding native function. The third column shows the corresponding underlying template the functions were implemented with.

Equation	BLAS Function
$\alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}$	$\mathbf{y} \leftarrow \text{GEMV}(N, N, \alpha, \mathbf{A}, \mathbf{x}, \beta, \mathbf{y})$
$\alpha \mathbf{x} + \mathbf{y}$	$\mathbf{y} \leftarrow \text{AXPY}(N, \alpha, \mathbf{x}, \mathbf{y})$
$\alpha \mathbf{x}$	$\mathbf{y} \leftarrow \text{SCAL}(N, \alpha, \mathbf{x})$
$\mathbf{x}^\top \mathbf{y}$	$r \leftarrow \text{DOT}(N, \mathbf{x}, \mathbf{y})$

Table 6.2: All used Basic Linear Algebra Subprograms (BLAS) functions for the Conjugate Gradient Method algorithm are listed here.

must be performed when using BLAS operations, as an exact implementation is not provided. In the case of the dot product, the BLAS kernel sets the result to zero, which is why this must not be performed before the kernel call.

Algorithm 6.7 The CG algorithm implemented with native functions and BLAS functions on the left and right side respectively are shown. The comments on the right side indicate the matching equation.

<pre> procedure CG($\tilde{\mathbf{A}}, \mathbf{b}, \tilde{\mathbf{x}}, N, i_{\max}, \varepsilon$) $\tilde{\mathbf{r}} \leftarrow \text{ymAx}(\tilde{\mathbf{A}}, \tilde{\mathbf{x}}, \mathbf{b}, N, N)$ $\mathbf{d} \leftarrow \tilde{\mathbf{r}}$ $\delta_{\text{New}} \leftarrow 0$ $\delta_{\text{New}} \leftarrow \text{VecReduc}(\tilde{\mathbf{r}}, \tilde{\mathbf{r}}, N)$ $\delta_0 \leftarrow \delta_{\text{New}}$ $i \leftarrow 0$ while $i < i_{\max} \wedge \varepsilon^2 \delta_0 < \delta_{\text{New}}$ do $\mathbf{q} \leftarrow \text{Ax}(\tilde{\mathbf{A}}, \mathbf{d}, N, N)$ $\tilde{\alpha} \leftarrow 0$ $\tilde{\alpha} \leftarrow \text{VecReduc}(\mathbf{d}, \mathbf{q}, N)$ $\tilde{\alpha} \leftarrow \delta_{\text{New}} / \tilde{\alpha}$ $\tilde{\mathbf{x}} \leftarrow \text{axpy}(\mathbf{d}, \tilde{\mathbf{x}}, \tilde{\alpha}, N)$ if $i \% 50$ then $\tilde{\mathbf{r}} \leftarrow \text{aAxy}(\tilde{\mathbf{A}}, \tilde{\mathbf{x}}, \mathbf{b}, -1, N, N)$ else $\tilde{\mathbf{r}} \leftarrow \text{axpy}(\mathbf{q}, \tilde{\mathbf{r}}, -\tilde{\alpha}, N)$ end if $\delta_{\text{Old}} \leftarrow \delta_{\text{New}}$ $\delta_{\text{New}} \leftarrow 0$ $\delta_{\text{New}} \leftarrow \text{VecReduc}(\tilde{\mathbf{r}}, \tilde{\mathbf{r}}, N)$ $\tilde{\beta} \leftarrow \delta_{\text{New}} / \delta_{\text{Old}}$ $\mathbf{d} \leftarrow \text{xpay}(\tilde{\mathbf{r}}, \mathbf{d}, \tilde{\beta}, N)$ $i \leftarrow i + 1$ end while return $\tilde{\mathbf{x}}$ end procedure </pre>	<pre> procedure CGblas($\tilde{\mathbf{A}}, \mathbf{b}, \tilde{\mathbf{x}}, N, i_{\max}, \varepsilon$) $\tilde{\mathbf{r}} \leftarrow \mathbf{d}$ // $\tilde{\mathbf{r}} = \mathbf{b} - \tilde{\mathbf{A}} \cdot \tilde{\mathbf{x}}$ $\tilde{\mathbf{r}} \leftarrow \text{gemv}(N, N, -1, \tilde{\mathbf{A}}, \tilde{\mathbf{x}}, 1, \tilde{\mathbf{r}})$ $\mathbf{d} \leftarrow \tilde{\mathbf{r}}$ $\delta_{\text{New}} \leftarrow \text{dot}(N, \tilde{\mathbf{r}}, \tilde{\mathbf{r}})$ // $\delta_{\text{New}} = \tilde{\mathbf{r}}^\top \cdot \tilde{\mathbf{r}}$ $\delta_0 \leftarrow \delta_{\text{New}}$ $i \leftarrow 0$ while $i < i_{\max} \wedge \varepsilon^2 \delta_0 < \delta_{\text{New}}$ do $\mathbf{q} \leftarrow \text{gemv}(1, \tilde{\mathbf{A}}, \mathbf{d}, 0, \mathbf{q}, N, N)$ // $\mathbf{q} = \tilde{\mathbf{A}} \cdot \mathbf{d}$ $\tilde{\alpha} \leftarrow \text{dot}(N, \mathbf{d}, \mathbf{q})$ // $\tilde{\alpha} = \frac{\delta_{\text{New}}}{\mathbf{d}^\top \cdot \mathbf{q}}$ $\tilde{\alpha} \leftarrow \delta_{\text{New}} / \tilde{\alpha}$ $\tilde{\mathbf{x}} \leftarrow \text{axpy}(N, \tilde{\alpha}, \mathbf{d}, \tilde{\mathbf{x}})$ // $\tilde{\mathbf{x}} = \tilde{\alpha} \cdot \mathbf{d} + \tilde{\mathbf{x}}$ if $i \% 50$ then $\tilde{\mathbf{r}} \leftarrow \mathbf{d}$ // $\tilde{\mathbf{r}} = \mathbf{b} - \tilde{\mathbf{A}} \cdot \tilde{\mathbf{x}}$ $\tilde{\mathbf{r}} \leftarrow \text{gemv}(N, N, -1, \tilde{\mathbf{A}}, \tilde{\mathbf{x}}, 1, \tilde{\mathbf{r}})$ else $\tilde{\mathbf{r}} \leftarrow \text{axpy}(N, \tilde{\alpha}, \mathbf{q}, \tilde{\mathbf{r}})$ // $\tilde{\mathbf{r}} = \tilde{\mathbf{r}} - \tilde{\alpha} \cdot \mathbf{q}$ end if $\delta_{\text{Old}} \leftarrow \delta_{\text{New}}$ $\delta_{\text{New}} \leftarrow \text{dot}(N, \tilde{\mathbf{r}}, \tilde{\mathbf{r}})$ // $\delta_{\text{New}} = \tilde{\mathbf{r}}^\top \cdot \tilde{\mathbf{r}}$ $\tilde{\beta} \leftarrow \delta_{\text{New}} / \delta_{\text{Old}}$ $\mathbf{d} \leftarrow \text{scal}(N, \tilde{\beta}, \mathbf{d})$ // $\mathbf{d} = \tilde{\mathbf{r}} + \tilde{\beta} \cdot \mathbf{d}$ $\mathbf{d} \leftarrow \text{axpy}(N, 1, \tilde{\mathbf{r}}, \mathbf{d})$ $i \leftarrow i + 1$ end while return $\tilde{\mathbf{x}}$ end procedure </pre>
--	--

