PADERBORN UNIVERSITY

DISSERTATION

Bringing Massive Parallelism and Hardware Acceleration to Linear Scaling Density Functional Theory Through Targeted Approximations

by
Michael Laß

A thesis submitted in partial fulfillment of the requirements for the degree of Dr. rer. nat.

in the

Faculty for Computer Science, Electrical Engineering and Mathematics

November 29, 2021

Acknowledgements

This thesis would not have been possible without the support of all the people who accompanied me on this journey.

First of all, I want to thank my supervisor Prof. Dr. Christian Plessl, who prompted my interest in computer engineering and custom computing early during my studies and allowed me to pursue my interests in the High-Performance IT Systems Research Group. Further, I want to thank Prof. Dr. Thomas Kühne, who steered my efforts into the field of quantum chemistry and with that allowed me to tackle a highly relevant research area with my work. I also want to thank the remaining members of the examination committee, Prof. Dr. Sybille Hellebrand, Prof. Dr. Marco Platzner and Prof. Dr. Friedhelm Meyer auf der Heide, for supporting me in the final phase of my doctoral studies.

Research thrives on discussions and exchange. I want to thank all members of the High-Performance IT Systems Research Group for their input and all the fruitful discussions we have had. Especially, I want to thank Dr. Robert Schade for the countless hours of discussion on the topics of quantum chemistry and performance engineering, Dr. Tobias Kenter for providing his invaluable expertise on FPGA design and Dr. Heinrich Riebler for his constant helpfulness and the good atmosphere in our shared office.

I want to thank the technical staff of the Paderborn Center for Parallel Computing for always being supportive and for providing the necessary resources for my research. A special thanks goes to Axel Keller who has been a great office colleague and who was immensely helpful in providing me with the technical resources and support I needed.

A big thanks also goes to all the former and current colleagues at the Paderborn Center for Parallel Computing who I have not mentioned by name. Working with you has always been a joy, just as the shared lunch and coffee breaks.

Finally, I want to thank all members of my family as well as my close friends for being so supportive over the last years. I am endlessly grateful that you are part of my life.

Abstract

Computational sciences play an ever-growing role in many academic disciplines. One particularly active field is computational chemistry where insight into chemical bonds and the structure of molecules and solids is gained by running large computer simulations, driving the development of new substances and materials. These simulations are often based on quantum mechanical calculations which provide high accuracy but are very compute intensive.

To be able to make effective use of today's large high performance computing (HPC) clusters for these simulations, the time consuming computations must be parallelized across thousands of compute nodes. Additionally, utilizing dedicated accelerator hardware such as GPUs and FPGAs becomes more and more important. These accelerators are particularly energy efficient and nowadays provide large parts of the available floating-point performance in many HPC clusters. However, using them effectively poses new requirements on the used methods and their implementation.

This thesis tackles both of these challenges in the context of ab-initio molecular dynamics simulations where the movement of a number of interacting atoms is simulated based on forces computed from first principles using electronic structure methods. In particular, we deal with linear scaling density matrix based density functional theory (DFT) as underlying electronic structure method. To reach the goal of massive parallelism and to make effective use of accelerator hardware, we leverage the idea of Approximate Computing. We introduce targeted approximations to break up dependencies between computations on different compute nodes and exploit low-precision arithmetic in the involved computations.

Specifically, we examine iterative algorithms for two central linear algebra kernels of the considered DFT method, namely matrix inverse p-th roots and the matrix sign function, and demonstrate their resilience against errors introduced by using low-precision arithmetic. Furthermore, to apply these algorithms to large sparse matrices in a massively parallel way, we introduce the *Submatrix Method* as an approximate method to distribute matrix operations on sparse matrices onto a large number of compute nodes and multiple accelerator devices. By partitioning the input matrix into submatrices and resolving dependencies between computations on these submatrices, the Submatrix Method provides a high level of parallelism and allows to easily distribute workload among many processing cores as well as any present accelerator devices. As a result, the computations can make effective use of large, heterogeneous HPC clusters and their different accelerators.

We demonstrate the practical integration of the presented methods in the open source quantum chemistry code CP2K. Doing so, we are able to integrate both GPUs and FPGAs in a highly scalable way and allow trading off accuracy of the results against the floating-point performance achieved on these accelerators. This lays the foundation to efficiently run ab-initio molecular dynamics simulations on modern, heterogeneous HPC systems and to apply them to larger molecules.

Zusammenfassung

Rechnergestützte Wissenschaften spielen in vielen akademischen Disziplinen eine zunehmend wichtige Rolle. Dies gilt besonders im Bereich der Computerchemie. In diesem werden mit Hilfe von großen, computergestützten Simulationen Einblicke in chemische Bindungen und den Aufbau von Molekülen und Festkörpern gewonnen und die Entwicklung neuer Stoffe und Materialien vorangetrieben. Oft basieren diese Simulationen auf quantenmechanischen Rechnungen, die die Realität gut abbilden, jedoch sehr rechenintensiv sind.

Um die heutigen großen Hochleistungsrechner für solche Simulationen effektiv auszunutzen, müssen die zeitaufwändigen Rechnungen auf Tausenden von Rechenknoten parallelisiert werden. Außerdem gewinnt die Nutzung spezieller Hardwarebeschleuniger, wie GPUs und FPGAs, zunehmend an Bedeutung, da diese besonders energieeffizient arbeiten und heutzutage in vielen Hochleistungsrechnern einen großen Anteil der verfügbaren Fließkomma-Rechenleistung bereit stellen. Diese Beschleuniger effektiv zu nutzen stellt jedoch neue Anforderungen an die verwendeten Methoden und deren Implementierung.

Diese Arbeit behandelt diese beiden Herausforderungen im Kontext von Ab-Initio Molekulardynamik-Simulationen. In diesen wird die Bewegung von interagierenden Atomen auf Basis von Kräften simuliert, die mit Hilfe von Elektronenstrukturmethoden ausgehend von grundlegenden physikalischen Gesetzen berechnet werden. Im Speziellen behandelt diese Arbeit linear skalierende Dichtematrix-basierte Dichtefunktionaltheorie (DFT) als zugrunde liegende Elektronenstrukturmethode. Um das Ziel einer massiv parallelen Ausführung zu erreichen, und die effektive Nutzung von Hardwarebeschleunigern zu ermöglichen, verfolgen wir den Ansatz von Approximate Computing. Wir führen gezielt Approximationen ein um Abhängigkeiten zwischen Rechnungen auf verschiedenen Rechenknoten aufzulösen und führen Rechnungen mit geringer Präzision aus.

Im Detail untersuchen wir iterative Algorithmen zur Berechnung zweier Funktionen aus der linearen Algebra, die in der betrachteten DFT-Methode elementar sind – die Berechnung inverser p-ter Wurzeln sowie der Signumfunktion von Matrizen – und demonstrieren ihre Robustheit gegen Fehler, die durch Rechnungen mit geringer Präzision eingeführt werden. Um diese Algorithmen für dünn besetzte Matrizen außerdem massiv zu parallelisieren führen wir die *Submatrix-Methode* ein. Diese approximative Methode verteilt Operationen auf dünn besetzten Matrizen auf eine große Anzahl von Rechenknoten und Hardwarebeschleuniger. Die Eingabematrix wird dabei in Submatrizen zerlegt und Abhängigkeiten zwischen Rechnungen auf diesen Submatrizen aufgelöst, sodass ein hoher Grad an Parallelität und eine einfache Verteilung von Rechnungen auf viele Prozessorkerne sowie vorhandene Hardwarebeschleuniger erreicht werden. Im Ergebnis können diese Rechnungen somit große, heterogene Hochleistungsrechner sowie ihre verschiedenen Hardwarebeschleuniger effektiv nutzen.

Wir demonstrieren die praktische Integration der vorgestellten Methoden in den Open Source Quantenchemie-Code CP2K. Dadurch sind wir in der Lage, sowohl GPUs als auch FPGAs in einer hoch skalierbaren Art einzubinden und erlauben eine Abwägung zwischen der Genauigkeit der Ergebnisse und der Fließkomma-Rechenleistung, die auf diesen Beschleunigern erreicht werden kann. Damit wird die Grundlage geschaffen, Ab-Initio Molekulardynamik-Simulationen effizient auf modernen, heterogenen Hochleistungsrechnern auszuführen und für die Simulation größerer Moleküle zu verwenden.