7 Results

7.1 Runtimes

The different kernels explained in Section 6.1 were investigated in the first step. In Figure 7.1, the runtimes of the different kernels are plotted using all implementations mentioned in Chapter 3. The subfigures show the execution time on the CPU and GPU of Chapter 2, respectively. In the case of matrix-vector operations, only square matrices ($N = M$) are investigated. For better clarity, not all measuring points and their variance are displayed.

For running on the CPU runtimes of a sequential, OpenMP, native SYCL™, and oneMKL implementation are compared. With smaller matrix sizes of $N = 64$, the sequential implementation of the matrix-vector operation provides $t_{\text{Sequ}} = 2 \mu\text{s}$. This is the fastest runtime on the CPU, as observable in Figure 7.1a. OpenMP has a runtime of $t_{\text{OpenMP}} = 22.6 \mu\text{s}$ and is more than ten times slower than the sequential implementation. With $t_{\text{SYCL}^\text{TM}} = 1,426 \mu\text{s}$ the SYCL™ implementation is by far the slowest implementation on the CPU. The oneMKL implementation provides a speedup of nearly ten compared to native SYCL™, but is still slower than the OpenMP implementation. Slower runtimes of the OpenMP, SYCL™, and oneMKL implementations are caused by non-neglectable overheads. As the matrices are small, the communication between threads as well as the data movement between the different caches result in a significant overhead compared to the computational intensity. With increasing dimension, the overhead becomes less significant. OpenMP already performs the operation faster than the sequential implementation at $N = 512$. For SYCL™ and oneMKL this is the case for $N = 1,024$. With larger matrices ($N = 65,536$) the sequential implementation takes with $t_{\text{Sequ}} = 7,040,549 \mu\text{s}$ nearly seven times longer than the other implementations. The sequential code only uses one core compared to the other implementations using all six. When taking overhead due to communication between cores into account, a factor of smaller than six is expected. Due to Hyper-Threading and therefore a more efficient usage of the cores this speedup increases more, explaining the factor of seven. OpenMP, SYCL™, and oneMKL show nearly the same runtimes of $t_{\text{OpenMP}} = 1,029,308.8 \mu\text{s}$, $t_{\text{SYCL}^\text{TM}} = 1,088,051.4 \mu\text{s}$, and $t_{\text{oneMKL}} = 1,148,045.4 \mu\text{s}$, respectively. The native SYCL™ implementation is slightly faster than the oneMKL implementation, but in smaller cases performs worse. Meaning, the oneMKL implementation provides a significant speedup over the native SYCL™ implementation, but only with smaller matrices. Still, on the CPU OpenMP provides the fastest implementation for large matrices.

Similar behavior on the CPU can be observed with vector addition in Figure 7.1c. The sequential implementation is the fastest for small vectors of size $N = 64$. It is seven times faster than OpenMP, 80 times faster than oneMKL and more than 6,000 times faster than the native SYCL™ implementation. Especially the overhead of the native SYCL™ implementation is enormous. With increasing vector size, the overheads of the implementations become less significant resulting in OpenMP and oneMKL catching up with the sequential implementation. With vector sizes $N > 4,096$

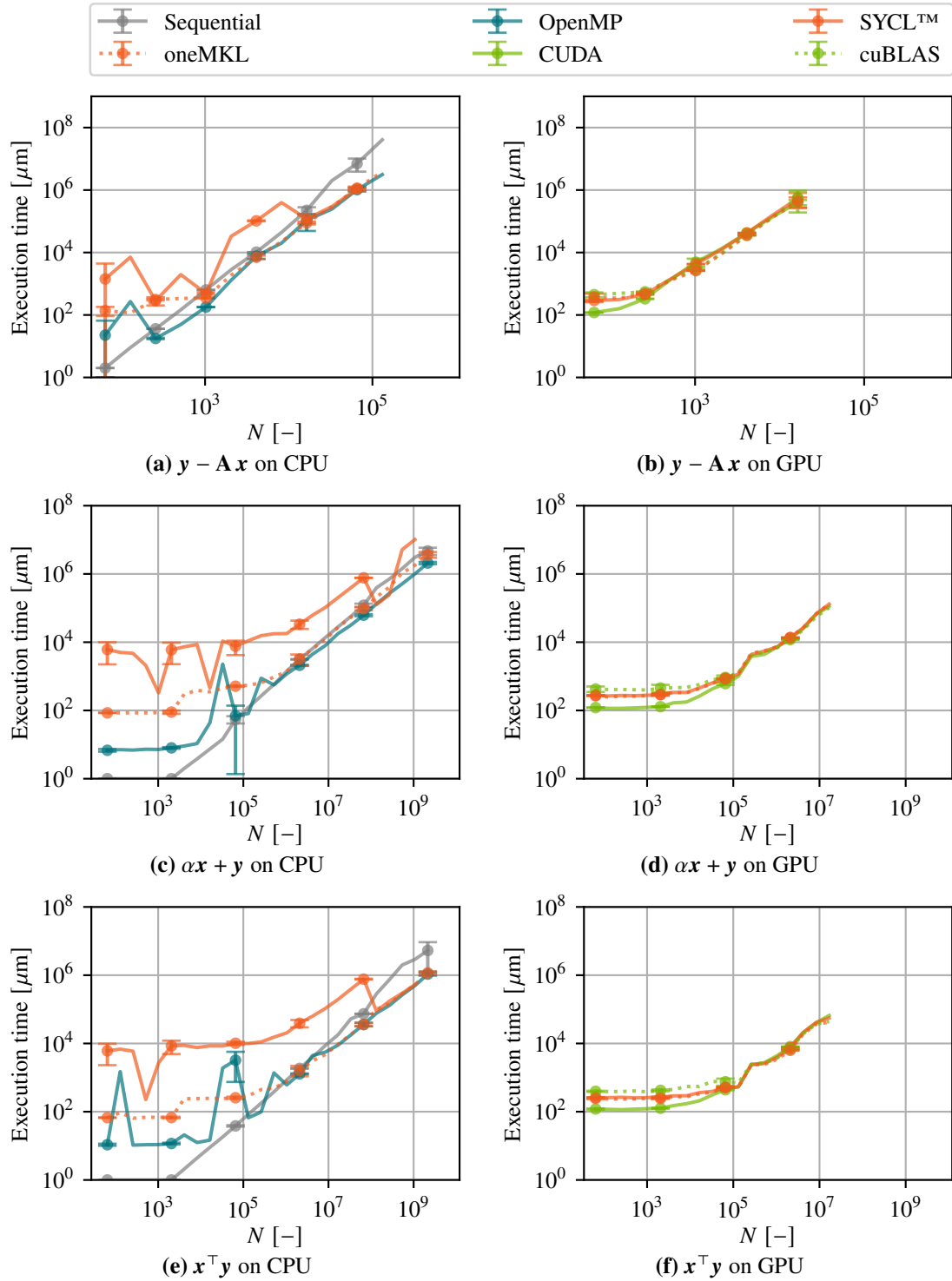


Figure 7.1: The graphs show runtimes over vector size N of representative linear algebra kernels. Not all measuring points and their variance are displayed for better readability. On the left side, the runtimes on the CPU are shown, on the right side the runtimes on the GPU. For the CPU OpenMP, SYCL™, oneMKL, and a sequential implementation are compared. For the GPU SYCL™, oneMKL, CUDA, and cuBLAS are compared.

both provide better runtimes than the sequential code. The native SYCL™ implementation is still the slowest. Vector sizes of $N = 1,073,741,824$ are the largest sizes that can be processed by all implementations. In this case, the OpenMP implementation is the fastest with a runtime of $t_{\text{OpenMP}} = 1,002,829.1 \mu\text{s}$. This is more than 1.7 times faster than the oneMKL implementation. The native SYCL™ implementation is more than ten times slower than the OpenMP implementation and still more than three times slower than the sequential implementation.

The vector reduction kernel (Figure 7.1e) shows a similar behavior as the vector addition kernel. The sequential implementation is the fastest for small vectors of size $N = 64$ on the CPU. It is more than ten times faster than OpenMP, 65 times faster than oneMKL and more than 6,000 times faster than the native SYCL™ implementation. Again OpenMP and oneMKL catch up with the sequential implementation with increasing vector size. In contrast to the vector addition kernel, the native SYCL™ implementation provides faster runtimes than the sequential implementation, but only for large vector sizes $N > 268,435,456$. Investigating the runtimes at $N = 1,073,741,824$, again, the OpenMP implementation is the fastest. It is more than six times faster than the sequential implementation. Again, this is explainable due to the sequential implementation only using one core instead of six and Hyper-Threading. Due to all threads accessing the same memory, the overhead is slightly bigger than in the matrix-vector operation. Therefore the speedup is minimal smaller. The oneMKL implementation as well as the native SYCL™ implementation provide equal runtimes and are only slightly slower (~ 1.1) than the OpenMP implementation.

For the GPU, the linear algebra kernels were implemented using CUDA, cuBLAS, SYCL™, and oneMKL. In the case of the matrix-vector operation, all four implementations provide nearly the same runtimes as shown in Figure 7.1b. Due to the smaller size of the DRAM of the GPU compared to the DRAM of the CPU, the sizes of the matrix and the vector are more limited. Therefore, not all matrix sizes can be processed by all implementations. With dimension set to $N = 64$ the differences in the overheads can be observed. The native CUDA implementation with $t_{\text{CUDA}} = 119.6 \mu\text{s}$ provides the fastest runtime. The cuBLAS implementation is more than 3.5 times slower and therefore the slowest implementation. The SYCL™ implementation needs more than two times as long as the native CUDA implementation and the oneMKL implementation more than 3 times.

The same behavior on the GPU can be observed for the vector addition (Figure 7.1d) and the vector reduction (Figure 7.1f). For small vectors, the differences in the overheads of the implementations become visible. Again, the native CUDA implementation is the fastest, while the cuBLAS implementation is the slowest. SYCL™ and oneMKL are in between. With increasing vector size, the computational intensity increases and the overheads of the implementations become less significant. Therefore, all four implementations again provide nearly the same runtimes. The fact that all implementations provide nearly the same runtimes for large problems indicates that the data movement between the CPU and the GPU is more time-consuming than the actual computation.

A comparison of the architectures is achieved in Figure 7.2 by plotting the runtime over the size of the problem for a representative of each device. Again, for readability reasons not all data points and their respective variance are plotted. In the case of the CPU, the OpenMP implementation is used as it provides the fastest results. The cuBLAS implementation is used for the GPU. For the FPGA, the native SYCL™ implementation had to be used as SYCL™ is the only library providing the possibility to include FPGA devices. Although oneMKL states that it is possible to use FPGA devices, the oneMKL implementation could not be run on the FPGA device. As the different

devices provide different computational power, the runtimes of the kernels can be scaled with their theoretical peak performance, resulting in a pseudo-efficiency. This pseudo-efficiency is plotted over the dimension N in Figure 7.2 as well.