Table of Contents

Chapter 1

Introduction

Since the dawn of electronic computers in the mid 20th century, there has been an immense increase in computing power and with that, computers became more and more part of our lives. Certainly most visible are consumer electronics, the abundance of web services and the automation of everyday processes, such as banking, accounting, etc. However, often only visible to scientists working with these systems, is the ever growing importance of large-scale computer simulations in scientific research. Examples for such scientific computing applications are climate simulations, atomistic simulations, the simulation of electromagnetic fields, biological processes such as protein folding, fluid dynamics and many more. These simulations provide us with weather forecasts and advisories, drive the development of entirely new materials and technologies, help understanding diseases and allow the development of new drugs.

One of the most prevalent research areas on today's High-Performance Computing (HPC) clusters is chemistry and material science. To understand how materials behave under certain conditions and how molecules form and interact, large atomistic simulations are performed. This *computational chemistry* is nowadays an important connecting link between theory and experiment in that it allows to perform and closely observe experiments in a virtual laboratory. One category of atomistic simulations is Ab-Initio Molecular Dynamics (AIMD), where the movement of atoms and their interaction is simulated from first principles, which means that the simulation is entirely based on quantum mechanical electronic structure methods such as Density Functional Theory (DFT). While this kind of simulation provides a high level of accuracy, it is computationally demanding and typically only allows to simulate systems of thousands or in the case of special Linear Scaling DFT (LSDFT) methods up to a million atoms.

To cope with the ever growing demand for computational power, increasingly large HPC clusters are required, driving the cost of these systems and therefore scientific computing overall. At the same time, the systems need to become more energy efficient in order to keep energy and cooling costs manageable. One visible trend to increase the energy efficiency is to make use of specialized hardware accelerators, such as GPUs or Field Programmable Gate Arrays (FPGAs). To efficiently scale applications over increasingly large clusters and to make use of hardware accelerators are major challenges when developing HPC methods and codes.

1.1 Contributions

In this thesis, we tackle both of these challenges in the context of one particular LSDFT method which is *density matrix based DFT*. It is based around the so-called matrix sign function which in the context of large AIMD simulations needs to be computed on large, sparse matrices that are stored in a distributed fashion on all compute nodes. Next to the matrix sign function, one other particularly important kernel is the computation of inverse square roots of matrices, or more generalized, the computation of inverse matrix p-th roots. We leverage the concept of Approximate Computing (AC) in that we allow small approximation errors in the computed results in order to increase performance and scalability of the involved algorithms and make them suitable for acceleration on GPUs and FPGAs. The main contributions of this thesis are:

1. We study the effect of using low-precision floating-point arithmetic and storage for iterative methods that are commonly used for computation of inverse matrix p-th roots and the matrix sign function. We evaluate the resulting errors using real-world input data and application-specific error metrics to specifically evaluate the use of low-precision arithmetics in the context of Linear Scaling DFT. This lays the foundation for using high-performance low-precision accelerators such as modern GPUs and FPGAs for the computation of these matrix functions.
2. We present the *Submatrix Method* as a general method to compute approximate results for unary matrix functions for large, sparse input matrices in a highly parallel fashion. It transforms the computation of a function on a large, sparse input matrix into independent computations of this function on a set of much smaller, dense submatrices. These computations can then be distributed over many compute nodes or accelerator devices, making the overall method highly scalable and efficient.
3. We describe and evaluate an implementation of the Submatrix Method in the open source code CP2K. In this way, we not only show how practical implementation challenges can be overcome, but also demonstrate the practicality of using the Submatrix Method in the context of Linear Scaling DFT. Our submatrix-based approach shows to be highly scalable and, if accuracy requirements do not exceed a certain limit, shows favorable performance compared to the previously used approach. The presented approach and our implementation has been included in CP2K.
4. We demonstrate the computation of the sign function for the generated submatrices using GPUs and FPGAs and evaluate both performance and energy efficiency for these implementations. For the GPU accelerator, we additionally evaluate using low or mixed precision arithmetic in order to fully exploit the capabilities of these devices. For FPGA acceleration, we describe in detail how the discussed iterative methods are implemented on an FPGA.

1.2 Thesis Structure

The thesis is structured as follows: In Chapter 2, we lay the foundation for the thesis contents by introducing basic terms and principles from the areas of High-Performance Computing, Approximate Computing and basic linear algebra. Chapter 3 provides a brief introduction into Ab-Initio Molecular Dynamics and Density Functional Theory, as well as the density matrix based DFT method that is the target of this work and its implementation in CP2K. Chapter 4 covers the effect of low-precision arithmetic and storage onto the considered iterative schemes. In Chapter 5, the Submatrix Method is presented and evaluated for different use cases, including but not limited to Linear Scaling DFT. The implementation of this method in CP2K as well as the evaluation of this implementation is presented in Chapter 6. Offloading the submatrix operations to accelerator devices is covered in Chapter 7. Finally, Chapter 8 summarizes the results of our work and concludes this thesis.

Chapter 2

Foundations

In this chapter we want to lay the foundation for the topics presented in this work. In particular, we describe the basic concepts of High-Performance Computing (HPC) and GPUs and FPGAs as accelerator technology in Section 2.1. We present the idea of Approximate Computing (AC) which drives the concepts of this work in Section 2.2 and introduce necessary concepts of linear algebra in Section 2.3.

2.1 High-Performance Computing Systems and Applications

High-Performance Computing clusters, or *supercomputers*, are optimized towards the workload that the applications from computational sciences pose. These typically contain large amounts of floating-point calculations on large datasets. To overcome the limitations of a single computer, supercomputers are in fact large clusters of compute nodes that communicate using a high-performance interconnect. Each node within the cluster provides powerful processors with a high core count and support for wide floating-point vector instructions, in order to maximize data parallelism already within a single node. Another focus lies on high memory bandwidth to be able to provide the processors with enough data to feed their compute pipelines. The nodes are connected using specialized networking technologies such as InfiniBand, which provide much lower latency than typical Ethernet networks.

The 500 most powerful publicly known supercomputers in terms of floating-point operations per second (FLOP/s) are regularly ranked in the TOP500 list [1]. As of writing this thesis, the most powerful system in this list is the *Fugaku* supercomputer in Japan which in total provides 7,630,848 CPU cores, providing a peak performance of 537 PFLOP/s. Apart from raw compute power, energy efficiency becomes more and more important nowadays. In the GREEN500 list [2], the most energy efficient systems in terms of GFLOP/J are ranked. Here, Fugaku only reaches 26th place with around 15 GFLOP/J while the top of the list reaches 39 GFLOP/J. Key to this high efficiency seems to be the use of accelerator devices, as the first 21 systems in the list use either GPUs or special-purpose chips in addition to regular CPUs.

Applications running on supercomputers must be able to distribute their data and computational load across the available nodes in an efficient manner to be able to exploit the available computational resources. This requires focusing on a high level of parallelism when developing methods and algorithms that are supposed to run on these machines. Optimally, they should be able to scale over thousands of nodes without communication and data exchange becoming a bottleneck. To be able to run on particularly energy-efficient clusters, applications need to be able to offload large parts of their computations onto accelerator devices, such as GPUs, FPGAs or special purpose accelerators.

To allow programming applications with this kind of parallelism, there are various established libraries and programming interfaces. One of these libraries is the Message Passing Interface (MPI) [3] which allows implementing communication in distributed memory systems, i.e., clusters comprised of numerous independent compute nodes. It supports different communication schemes like point-to-point communication and collective routines which exchange data between multiple nodes. It also supports Remote Direct Memory Access (RDMA) operations, where data can be directly loaded from or stored to memory of other nodes, without involving the remote processor. Another widely used framework is OpenMP [4], which is a pragma-based language extension to C, C++ and Fortran. It focuses on shared-memory parallelism through threading and data-level parallelism using the processor's vector instructions. The latter is also referred to as Single Instruction Multiple Data (SIMD) because vectors of multiple data elements are processed using a single instruction. OpenMP also provides basic support for GPU programming in that code can be annotated to be automatically offloaded to GPUs.

2.1.1 HPC Clusters Used in This Work

Throughout this work, concepts and implementations will be evaluated on two HPC clusters hosted at the Paderborn Center for Parallel Computing[1].