In all cases (matrix-vector operation, vector addition, and vector reduction), the CPU was the fastest device. However, this is because of a considerable overhead. The matrices and vectors must be moved back and forth from the CPU to the GPU and FPGA, respectively. As only one simple linear algebra operation is performed, the overhead is significant, resulting in the runtime of the CPU being much lower than the runtime of the GPU and FPGA.

Figure 7.2a, showing runtimes for the matrix-vector multiplication, indicates a faster execution of the GPU in comparison to the FPGA, while the scaling is similar. The GPU is able to perform the matrix-vector multiplication around five times faster than the FPGA independent of the sizes. The CPU is significantly faster and has a runtime of $t_{\text{CPU}} = 19,990.7 \mu\text{s}$ for $N = 8,192$. With a matrix of size $8,192 \times 8,192$ and three vectors of size 8,192, the minimal duration of copying the data back and forth to the GPU would be $33,566.72 \mu\text{s}$ as the GPU is connected via PCIe 3.0 x16. This is more than 1.5 times the runtime of the CPU, explaining why the CPU is faster than the GPU. The same holds for the FPGA.

Figure 7.2c shows that the FPGA is significantly slower and, more importantly, scales a lot worse than the CPU and GPU in case of vector addition. For $N = 64$ the FPGA needs $t_{\text{FPGA}} = 644.6 \mu\text{s}$ to perform the vector addition, while the CPU needs $t_{\text{CPU}} = 6.8 \mu\text{s}$ and the GPU $t_{\text{GPU}} = 425.1 \mu\text{s}$. In this case, the CPU is around 60 times faster than the GPU and even nearly 100 times faster than the FPGA. The GPU becomes faster for larger dimensions resulting in the runtime only being a factor of five slower for $N = 16,777,216$. For the FPGA the runtime is more than 1,000 times slower, underlying the insufficient scaling of the FPGA for vector addition.

The vector reduction is the only kernel where the FPGA can compete with the GPU, as shown in Figure 7.2e. Both devices are still significantly slower than the CPU, but this is again because of the overhead. With $t_{\text{GPU}} = 396.3 \mu\text{s}$ and $t_{\text{FPGA}} = 421.1 \mu\text{s}$ the GPU and FPGA, respectively, provide similar runtimes for $N = 64$. For sizes of $N = 16,777,216$ the CPU is around five times faster than the GPU and around 40 times faster than the FPGA. The GPU is still faster than the FPGA, but the difference is not as significant as for the vector addition.

The matrix-vector operation reveals an equal pseudo-efficiency for the GPU and the FPGA, while the CPU is significantly more efficient with a factor of at least 380. Still, this pseudo-efficiency also contains the overhead due to the data transfer, resulting in a significant difference between the pure kernel launch pseudo-efficiency and the shown pseudo-efficiency in Figure 7.2b. As already observed in the runtime of the vector addition, Figure 7.2d states similar results for the efficiency. The FPGA can not keep up with the GPU. The vector reduction is the only kernel where the FPGA provides a better pseudo-efficiency than the GPU, as one can see in Figure 7.2f. Still, the overhead of the data transfer is significant, resulting in the pseudo-efficiency of the FPGA being lower than the pseudo-efficiency of the CPU.

The runtimes of the complete CG implementation point out similar relations as the runtimes of the kernels for the CPU (Figure 7.3). The sequential algorithm is the fastest for small vector sizes, because there is no implied overhead. It is more than 15 times faster than OpenMP, nearly 100 times faster than oneMKL and 300 times faster than the native SYCL™ implementation in case of $N = 64$. For increasing problem sizes the OpenMP implementation becomes the fastest code. For $N = 8,192$ the OpenMP implementation is around five times faster than the sequential code.

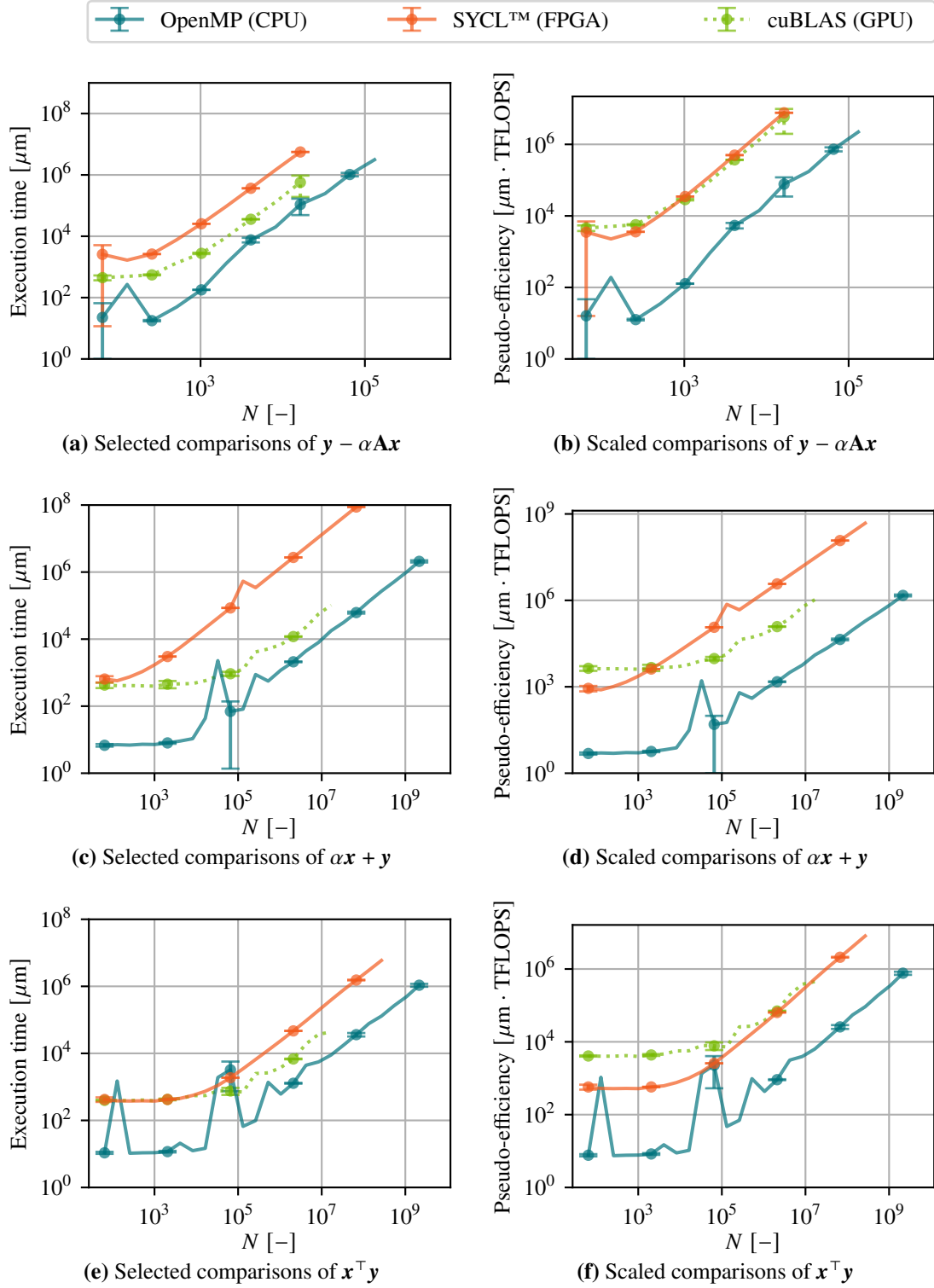


Figure 7.2: The plots illustrate runtimes and pseudo-efficiency over vector size N of representative linear algebra kernels. For better clarity, not all measuring points and their variance are plotted. OpenMP is used as representative for the CPU, cuBLAS for the GPU, and SYCL™ for the FPGA. The pseudo-efficiency is calculated by multiplying the runtime with the theoretical peak performance.

SYCL[™] and oneMKL provide similar runtimes that are approximately two times slower than the OpenMP implementation. It is observable in Figure 7.3 that the optimizations of oneMKL are effective only for smaller problem sizes, as it only provides faster runtimes for smaller N .

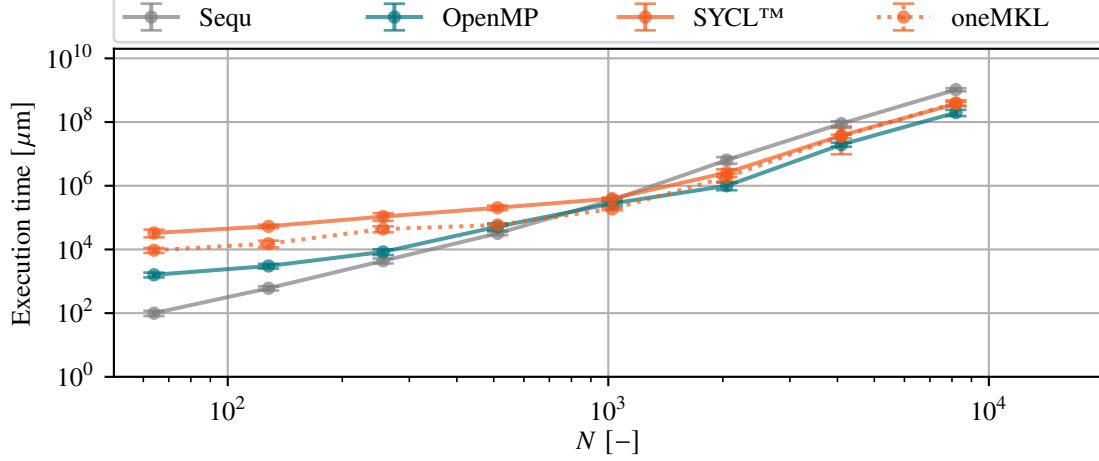


Figure 7.3: Runtimes over problem size N of the CG implementations on the CPU are compared here. The implementations are the sequential code, the OpenMP implementation, the native SYCL[™] implementation and the oneMKL implementation.

Figure 7.4 shows the runtimes of the CG implementations on the GPU. In this case, the overhead of moving the data to the GPU is negligible compared to the runtime of the kernels. For smaller problems, both implementations using SYCL[™] indicate a slightly larger overhead compared to the CUDA implementations. This results in the CUDA implementations being around three times faster than the SYCL[™] implementations in case of $N = 64$. For larger problems, the overhead of the SYCL[™] implementations is negligible. Overall, the BLAS libraries oneMKL and cuBLAS, increase efficiency with increasing problem size compared to the native implementations. For a problem size of $N = 16,384$ the cuBLAS implementation and the oneMKL implementation need $t_{\text{cuBLAS}} = 773$ s and $t_{\text{oneMKL}} = 785$ s, respectively. This is around four times faster than the native SYCL[™] $t_{\text{SYCL}^{\text{™}}} = 3,067$ s and CUDA $t_{\text{CUDA}} = 3,318$ s implementations.

Again, the OpenMP implementation as the fastest representative for the CPU can be compared with the cuBLAS implementation as the fastest representative for the GPU and the SYCL[™] implementation as the only one that runs on the FPGA. This is shown in Figure 7.5. Since the computational effort increases significantly compared to a single linear algebra kernel, the overhead of moving the data back and forth from CPU to GPU and FPGA is less significant. Therefore, the comparison is more meaningful. Independent of the vector size, the FPGA is always by far the slowest device. The runtimes are always at least ten times longer than the ones of the CPU and GPU. This is due to the slow performance of the vector addition. For small problems, the OpenMP implementation is the fastest and close to two times faster than the cuBLAS code. It is important to mention that, due to the overhead, OpenMP was not the fastest implementation on the CPU in cases of small problems. Therefore, the sequential implementation would state an even more significant advantage of the CPU over the GPU with a factor of 30. With increasing problem size, the GPU becomes significantly faster than every other device, solving for the solution vector the fastest. In