OCuLUS

The *OCuLUS* cluster has been installed in 2013. It features different classes of compute nodes with varying features. In total, it contains of 619 nodes that are connected by a 40 Gbit/s InfiniBand network and have access to 500 TB of shared storage [5]. In this work, we use two kinds of nodes of the OCuLUS cluster:

Compute Nodes contain two Intel Xeon E5-2670 CPUs with a total of 16 Sandy Bridge EP cores with a base clock of 2.6 GHz. The nodes are equipped with 64 GiB of DDR3 main memory.

GPU Nodes additionally contain an NVIDIA RTX 2080 Ti [6] GPU card equipped with 11 GiB of on-board GDDR6 memory. It is connected to the host via PCIe-3.0 x16.

Noctua 1

The *Noctua 1* cluster has been inaugurated in 2018. Its 272 nodes are connected via a 100 Gbit/s Intel Omni-Path network and have access to 720 TB of shared storage. In total, it contains 10,880 CPU cores and 51 TiB of main memory and has been benchmarked with a total of 535 TFLOP/s [7]. The cluster contains two different kinds of nodes which are both used in this work:

Compute Nodes contain two Intel Xeon Gold 6148 CPUs with a total of 40 Skylake SP cores with a base clock of 2.4 GHz. The nodes are equipped with 192 GiB of DDR4 main memory.

FPGA Nodes are based around two Intel Xeon Gold 6148F CPUs and additionally contain two BittWare 520N [8] cards connected via PCIe-3.0 x8. These cards each contain an Intel Stratix 10 GX 2800 FPGA and 32 GiB of on-board DDR4 memory.

[1]https://pc2.uni-paderborn.de/

Table 2.1: Overview over different floating-point types. The table lists the number of stored mantissa bits, not counting the implicit bit. Each data type contains an additional sign bit.

Name	Total Bits	Exponent Bits	Mantissa Bits
Double-Precision (FP64)	64	11	52
Single-Precision (FP32)	32	8	23
TensorFloat-32 (TF32)	19	8	10
Half-Precision (FP16)	16	5	10
Brain Floating Point (bfloat16)	16	8	7

2.1.2 GPUs as Accelerator Platform in HPC

GPUs have their roots in the acceleration of 3D graphics applications. They became interesting for general computational workloads such as fast linear algebra when they started to provide support for floating point arithmetic. While in the beginning any computational problem needed to be formulated using graphic primitives, NVIDIA in 2006 released the CUDA programming model [9] that allows general purpose programming for their own GPUs. In 2009 OpenCL [10] was introduced as a vendor independent language and framework for programming GPUs. Both CUDA and OpenCL are currently widely used for GPU programming with CUDA being limited to NVIDIA devices.

Generally, a GPU can be considered as a collection of multi-threaded SIMD processors, i.e., processors that support running many hardware threads in parallel where each of these threads is a sequence of SIMD instructions [11, Ch. 4.4]. Conceptually this is not entirely different from modern multi-core CPUs that typically support hardware multi-threading and SIMD instructions as well. However, GPU architectures are optimized for high throughput of many concurrent and independent calculations. In particular, the number of processors and the number of hardware threads running on each processor are typically much higher than on a general purpose CPU. Additionally, the GPU processing cores focus entirely on the execution of SIMD operations. Conditions and branches can be executed without jumping between different instructions by using predication and mask registers. To support a high number of concurrent hardware threads, each SIMD lane features a large number of registers that are divided among the running threads.

To better suit machine learning and in particular deep neural network applications that gained much attention over the last years, GPUs for high-performance applications have gained special support for low-precision matrix multiplications. For example, NVIDIA GPUs provide specialized *tensor cores* that perform Multiply-ACcumulate (MAC) operations on matrix blocks in different precisions, such as double-precision (FP64), single-precision (FP32), half-precision (FP16) and more specialized floating-point types such as bfloat16 (BF16) and TensorFloat-32 (TF32) as well as different integer types (INT8, INT4, BINARY) [12]. An overview over different floating-point data types is given in Table 2.1. Mixed-precision multiply-accumulate where multiplications are performed in low precision (FP16, BF16, TF32) and accumulation is performed using single-precision are supported as well. Utilizing these tensor cores, especially using low-precision, allows to fully exploit the performance of these GPUs. For example, the NVIDIA A100 [12] GPU has a peak performance of 9.7 TFLOP/s when using conventional FP64 arithmetic. Utilizing tensor cores raises this number to 19.5 TFLOP/s (FP64), 156 TFLOP/s (FP32) and

312 TFLOP/s (FP16, BF16) and even up to 624 TOP/s (INT8) and 1,248 TOP/s (INT4) when performing integer instead of floating-point arithmetic.

Programming GPUs in CUDA or OpenCL is done on the level of single SIMD lane operations (referred to as *thread* in CUDA or *work item* in OpenCL) and groups of these operations that are executed on the same processor and therefore have access to a shared local memory (referred to as *thread block* in CUDA or *work group* in OpenCL). Combining these operations into threads of SIMD instructions (referred to as *warp* in CUDA) is then performed by the compiler. Scheduling the thread blocks on the different processors is performed at runtime by the GPU's thread block scheduler.

Due to the different programming models and terminologies compared to CPU programming, and due to the need to optimize algorithms and code towards the architecture of GPUs, utilizing the full potential of GPUs can be challenging. To ease the development for GPUs, pragma-based programming standards such as OpenMP and OpenACC [13] have been developed that allow offloading computationally expensive parts of code to GPUs without explicitly writing CUDA or OpenCL code. While this removes the burden of learning entirely new programming models, a good understanding of the underlying architecture is still required to achieve good performance. Over time, a number of libraries has been published that can be used to utilize GPUs without the need for any custom GPU programming. Many computational tasks from high performance computing applications have been implemented, such as dense and sparse linear algebra and FFTs. In the context of this work, we will make use of NVIDIA's cuBLAS [14] library that implements basic dense linear algebra on NVIDIA GPUs.

2.1.3 FPGAs as Accelerator Platform in HPC

With GPUs being the most prevalent type of accelerators in HPC systems, providing high floating-point throughput for highly parallel computations, application-specific accelerators can be found as well. While these may offer the best possible performance for a certain application, the development and production of Application-Specific Integrated Circuits (ASICs) comes with high costs and is therefore commercially suitable only for widely used applications, with machine learning being the most dominant example in the current HPC landscape.

Another promising hardware architecture for accelerators are Field Programmable Gate Arrays (FPGAs). These are also referred to as *programmable hardware*, as they can be programmed by the developer to behave like any arbitrary digital circuit. This allows profiting of a special-purpose hardware accelerator without the need of going through the development and production of completely custom ASICs. Instead, they can be bought readily available from vendors, together with the required software for programming.

Structure

FPGAs are available from different vendors, with Xilinx and Intel being the two most dominant. While the hardware architecture slightly differs, FPGAs from both vendors basically contain the following elements:

- Lookup Tables (LUTs) that can be used to model arbitrary combinatorial logic,
- registers or Flip Flops (FFs) for data storage and to implement synchronous logic,

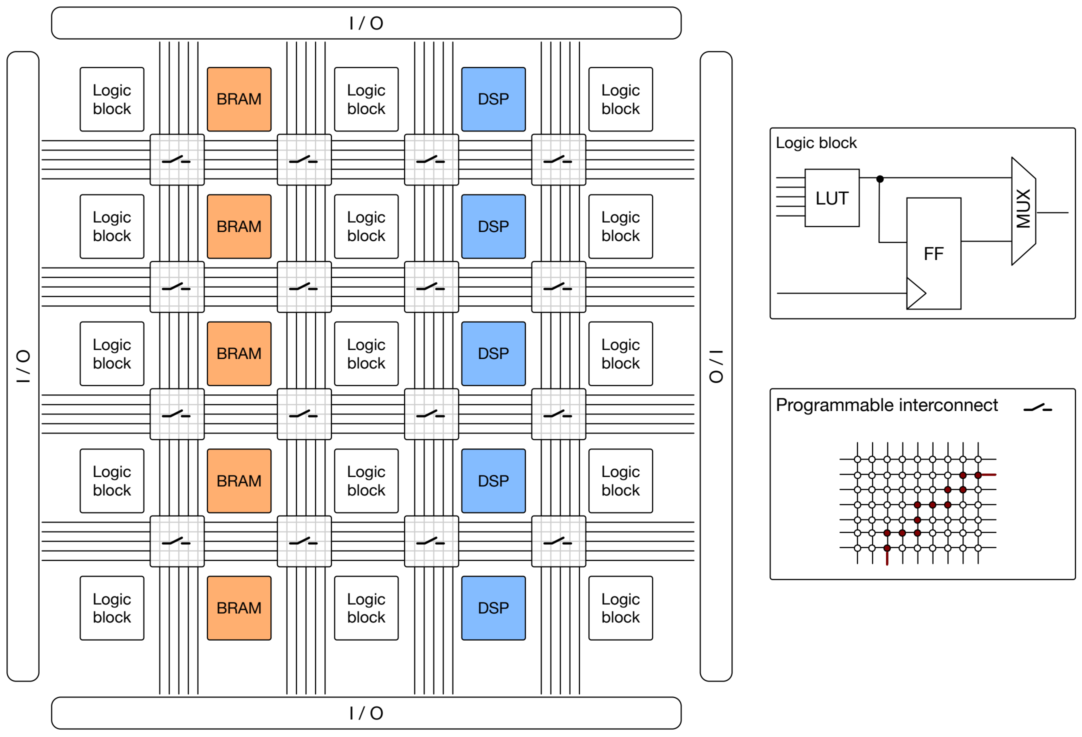

Figure 2.1: Simplified illustration of the typical architecture of an FPGA.