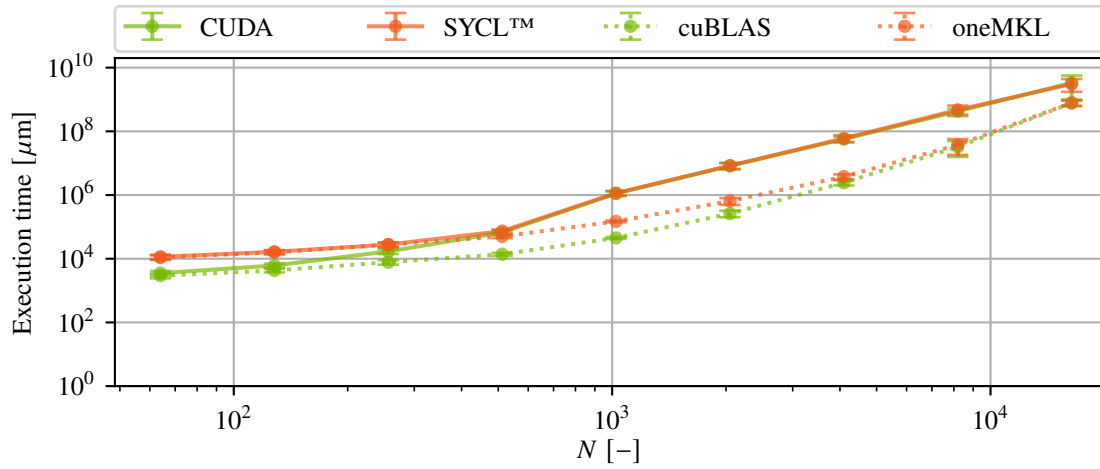


Figure 7.4: Graphs show runtimes over problem size N of the CG implementations on the GPU. SYCLTM, oneMKL, CUDA, and cuBLAS are compared.

case of $N = 8,192$ it is already faster than the CPU by a factor of six. With the runtimes increasing rapidly, not all devices were able to solve the problem in a reasonable time. That is why CPU and especially FPGA runtimes are not depicting runtimes for large problems.

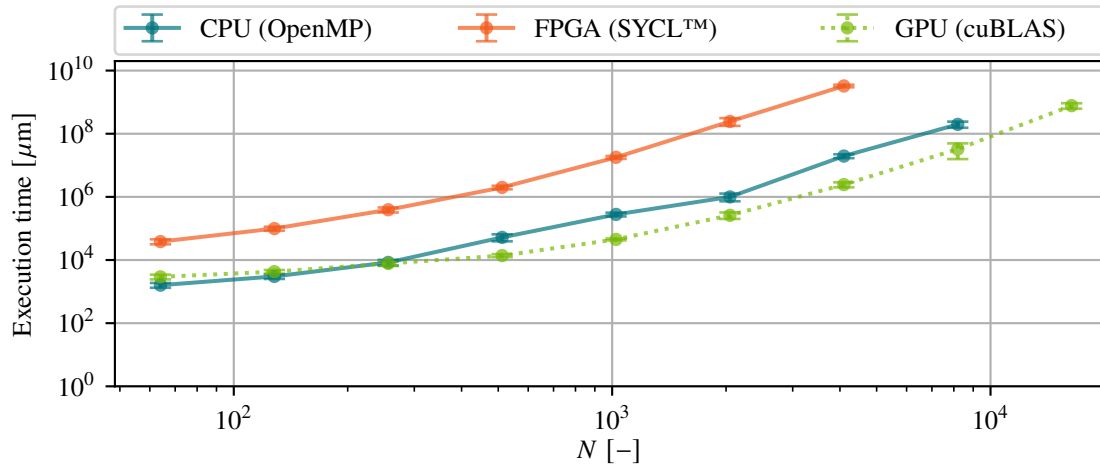


Figure 7.5: The graphs illustrate runtimes over problem size N of the CG implementations. OpenMP is used as the representative for the Central Processing Unit (CPU), cuBLAS for the GPU and SYCLTM for the FPGA.

Also, in the case of the CG, the runtimes can be scaled with the theoretical peak performance of the devices. This is shown in Figure 7.6. This points out the CPU as the most efficient device for the CG implementation. Even for large problem sizes like $N = 8,192$ the pseudo-efficiency is higher by 1.66 compared to the pseudo-efficiency of the GPU. This makes the GPU the second most efficient device, while the FPGA still performs poorly.

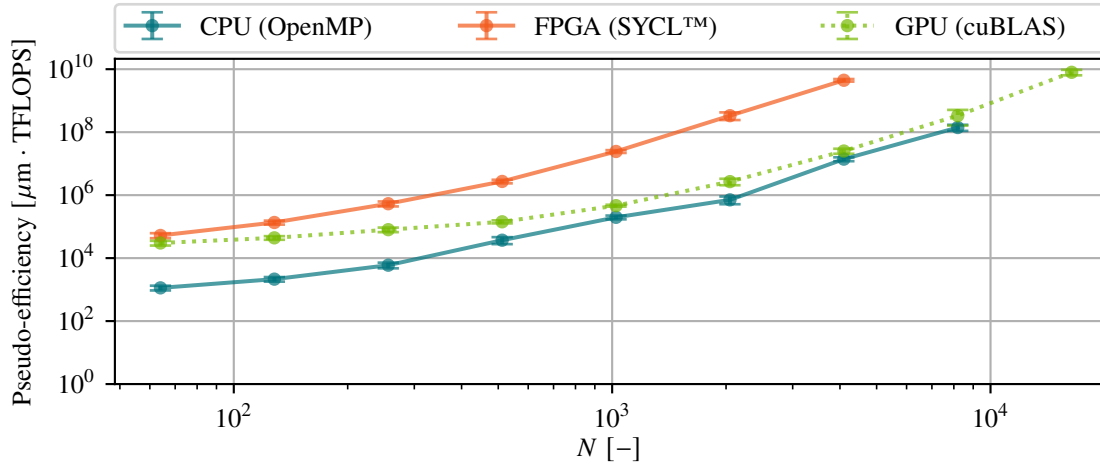


Figure 7.6: Pseudo-efficiencies over problem size N of the CG implementations are plotted here. OpenMP is used as the representative for the CPU, cuBLAS for the GPU and SYCL™ for the FPGA. The pseudo-efficiency is calculated as the multiplication of the theoretical peak performance of the respective device and the runtime of the implementation.

7.2 Performance Portability Analysis

In order to perform a performance portability analysis, the metric described in Chapter 5 is used. The analysis is applied to the CG implementations with a vector size of $N = 4096$, as this is the largest problem size that all devices were capable of solving. Again each implementation is executed ten times on each device. These runtimes are then averaged. The best averaged runtimes of each device are listed in Table 7.1.

p	$\min_{q \in Q} t [\mu s]$
CPU	1.960e7
GPU	2.445e6
FPGA	3.259e9

Table 7.1: Fastest achieved averaged runtimes of the CG implementations on each device for $N = 4,096$ are listed here.

The amount of ideal floating point operations must be determined to calculate the architectural efficiency. In Table 7.2, the ideal amount of operations for the linear algebra operations used in the while-loop of the CG algorithm is listed.

Table 7.3 shows runtimes and the corresponding applicational as well as architectural efficiencies for the CG implementations. The architectural efficiency is calculated by using only one iteration of the loop. Therefore, the runtime of the CG is divided by the amount of iterations. This implies an overhead since moving the data and calculations before the loop can not be excluded from the runtime.

Routine	$\mathcal{O}_{\text{ideal}}$
$\mathbf{q} = \tilde{\mathbf{A}} \cdot \mathbf{d}$	$(N + (N - 1)) \cdot N$
$\tilde{\alpha} = \frac{\delta_{\text{New}}}{\mathbf{d}^\top \cdot \mathbf{q}}$	$1 + N + (N - 1)$
$\tilde{\mathbf{x}} = \tilde{\alpha} \cdot \mathbf{d} + \tilde{\mathbf{x}}$	$N + N$
$\tilde{\mathbf{r}} = \mathbf{b} - \tilde{\mathbf{A}} \cdot \tilde{\mathbf{x}}$	$(N + (N + (N - 1)) \cdot N) \cdot 1/50$
$\tilde{\mathbf{r}} = \tilde{\mathbf{r}} - \tilde{\alpha} \cdot \mathbf{q}$	$(N + N) \cdot 49/50$
$\delta_{\text{New}} = \tilde{\mathbf{r}}^\top \cdot \tilde{\mathbf{r}}$	$N + (N - 1)$
$\tilde{\beta} = \frac{\delta_{\text{New}}}{\delta_{\text{Old}}}$	1
$\mathbf{d} = \tilde{\mathbf{r}} + \tilde{\beta} \cdot \mathbf{d}$	$N + N$
$i = i + 1$	1
Total	$\frac{51}{25}N^2 + \frac{224}{25}N + 1$

Table 7.2: Here, the ideal amount of floating point operations of each subtask for the CG implementation is calculated.

q	p	t [μs]	$e_{\text{App. Eff.}}$ [%]	$e_{\text{Arch. Eff.}}$ [%]
OpenMP	CPU	1.960e7	100	0.02518
OpenMP	GPU	-	-	-
OpenMP	FPGA	-	-	-
CUDA	CPU	-	-	-
CUDA	GPU	5.685e7	4.301	0.0005215
CUDA	FPGA	-	-	-
cuBLAS	CPU	-	-	-
cuBLAS	GPU	2.445e6	100	0.01176
cuBLAS	FPGA	-	-	-
SYCL TM	CPU	3.778e7	51.88	0.01299
SYCL TM	GPU	6.022e7	4.060	0.0005219
SYCL TM	FPGA	3.259e9	100	7.966e-5
oneMKL	CPU	3.564e7	54.99	0.01269
oneMKL	GPU	3.778e6	64.72	0.007607
oneMKL	FPGA	-	-	-

Table 7.3: The table lists runtimes and efficiencies of the CG implementations with $N = 4,096$ on the CPU, GPU, and FPGA using OpenMP, CUDA, cuBLAS, SYCLTM, and oneMKL.

As already discussed in Section 7.1, the OpenMP implementation is the fastest implementation for the CPU. This is the reason why its applicational efficiency is 100 %. The same holds for the cuBLAS code on the GPU. As the native SYCLTM implementation is the only one running on the FPGA it also has a applicational efficiency of 100 %.

When looking at the architectural efficiencies, it is observable that all CPU implementations are more efficient than the ones on the GPU. This means that even for the CG implementation, consisting of basically only matrix and vector operations, the GPU is not more efficient than the CPU. This underlines the statement of Chapter 2, that it is challenging to design a GPU kernel that is more efficient than the CPU implementation. In the case of the FPGA the efficiency is extremely low. As originally the FPGA promised high efficiency, this is unexpected and underlines the need for further research and optimizations.

The final results of the performance portability analysis are shown in Table 7.4. Only SYCLTM is able to achieve performance portability that is not zero. It is still worth mentioning that separate kernels for the FPGA and GPU had been written in the native SYCLTM implementation. Otherwise, the performance would have been significantly worse.

q	$\mathcal{P}_{\text{App. Eff.}} [\%]$	$\mathcal{P}_{\text{Arch. Eff.}} [\%]$
OpenMP	0	0
CUDA	0	0
cuBLAS	0	0
SYCL TM	10.89	2.062e-4
oneMKL	0	0

Table 7.4: Results of the performance portability analysis for the CG with $N = 4,096$ implementations are shown in this table.

8 Conclusion and Outlook

The main goal of this work was to investigate the performance of SYCL™ on different hardware architectures. Therefore, a performance portability analysis was performed. At first, different hardware architectures were briefly described and APIs to execute code on these hardware devices. Next, the CG algorithm was briefly explained as it provides a good benchmark for the performance of the APIs. Also, a metric to perform a performance portability analysis was introduced.

In order to be able to compare the performance of SYCL™ to other APIs, a reference implementation was needed. Therefore, a CUDA implementation was written for the GPU as well as an OpenMP implementation for the CPU. The different linear algebra operations needed for the CG were investigated separately for a deeper understanding. The results show that the runtimes of SYCL™ are in the range of OpenMP implementation for the CPU. The same holds for a native implementation of the linear algebra operations on the GPU with CUDA and SYCL™. When using cuBLAS, the performance is significantly better. With the oneMKL, similar runtimes can be achieved. Additionally, SYCL™ was able to run on the FPGA, although the performance was not as good as on the CPU and GPU, especially in the case of vector addition.