- a programmable interconnect that can be configured to route signals between different logic elements,
- dedicated SRAM blocks (referred to as BRAM) that can hold several kilobytes each,
- specialized Digital Signal Processing (DSP) blocks used for arithmetic operations,
- Phase-Locked Loops (PLLs) to generate clock signals and dedicated clock networks to route them across the chip, and
- I/O banks and serial transceivers to communicate with peripherals.

A simplified illustration of the architecture of an FPGA is shown in Figure 2.1. For the use of FPGAs in HPC systems, the FPGA is typically combined with additional memory (DRAM or High Bandwidth Memory) on a PCIe board that can be installed in the compute nodes of a cluster.

The single elements of the chip can be combined to build larger structures and are often configurable in itself. One noticeable difference between the two main vendors lies in the configurability of the DSP blocks. With respect to floating-point operations, on current Intel Stratix 10 FPGAs a single DSP block performs one MAC operation in single-precision [15]. On current Xilinx UltraScale+ devices, a DSP block works on two inputs of 27 bit and 18 bit length, such that multiple blocks or a block and additional logic need to be combined to perform a single-precision MAC operation [16]. While the latter architecture seems less optimized for single-precision computations, it provides more flexibility in input data widths and in choosing the resources used for arithmetic operations.

Capabilities

One of the main advantages of FPGAs is the possibility to physically distribute the logic across the chip and therefore allow highly parallel execution. For example, independent iterations of a loop do not have to be computed sequentially by a processing core but instead can be unrolled in space and computed in parallel using different logic elements of the FPGA. To deal with data dependencies, deep computing pipelines can be constructed, where data is processed in ten thousands of consecutive pipeline stages, avoiding the need for any memory loads and stores.

Due to these possibilities, FPGAs are most common in streaming-based applications, where data is constantly streamed into the FPGA, processed by the application-specific pipeline and results are constantly streamed out. However, while this kind of application is a strong case for using FPGAs, they are also well suited in other applications. For example, we have demonstrated highly efficient execution of a branch and bound search algorithm on FPGAs [A7], utilizing the flexibility of FPGAs by generating problem-specific hardware designs for each concrete search problem. Others have successfully used FPGAs to implement finite difference methods and finite element methods to solve Maxwell's equations [17, 18] and to implement shallow water simulations [19]. Large FPGAs that are specifically tailored towards HPC environments also provide high floating-point compute capabilities. However, creating designs that achieve near peak floating-point performance is often challenging because designs need to utilize most of the DSP blocks in each clock cycle and run at high frequencies.

Development Models

Traditionally, FPGAs are programmed using Hardware Description Languages, such as Verilog, SystemVerilog and VHDL. These languages provide some abstraction over the concrete logic design in that circuits can be described based on their behavior. Tools provided by the vendor then synthesize a hardware netlist from these descriptions, map parts of these netlists to components available on the FPGA, finally allocating specific elements on the FPGA and generating a routing to connect all elements accordingly. However, the behavioral description is still on a low level, requiring precise description of combinatorial logic and clock synchronous logic.

To allow for an easier development process and to allow the design of more complex logic on large FPGAs, higher-level abstractions and High-Level Synthesis (HLS) tools are nowadays very common. Here, the logic behavior is automatically synthesized from a software-like description in C, C++ or OpenCL. There are also domain specific languages such as Maxeler's *MaxJ* that is a Java-based language used to describe data flow engines that are then synthesized into FPGA designs [20]. Apart from the much more accessible development flow that is provided by HLS tools, they can often generate deeply pipelined designs that allow high clock rates and that would be barely possible to design by hand.

However, even when using these high level abstractions, the developer still needs to guide the design tools with pragmas or similar code annotations. For example, unrolling factors for loops can be specified to control the number of loop replicas and therefore the amount of parallelism in the FPGA design. Additionally, the developer has to closely examine compiler reports to identify any performance obstacles in the design, locate opportunities for further optimization and to scale the logic depending on the available resources. Due to the fact that hardware synthesis often takes hours or even days, this can be a very time consuming process.

2.2 Approximate Computing

Calculations with limited precision and the use of approximate algorithms are not uncommon in computing. In fact, all of the commonly used floating-point data types shown in Table 2.1 on page 7 have inherently limited precision. Approximation algorithms allow finding solutions to optimization problems that are not necessarily the best solution but good enough.

In recent years, Approximate Computing (AC) has gained attention as a paradigm to specifically exploit approximations and imprecise calculations in order to increase the performance or energy efficiency of computing systems [21, 22]. The approximations can thereby reach across all parts of the computing stack, i.e., down from the design of logic circuits and their operating conditions up to the running application with its internal algorithms and data structures. Optimally, an AC system would exploit opportunities for approximation on all these levels of the computing stack in a way to maximize performance or efficiency while retaining the functionality of the application. Working towards this goal makes AC a very wide and complex research field, requiring understanding of the application domain, the involved algorithms as well as the underlying computer architecture. In the following, goals, techniques and typical areas of application are briefly presented.

2.2.1 Goals and Restrictions

Approximate Computing can be used to optimize a system towards different goals. One of these goals is performance, i.e., minimizing the time required to come to a solution (*time to solution*) or growing the maximum problem size that can be handled in a certain amount of time. Other possible goals are to minimize power or energy consumption of a system. Since energy consumption is determined as a product of the time to solution and the average power consumption, it can often be reduced by optimizing towards performance or power consumption. Naturally, power and energy consumption have always played an important role in the design of mobile systems. However, nowadays these two factors become more and more important also in large-scale computing systems such as HPC clusters. Due to increasing energy costs and increasing cooling effort, these systems require operators and users to make the best possible use of the consumed electrical energy.

Next to performance, power and energy efficiency considerations, also hardware area constraints can be a reason to consider approximations in a systems and application design. For example, when designing ASICs or programming FPGAs, the available logic resources are limited. In addition, large and complex logic designs can make routing difficult and thus lower the achievable clock rates of the design.

Calculations cannot be approximated arbitrarily as the final result still has to be acceptable. However, what result accuracy can be deemed acceptable and how large the final error is for a given type of approximation in the involved calculations is very application specific. In fact, it can also heavily depend on the input data given to that application. Error metrics used to assess the quality of results need to be chosen accordingly. Certain applications may also allow to refine approximate results if necessary or repeat calculations where the resulting error exceeds a certain threshold, allowing more aggressive approximations.

2.2.2 Applications

Certain kinds of applications are predestined to profit from approximations [23]. One group of applications are those where the output is targeted at human perception which is often incapable of noticing small variations or errors. Examples are image, audio and video processing algorithms. In fact, lossy compression techniques, which are common in this area of application, can be considered as approximate storage. Another area are applications that process data from sensors that is inherently noisy or output data to actors that have limited accuracy. Here, small approximation errors may be entirely hidden by noise that is inherent to the input or output of the system. A last category of typical candidates are applications where no golden output exists or we cannot determine this golden result in a practical amount of time. Examples are heuristics, machine learning and approximation algorithms. Here we already accept results that are good enough, so considering the introduction of further approximations seems natural.

Next to the kind of application, also the internally used algorithms can provide opportunities for approximation. In particular, iterative algorithms may be able to correct errors introduced in one iteration in any subsequent iteration. We will look at this potential more closely in Chapter 4.

So far, scientific applications are rarely considered a typical target of AC. However, they often exhibit some of the aforementioned properties. In particular, they often already use certain approximations in their underlying models and might use random or noisy input data. Scientific codes may also use iterative algorithms to perform their computations. For these reasons we consider AC a promising paradigm also for HPC environments.

2.2.3 Levels of and Techniques for Approximation

As Approximate Computing covers all parts of the compute stack, manifold techniques to exploit approximations have been proposed over time. Here we only give a coarse overview to convey the width of the field.

On the bit level of computations, a natural approach is to lower the precision of used data types and provide support for such low-precision computations in hardware. In fact, over the last years we have seen increasing support for low-precision floating-point types such as half-precision or bfloat16 in hardware, mainly to accelerate inference in neural networks. However, also more unconventional paths can be chosen. For example, arithmetic circuits such as adders and multipliers can be designed in a way to require fewer logic elements but introduce small errors for certain inputs [24, 25]. Logic circuits may also be overclocked or operated at supply voltages that are lower than what is required to guarantee correct operation (*voltage over-scaling*) [26, 27]. Another low level approach is to use entirely different computing paradigms such as analog computing [28] or stochastic computing [29] where numbers are not represented by fixed bit patterns but either by analog voltage levels or stochastic bit streams.

On the application level, entirely new approximate methods and algorithms can be developed that partly or entirely replace accurate computations. One approach that does not require explicit modelling of the approximation is to train and use neural networks to replace compute expensive parts of the application [30]. Many of these approaches are highly experimental and are difficult to apply in productive HPC environments. In this work, we therefore focus on the use of low-precision data types and the explicit design of approximate methods. Generally, low-precision data types can be used for any arithmetic calculations, for the storage of data or both.

Approximate Arithmetic

Approximating the arithmetic operations using low precision can be beneficial for overall performance or the energy efficiency, depending on the underlying compute platform. As discussed in Section 2.1.2, recent GPUs targeting the data center, such as the NVIDIA A100, support low-precision data types such as half-precision or bfloat16, often doubling the peak performance compared to single-precision arithmetic and in case of the A100 even increasing performance sixteenfold compared to double-precision arithmetic.

For custom hardware accelerators, the data width also plays an important role. Reduced precision can significantly lower resource requirements on FPGAs, e.g. for current Xilinx devices using less than 17 stored mantissa bits reduces the number of required DSPs by half compared to single-precision. For custom CMOS designs it has been shown that the power consumption of multipliers rises at least quadratically with the number of input bits [31]. Because the delay increases too, this has an amplified effect on the energy consumption. Using fixed-point arithmetic with only few bits can further simplify custom designs for FPGAs or ASICs.

Approximate Storage

In scientific applications, often large amounts of data need to be stored, communicated and processed. This not only leads to great computational demands but also requires large memories and high memory bandwidth. Even if calculations are performed precisely, the required memory space and bandwidth can be reduced by storing intermediate results as well as the input data itself with less precision, using fewer bits. If the computations are to be performed on special hardware accelerators such as GPUs or FPGAs, the data needs to be transferred between the host and these devices. In this case, bandwidth quickly becomes a limiting factor, motivating the use of low-precision representations for the transferred data.

We will consider both scenarios, approximate arithmetic and approximate storage, in Chapter 4 for two linear algebra algorithms.

2.3 Linear Algebra Basics and Definitions

In the following, we define fundamental terms from the field of linear algebra and briefly recollect basic rules that are relevant in the context of this thesis. After that, two particularly important matrix functions, the inverse p-th root and the sign function, are presented in detail.

2.3.1 Fundamental Terms

Definition 2.1. We call a set of vectors *linearly independent*, if none of the vectors can be represented as a linear combination of other vectors in the set.

Definition 2.2. We call two vectors $\vec{a}$ and $\vec{b}$ with nonzero length *orthogonal* if their dot product is zero, i.e., $\vec{a} \cdot \vec{b} = 0$. A set of vectors is orthogonal if all vectors in the set are pairwise orthogonal. If vectors are orthogonal and additionally have a norm of 1, we call them *orthonormal*.

Definition 2.3. With $\boldsymbol{I}$ we denote the *identity matrix* defined as

$$\boldsymbol{I} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

Definition 2.4. If for a square matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ we can find a matrix $\boldsymbol{B} \in \mathbb{C}^{n \times n}$ for which

$$\boldsymbol{AB} = \boldsymbol{BA} = \boldsymbol{I}$$

holds, we call $\boldsymbol{A}$ invertible and $\boldsymbol{B} = \boldsymbol{A}^{-1}$ its inverse.

Definition 2.5. If for a matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ it holds that $\boldsymbol{AA} = \boldsymbol{I}$, i.e., $\boldsymbol{A}$ is its own inverse, we call $\boldsymbol{A}$ *involutory*.

Definition 2.6. For a matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ we define the *trace* of $\boldsymbol{A}$ as

$$\operatorname{Tr}(\boldsymbol{A}) = \sum_{i=1}^{n} a_{ii}.$$

Definition 2.7. For a matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ we call the matrix given by transposition of the matrix and complex conjugation of all elements its *conjugate transpose* or *Hermitian transpose* and write $\boldsymbol{A}^H$ or $\boldsymbol{A}^*$.

Definition 2.8. If for a matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ it holds that $\boldsymbol{A} = \boldsymbol{A}^\mathsf{T}$, we call $\boldsymbol{A}$ *symmetric*. If for a matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ it holds that $\boldsymbol{A} = \boldsymbol{A}^H$, we call it *Hermitian*.

Definition 2.9. A Hermitian matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ is called *positive-definite* if for all vectors $\vec{x} \in \mathbb{C}^n \setminus \{0\}$ it holds that $\vec{x}^* \boldsymbol{A} \vec{x} > 0$. It is called *positive semi-definite* if for all vectors $\vec{x} \in \mathbb{C}^n$ it holds that $\vec{x}^* \boldsymbol{A} \vec{x} \geq 0$.

Definition 2.10. For a square matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ we call vectors with nonzero length $\vec{v}_i$ for which

$$\boldsymbol{A}\vec{v}_i = \lambda_i \vec{v}_i,$$

holds *(right) eigenvectors*. The corresponding λ_i is called the *corresponding eigenvalue*.

Definition 2.11. If a square matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ has n linearly independent eigenvectors, it is called *diagonalizable* and can be represented as

$$\boldsymbol{A} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^{-1},$$

where $\boldsymbol{Q} = (\vec{v}_1 \vec{v}_2 \cdots \vec{v}_n)$ is the matrix constructed from the eigenvectors and

$$\boldsymbol{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \lambda_n \end{pmatrix}$$

is a diagonal matrix with the corresponding eigenvalues on its diagonal. This particular decomposition of $\boldsymbol{A}$ is called *eigendecomposition*. If $\boldsymbol{A}$ is symmetric, then orthonormal eigenvectors can be chosen such that $\boldsymbol{Q}^{-1} = \boldsymbol{Q}^\mathsf{T}$. Real symmetric matrices are always diagonalizable. If $\boldsymbol{A}$ is diagonalizable and all eigenvalues of $\boldsymbol{A}$ are nonzero, it is also invertible and its inverse can be computed by inverting the individual eigenvalues in $\boldsymbol{\Lambda}$:

$$\begin{aligned} \boldsymbol{A}^{-1} &= \left(\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}\right)^{-1} \\ &= \boldsymbol{Q}\boldsymbol{\Lambda}^{-1}\boldsymbol{Q}^{-1} \end{aligned}$$

Definition 2.12. For a matrix $\boldsymbol{A} \in \mathbb{C}^{m\times n}$, we define the following matrix norms:

$$\begin{aligned} \|\boldsymbol{A}\|_1 &= \max_{1\le j\le n} \sum_{i=1}^{m} \left|a_{ij}\right|, \\ \|\boldsymbol{A}\|_\infty &= \max_{1\le i\le n} \sum_{j=1}^{m} \left|a_{ij}\right|, \\ \|\boldsymbol{A}\|_2 &= \sqrt{\lambda_{\max}(\boldsymbol{A}^*\boldsymbol{A})}, \\ \|\boldsymbol{A}\|_F &= \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} \left|a_{ij}\right|^2}, \end{aligned}$$

where $\lambda_{\max}$ denotes the largest eigenvalue. $\|\boldsymbol{A}\|_2$ is referred to as *spectral norm*. $\|\boldsymbol{A}\|_F$ is referred to as *Frobenius norm*.

2.3.2 Matrix Powers and (Inverse) p-th Roots

As for scalar values, matrix powers with integer exponents can be seen as multiple matrix multiplications, i.e.,

$$A^n = \underbrace{A \cdot A \cdot \ \ldots \ \cdot A}_{n \text{ times}}.$$

Based on this, we can also define *matrix roots*:

Definition 2.13. Let $\boldsymbol{B}$ be a matrix such that $\boldsymbol{B}^p = \boldsymbol{A}$. Then we call $\boldsymbol{B}$ a p-th root of $\boldsymbol{A}$.

However, this is not a unique definition for $\boldsymbol{B}$. This can easily be seen by the fact that all involutory matrices are square roots of the identity matrix $\boldsymbol{I}$. We therefore literally adopt the definition by Higham [32, Theorem 7.2] for a unique *principal p-th root*:

Definition 2.14. Let $\boldsymbol{A} \in \mathbb{C}^{n\times n}$ have no eigenvalues on $\mathbb{R}^-$. There is a unique p-th root $\boldsymbol{B}$ of $\boldsymbol{A}$ all of whose eigenvalues lie in the segment $\{z : -\pi/|p| < \arg(z) < \pi/|p|\}$, and it is a primary matrix function of $\boldsymbol{A}$. We refer to $\boldsymbol{B}$ as the principal p-th root of $\boldsymbol{A}$ and write $\boldsymbol{B} = \boldsymbol{A}^{1/p}$. If $\boldsymbol{A}$ is real then $\boldsymbol{A}^{1/p}$ is real.

Definition 2.15. For a matrix $\boldsymbol{B} = \boldsymbol{A}^{-1/p}$ with $p > 0$ we use the term *inverse p-th root* of $\boldsymbol{A}$.

Note that, to avoid any ambiguity with the inverse of the root function (i.e., the matrix power), a more distinct term would be the *reciprocal p-th root*. However, the term *inverse p-th root* is commonly used in literature and therefore also throughout this thesis.

Calculation of Inverse Matrix p-th Roots

For the calculation of the p-th root $\boldsymbol{A}^{1/p}$ of a matrix $\boldsymbol{A}$, a commonly used algorithm is the one described by Higham and Lin [33, 34]. For the calculation of inverse p-th roots, i.e., $\boldsymbol{A}^{-1/p}$, there are also iterative methods available, such as the one described by Bini et al. [35] which is briefly presented in the following.

Let $\boldsymbol{X}_k,\ k = 0, 1, \dots$ be the sequence of intermediate result matrices. Starting with an initial guess $\boldsymbol{X}_0$, in each iteration the result is refined by

$$\boldsymbol{X}_{k+1} = \frac{1}{p}\left((p+1)\boldsymbol{X}_k - \boldsymbol{X}_k^{p+1}\boldsymbol{A}\right). \tag{2.1}$$

If the initial guess $\boldsymbol{X}_0$ was already close to $\boldsymbol{A}^{-1/p}$ such that

$$\left\|\boldsymbol{I} - \boldsymbol{X}_0^p\boldsymbol{A}\right\|_2 < 1, \tag{2.2}$$

$\boldsymbol{X}_k$ for $k \to \infty$ converges against $\boldsymbol{A}^{-1/p}$. For $p = 1$ the algorithm corresponds to the well-known Newton-Schulz method [36] used to iteratively calculate inverse matrices. There are different possible choices for $\boldsymbol{X}_0$. Throughout this work, we use

$$\boldsymbol{X}_0 = \left(\|\boldsymbol{A}\|_1 \cdot \|\boldsymbol{A}\|_\infty\right)^{-1}\boldsymbol{A}^\mathsf{T}, \tag{2.3}$$

which is proven to always fulfill constraint (2.2) and therefore guarantees convergence [A4, Proposition 4.1]. The shown method can be extended to perform more matrix multiplications within a single iterations and therefore require fewer iterations overall which we have shown in related work [A4]. For simplicity, we will however stick to the original method throughout this work.

An alternative to using iterative methods is to compute the eigendecomposition of A and compute the inverse p-th roots of all eigenvalues:

$$\begin{aligned}\boldsymbol{A} &= \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}\\ \Lambda'_{i,i} &= \Lambda_{i,i}^{-1/p}\\ \boldsymbol{A}^{-1/p} &= \boldsymbol{Q}\boldsymbol{\Lambda}'\boldsymbol{Q}^{-1}.\end{aligned} \tag{2.4}$$

For symmetric matrices, the same approach can be followed using $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\mathsf{T}$ as a decomposition.

2.3.3 Matrix Sign Function

For scalar values, the sign function denotes whether the real part of that value is positive or negative. Formally, it can be defined as follows.

Definition 2.16. For a scalar value $x \in \mathbb{C}$ with x not on the imaginary axis, i.e., $\mathrm{Re}(x) \neq 0$, the sign function is defined as

$$\mathrm{sign}(x) = \begin{cases} +1, & \text{if } \mathrm{Re}(x) > 0\\ -1, & \text{if } \mathrm{Re}(x) < 0\end{cases}$$

The sign function can also be computed as

$$\mathrm{sign}(x) = \frac{x}{\sqrt{x^2}} = x\left(x^2\right)^{-1/2}.$$

Using this scheme, we define the sign function for square matrices as follows.

Definition 2.17. For a matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ with no eigenvalues on the imaginary axis, the matrix sign function is defined as

$$\operatorname{sign}(\boldsymbol{A}) = \boldsymbol{A}\left(\boldsymbol{A}^2\right)^{-1/2}.$$

The sign function maps all eigenvalues $\lambda_i(\boldsymbol{A}), i = 1 \dots n$ to

$$\lambda_i(\operatorname{sign}(\boldsymbol{A})) = \begin{cases} +1, & \text{if } \operatorname{Re}(\lambda_i(\boldsymbol{A})) > 0 \\ -1, & \text{if } \operatorname{Re}(\lambda_i(\boldsymbol{A})) < 0 \end{cases} \tag{2.5}$$

while leaving the eigenvectors of the matrix unchanged. From Definition 2.17 it is obvious that the matrix sign function can be computed using matrix powers and inverse matrix square roots. Additionally, there are different iterative schemes available to compute the matrix sign function. One of them is the Newton-Schulz iteration [36, 32, Chapter 5.3] given by

$$\begin{aligned} \boldsymbol{X}_0 = \boldsymbol{A}, \quad \boldsymbol{X}_{k+1} &= \frac{1}{2}\boldsymbol{X}_k(3\boldsymbol{I} - \boldsymbol{X}_k^2) \\ \operatorname{sign}(\boldsymbol{A}) &= \lim_{k\to\infty} \boldsymbol{X}_k, \end{aligned} \tag{2.6}$$

Convergence is guaranteed if $\left\|\boldsymbol{I} - \boldsymbol{A}^2\right\| < 1$ for any of the norms from Definition 2.12. Beyond the Newton-Schulz method, there is the Padé family of iterations [37, 32, Chapter 5.4] whose member functions converge with different order. For example, a possible third order scheme is given by

$$\begin{aligned} \boldsymbol{X}_0 = \boldsymbol{A}, \quad \boldsymbol{X}_{k+1} &= \frac{1}{8}\boldsymbol{X}_k(15\boldsymbol{I} - 10\boldsymbol{X}_k^2 + 3\boldsymbol{X}_k^4) \\ \operatorname{sign}(\boldsymbol{A}) &= \lim_{k\to\infty} \boldsymbol{X}_k. \end{aligned} \tag{2.7}$$

Next to the use of iterative schemes, the sign function can be computed based on the eigendecomposition of the input matrix by applying the scalar sign function from Definition 2.16 to all eigenvalues:

$$\begin{aligned} \boldsymbol{A} &= \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1} \\ \Lambda'_{i,i} &= \operatorname{sign}(\Lambda_{i,i}) \\ \operatorname{sign}(\boldsymbol{A}) &= \boldsymbol{Q}\boldsymbol{\Lambda}'\boldsymbol{Q}^{-1}. \end{aligned} \tag{2.8}$$

For symmetric matrices, the same approach can be followed using $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\mathsf{T}$ as a decomposition. Since $\boldsymbol{\Lambda}'$ is a matrix with only ± 1 on its diagonal and all other of its elements being zero, it is involutory. From this we can easily deduct that for all matrices $\boldsymbol{A}$, $\operatorname{sign}(\boldsymbol{A})$ is involutory:

$$(\operatorname{sign}(\boldsymbol{A}))^2 = \boldsymbol{Q}\boldsymbol{\Lambda}'\boldsymbol{Q}^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}'\boldsymbol{Q}^{-1} = \boldsymbol{Q}\boldsymbol{\Lambda}'\boldsymbol{\Lambda}'\boldsymbol{Q}^{-1} = \boldsymbol{Q}\boldsymbol{Q}^{-1} = \boldsymbol{I} \tag{2.9}$$

Chapter 3

Ab-Initio Molecular Dynamics and Electronic Structure Calculations

Electronic structure calculations such as Density Functional Theory (DFT) calculations nowadays make up a large portion of the worldwide HPC workload. Statistics published by some of the TOP500 HPC centers specify 25% to 50% of the compute time being dedicated to chemistry and material science applications, leading to estimates of the share of compute time consumed by these applications to about 35% [38]. With that, this field also significantly contributes to the high energy demands of today's HPC systems and optimizations of widely used methods potentially have a large global impact. In this chapter, one particular application of electronic structure methods, namely Ab-Initio Molecular Dynamics (AIMD), and CP2K as the DFT and Molecular Dynamics (MD) code used in this work are briefly presented.

3.1 Molecular Dynamics Simulations

Computational chemistry is, next to theoretical chemistry and practical lab experiments, one of the major sources of new insights into chemistry. MD simulations [39, 40] can be regarded as experiments in a virtual lab, allowing to closely observe the interaction of up to trillions of atoms [41, 42] in precisely determined environments. With that, they are especially well suited to determine chemical properties of certain substances under conditions that cannot be easily produced or observed in laboratories, or to get insights on a molecular level that can barely made visible under modern scanning tunneling microscopes [43]. Apart from basic research, helping to understand physical, chemical and biological processes, these simulations drive the development of new materials, technologies and drugs.

MD simulations are a kind of n-body simulation in which the dynamic of a system comprised of a number of particles (here: atomic nuclei) is simulated in time. For that, each of the particles is assigned an initial position $\vec{R}_i(t_0)$ and velocity $\vec{v}_i(t_0)$. In each simulation step that covers a certain discrete time interval Δt, the forces $\vec{F}_i(t)$ that act on the particles at time t are computed and positions and velocities are updated accordingly.

3.1.1 Computation and Integration of Forces

In an MD simulation, position and velocity of each particle are updated in every simulation step based on the forces that act on the particle. As these forces continuously act on the particles, it is necessary to integrate Newton's equations of motion. There are many different integrators available. One relatively simple and popular one is the *Velocity Verlet* method [44]. It approximates positions and velocities of the particles as follows:

$$\vec{R}_i(t+\Delta t) = \vec{R}_i(t) + \vec{v}_i(t)\cdot\Delta t + \frac{\vec{F}_i(t)}{2}\cdot\Delta t^2 \tag{3.1}$$

$$\vec{v}_i(t+\Delta t) = \vec{v}_i(t) + \frac{\vec{F}_i(t+\Delta t)+\vec{F}_i(t)}{2}\cdot\Delta t \tag{3.2}$$

The forces between the particles can be determined using different approaches. One is to use classical empirically determined force fields. Nowadays, also artificial neural networks can be trained and used to process atomic positions and output forces [45, 46, 47]. However, both of these methods rely on reference data and the accuracy of the resulting models. A much more general but also more compute intensive method is to compute the forces from first principles. This last form is called Ab-Initio Molecular Dynamics (AIMD) [48] and is the method targeted by this work.

The force that acts on a particle j can be determined as negative gradient of the total energy of the system:

$$\vec{F}_j(t) = -\frac{\mathrm{d}E(\vec{R}_0(t),\dots,\vec{R}_j(t),\dots,\vec{R}_i(t))}{\mathrm{d}\vec{R}_j(t)} \tag{3.3}$$

This energy can be split into two parts: The electrostatic energy between the nuclei E_{ion} and the energy of the electrons in their ground state E_{elec}, i.e.:

$$E = E_{\mathrm{ion}} + E_{\mathrm{elec}}. \tag{3.4}$$

E_{ion} is purely based on the position of the nuclei and can be computed based on classical physics:

$$E_{\mathrm{ion}} = \sum_{i\neq j}\frac{Z_iZ_je^2}{4\pi\varepsilon_0\left|\vec{R}_i-\vec{R}_j\right|}, \tag{3.5}$$

where Z_i and Z_j are the positive charges of the corresponding nuclei in units of the elementary charge e[1] and ε_0 is the vacuum permittivity[2]. To compute the remaining part E_{elec} of the total energy, and therefore to be able to compute the forces in an AIMD simulation, quantum chemical electronic structure methods need to be used. DFT as one very popular electronic structure method will be described in Section 3.2.

3.1.2 Statistical Ensembles

MD simulations can be performed for different statistical ensembles. These ensembles describe which observables of the system are fixed and which are variable. Three such ensembles are relevant in the context of this work, namely the microcanonical ensemble, the canonical ensemble and the grand canonical ensemble.

In the microcanonical ensemble (also called *NVE* ensemble), the number of particles N, the volume V and the total energy of the system E is fixed, with the temperature T being variable. This can be imagined as a perfectly isolated container that does not allow particles or energy to enter or leave the system, as illustrated in

[1] $e = 1.602\ldots\times10^{-19}C$
[2] $\varepsilon_0 = 8.854\ldots\times10^{-12}\frac{C}{Vm}$

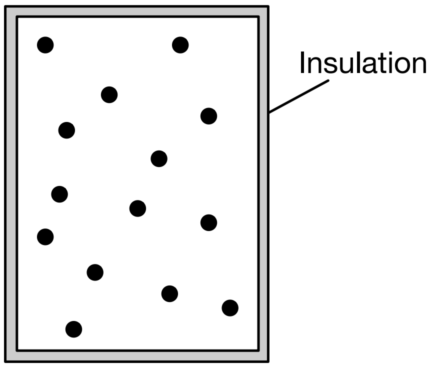

Figure 3.1: Microcanonical (NVE) ensemble: Fixed number of particles, volume and energy.

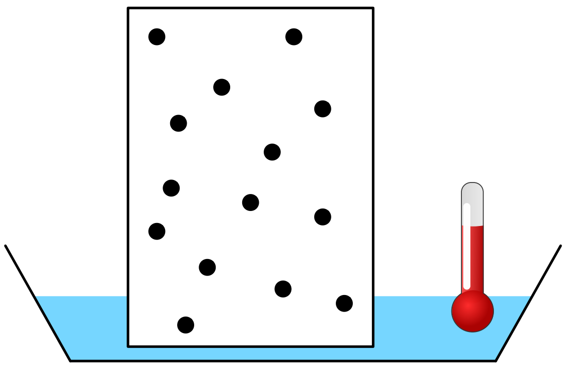

Figure 3.2: Canonical (NVT) ensemble: Fixed number of particles, volume and temperature.

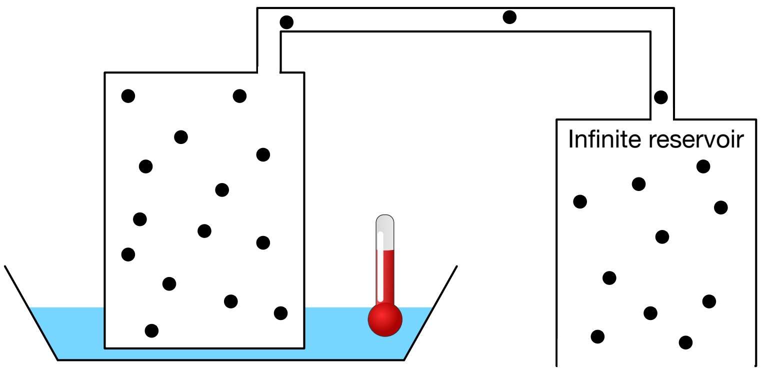

Figure 3.3: Grand canonical (μVT) ensemble: Fixed chemical potential, volume and temperature.

Figure 3.1. This ensemble is naturally simulated when integrating the equations of motion as discussed in the previous section.

In the canonical ensemble (also called *NVT* ensemble), the number of particles, the volume and the temperature are fixed, while the energy of the system is variable. This can be imagined as a closed container in which a known number of particles is enclosed and which is held at a constant temperature by either adding or removing thermal energy from the system, as illustrated in Figure 3.2. In classical physics, the temperature is determined by the average kinetic energy of the particles. Hence, when simulating a canonical ensemble, where a fixed temperature needs to be held during the entire simulation, the kinetic energies need to be controlled accordingly. To achieve this, a so-called *thermostat* needs to be implemented that for example rescales the velocities after applying the forces to match a given temperature.

In the grand canonical ensemble (also called μVT ensemble), the number of particles as well as the energy of the system are variable. Volume, temperature and additionally, the so-called chemical potential μ, are fixed. The chemical potential μ is a potential that is conjugate to the number of particles. Therefore a larger chemical potential will lead to a decrease in the number of particles. The grand canonical ensemble can be imagined as a container connected to an infinite reservoir of particles, so that the number of particles can vary to match the given temperature and chemical potential, as illustrated in Figure 3.3.

3.2 Density Functional Theory

So far we have discussed the general mechanisms of MD simulations and the fundamental idea to compute forces from first principles as gradients of the total energy of the simulated system. However, to compute the energy of the electrons in their ground state E_{elec}, quantum chemical electronic structure methods are required. One very wide-spread electronic structure method is Density Functional Theory (DFT) [49, 50, 51, 52, 53]. It has gained such a high relevance in the field that in 1998 Walter Kohn was awarded with the Nobel Prize in chemistry for its invention [54, 55]. In DFT, the electron density is computed and used to determine fundamental properties of the system. In the context of AIMD, the electron density determines E_{elec} and therefore the so far missing part of the total energy from Equation 3.4 which ultimately determines the forces. In the following, we briefly describe how E_{elec} can be determined using DFT.

The state of an isolated electron i is described by a so-called wave function

$$\psi_i(\vec{r}) : \mathbb{R}^3 \to \mathbb{C}. \tag{3.6}$$

This wave function assigns a complex value to each point in the three dimensional space. Given the wave function ψ_i of an electron, the density distribution of that electron $n_i(\vec{r})$ is defined as

$$n_i(\vec{r}) = |\psi_i(\vec{r})|^2 \tag{3.7}$$

When describing a system with multiple electrons, the domain of the wave function grows exponentially, rendering computations with many electrons infeasible. In DFT, therefore the so-called *Kohn-Sham system* is considered. This is a fictitious system in which different electrons do not interact with each other. Therefore the electron states can be described by separate single-electron wave functions as introduced in Equation 3.6. An additional fictitious potential is then added to the system that acts on the electrons in a way such that the resulting electron density matches that of the original many-electron system. Due to this approach, the method described here is also referred to as Kohn-Sham DFT (KS-DFT).

Another simplification is to assume that all electrons, due to the very small mass of electrons compared to that of the nuclei, immediately follow any movements of the nuclei and therefore not leave their ground state. This simplifying assumption is called *Born-Oppenheimer approximation* [56].

To determine the electron density for the ground state of the system, the wave functions corresponding to the ground state need to be identified, i.e., the wave functions that correspond to the lowest energy. All possible electron states and their energies are described by the Schrödinger equation. The Schrödinger equation for the Kohn-Sham system is given as

$$\left[-\frac{\hbar^2\nabla^2}{2m_e} + V_{e-ion}(\vec{r}) + V_e(\vec{r}, n(\vec{r}))\right]\psi_i(\vec{r}) = \varepsilon_i\psi_i(\vec{r}), \tag{3.8}$$

where $\psi_i(\vec{r})$ are the single-electron wave functions and ε_i are the corresponding energies. The kinetic energy of the electron is described by

$$-\frac{\hbar^2\nabla^2}{2m_e}, \tag{3.9}$$

where $\hbar$ is the reduced Planck constant[3], ∇^2 is the Laplace operator and m_e is the electron mass[4]. The electrostatic potential of the nuclei acting on the electron is described by

$$V_{e-ion}(\vec{r}) = -\sum_i \frac{e^2 Z_i}{4\pi\varepsilon_0 \left|\vec{r} - \vec{R}_i\right|}. \tag{3.10}$$

Lastly,

$$V_e(\vec{r}, n(\vec{r})) \tag{3.11}$$

is the fictitious potential added to model electron-electron interaction of the original many-electron system.

To make the wave functions accessible by numerical computations, they are represented as a combination of a finite number of so-called basis functions f_j, i.e.,

$$\psi_i(\vec{r}) = \sum_j c_{ij} \cdot f_j(\vec{r}). \tag{3.12}$$

These basis functions can be plane waves, in which case the representation resembles a Fourier series, localized functions that are centered around the nuclei or combinations of these. Examples for localized functions are Slater-type orbitals [57], Gaussian-type orbitals [58] or wavelets [59]. From here on, we will only consider localized, atom-centered basis functions. The concrete basis functions used in this work will be described in more detail in Section 3.3.2.

In terms of these basis functions f_j and coefficients c_{ij}, the so-called *Hamilton matrix* $\boldsymbol{H}$ can be defined with elements

$$H_{k,j} = \int d^3\vec{r} f_k^*(\vec{r}) \left[-\frac{\hbar^2\nabla^2}{2m_e} + V_{e-ion}(\vec{r}) + V_e(\vec{r}, n(\vec{r}))\right] f_j(\vec{r}). \tag{3.13}$$

[3] $\hbar = 1.054\ldots \times 10^{-34} Js$

[4] $m_e = 9.109\ldots \times 10^{-31} kg$

It is also referred to as *Hamiltonian* or *Kohn-Sham matrix*. The so-called *overlap matrix* $\boldsymbol{S}$ can be defined with elements

$$S_{k,j} = \int d^3\vec{r} f_k^*(\vec{r}) f_j(\vec{r}) \tag{3.14}$$

The Schrödinger equation can then simply be written as

$$\boldsymbol{H}\vec{c}_i = \varepsilon_i \boldsymbol{S}\vec{c}_i. \tag{3.15}$$

This form resembles a generalized eigenvalue problem. Its solution provides possible electron states in terms of the coefficients $\vec{c}_i$ and the corresponding energies ε_i. Orthogonalizing the Hamilton matrix leads to a conventional eigenvalue problem:

$$\tilde{\boldsymbol{H}}\vec{c}_i = \varepsilon_i \vec{c}_i, \tag{3.16}$$

where $\tilde{\boldsymbol{H}} = \boldsymbol{S}^{-1}\boldsymbol{H}$ (asymmetric orthogonalization) or $\tilde{\boldsymbol{H}} = \boldsymbol{S}^{-1/2}\boldsymbol{H}\boldsymbol{S}^{-1/2}$ (symmetric Löwdin orthogonalization [60]).

3.2.1 SCF Cycle

As shown in Equation 3.11, the electron density is required to compute the fictitious potential V_e and therefore to compute the electron density itself in KS-DFT. To handle this cyclic dependency, the density is calculated iteratively, following the so-called Self Consistent Field (SCF) iteration:

1. Start with an initial guess of the electron density $n(\vec{r})$.
2. Compute V_e based on currently assumed electron density $n(\vec{r})$.
3. Construct Hamilton matrix $\boldsymbol{H}$.
4. Solve eigenvalue problem from Equation 3.16 to determine ground state.
5. Compute electron density $n(\vec{r})$.
6. If electron density changed: Go back to step 2, otherwise finish.

In practice, this iteration scheme is often slightly altered to improve convergence behavior. In particular, the newly computed electron density is typically mixed with the previous one to reduce the step size of this iterative algorithm.

3.2.2 Density Matrix Based DFT

Although the electron density is sufficient to describe observables of the Kohn-Sham system, a more extensive but useful representation is the single-particle density matrix. At thermodynamic equilibrium and zero temperature it is defined as

$$D(\vec{r}, \vec{r'}) = \sum_i \Theta(\mu - \varepsilon_i)\psi_i(\vec{r})\psi_i^*(\vec{r'}), \tag{3.17}$$

where Θ is the Heaviside step function. The term $\Theta(\mu - \varepsilon_i)$ assigns the value 1 to all energy levels ε_i that are below the chemical potential μ, and 0 to all energy levels above μ. This term therefore mathematically describes the fact that in the ground state, the lowest energy levels up to the chemical potential μ are occupied with electrons.