After investigating the linear algebra operations separately, the CG algorithm was implemented in SYCL™ and compared to the reference implementations. The results show that the performance of SYCL™ is in the range of the OpenMP implementation for the CPU and the CUDA implementation for the GPU as well. The same holds for the oneMKL implementation on the GPU compared to cuBLAS. When executing on the FPGA, the runtimes were also poor.

The performance portability analysis led to unexpected disappointing results. Except for the native SYCL™ implementation, none of the programs could run on all devices and end up with a score of zero. As using oneMKL on the FPGA was not possible, the efficiency of the native SYCL™ implementation on the GPU was unsatisfactory, leading to a extremely low scoring. Overall, the performance of SYCL™ was not as good as expected.

As a next step, possible optimizations for the FPGA could be investigated and implemented, especially in the case of vector addition. Combining oneMKL with the native FPGA implementation for the FPGA could be a way to enhance the performance portability, especially as oneMKL performed well on the GPU and CPU.

If INTEL® provides an oneMKL implementation for the FPGA in the future, a performance portability analysis could be performed again. This work showed that the performance of oneMKL is very good compared to native SYCL™ implementations, so a higher score could be expected.

Bibliography

- [1] I. Baratta, C. Richardson, G. Wells. “Performance analysis of matrix-free conjugate gradient kernels using SYCL”. In: *International Workshop on OpenCL. IWOCL’22*. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1–10. ISBN: 978-1-4503-9658-5. DOI: 10.1145/3529538.3529993. URL: <https://dl.acm.org/doi/10.1145/3529538.3529993> (visited on 04/12/2023) (cit. on p. 15).
- [2] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Du, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. S. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, J. W. V. Gudenberg. “Document for the Basic Linear Algebra Subprograms (BLAS) standard: BLAS Technical Forum”. In: 2001. URL: <https://www.semanticscholar.org/paper/Document-for-the-Basic-Linear-Algebra-Subprograms-Blackford-Corliss/6986ad2142883178202a709c870cb0b690fa5c77> (visited on 03/23/2023) (cit. on p. 25).
- [3] S. Cali, W. Detmold, G. Korcyl, P. Korcyl, P. Shanahan. *Implementation of the conjugate gradient algorithm for heterogeneous systems*. arXiv:2111.14958 [hep-lat]. Nov. 2021. DOI: 10.48550/arXiv.2111.14958. URL: <http://arxiv.org/abs/2111.14958> (visited on 04/12/2023) (cit. on p. 15).
- [4] L. Dagum, R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering 5.1* (Jan. 1998). Conference Name: IEEE Computational Science and Engineering, pp. 46–55. ISSN: 1558-190X. DOI: 10.1109/99.660313 (cit. on p. 23).
- [5] M. Hestenes, E. Stiefel. “Methods of conjugate gradients for solving linear systems”. en. In: *Journal of Research of the National Bureau of Standards* 49.6 (Dec. 1952), p. 409. ISSN: 0091-0635. DOI: 10.6028/jres.049.044. URL: https://nvlpubs.nist.gov/nistpubs/jres/049/jresv49n6p409_A1b.pdf (visited on 03/03/2023) (cit. on p. 27).
- [6] Intel®. *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. en. Dec. 2022 (cit. on p. 18).
- [7] Intel®. *Intel® Arria® 10 Device Datasheet*. en. Feb. 2022 (cit. on p. 21).
- [8] Intel®. *Intel® Arria® 10 Product Table (Gen-1010-2.1)*. en (cit. on p. 21).
- [9] Intel®. *Intel® oneAPI Math Kernel Library*. 2023 (cit. on p. 25).
- [10] Intel®. *Intel® Xeon® Gold 6128 Processor (19.25M Cache, 3.40 GHz) Product Specifications*. en. URL: <https://www.intel.com/content/www/us/en/products/sku/120482/intel-xeon-gold-6128-processor-19-25m-cache-3-40-ghz.html> (visited on 04/03/2023) (cit. on p. 18).

- [11] M. Khalilov, A. Timoveev. “Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU”. en. In: *Journal of Physics: Conference Series* 1740.1 (Jan. 2021). Publisher: IOP Publishing, p. 012056. ISSN: 1742-6596. DOI: 10.1088/1742-6596/1740/1/012056. URL: <https://dx.doi.org/10.1088/1742-6596/1740/1/012056> (visited on 04/12/2023) (cit. on p. 16).
- [12] M. Krainiuk, M. Goli, V. R. Pascuzzi. “oneAPI Open-Source Math Library Interface”. In: *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Nov. 2021, pp. 22–32. DOI: 10.1109/P3HPC54578.2021.00006 (cit. on p. 16).
- [13] NVIDIA. *cuBLAS (Release 12.1)*. Feb. 2023. URL: https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf (cit. on p. 25).
- [14] NVIDIA. *CUDA C++ Programming Guide (Release 12.1)*. Feb. 2023. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (cit. on pp. 19, 24).
- [15] NVIDIA. *NVIDIA QUADRO GP100 Data Sheet*. en. URL: <https://www.pny.com/file%20library/company/support/product%20brochures/nvidia%20quadro/english/pny-nvidia-quadro-gp100-datasheet-mixed-mode-computation.pdf> (cit. on p. 19).
- [16] NVIDIA. *NVIDIA Tesla P100*. en. URL: <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf> (cit. on p. 19).
- [17] OpenMP Architecture Review Board. *OpenMP Application Program Interface (Version 4.0)*. July 2013. URL: <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf> (cit. on p. 23).
- [18] S.J. Pennycook, J.D. Sewall, V.W. Lee. *A Metric for Performance Portability*. arXiv:1611.07409 [cs]. Nov. 2016. URL: <http://arxiv.org/abs/1611.07409> (visited on 03/08/2023) (cit. on p. 29).
- [19] J. R. Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Technical Report. USA: Carnegie Mellon University, 1994 (cit. on p. 27).
- [20] The Khronos® SYCL™ Working Group. *SYCL™ 2020 Specification (revision 6)*. Nov. 2022. URL: <https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf> (cit. on p. 25).

All links were last followed on April 11, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature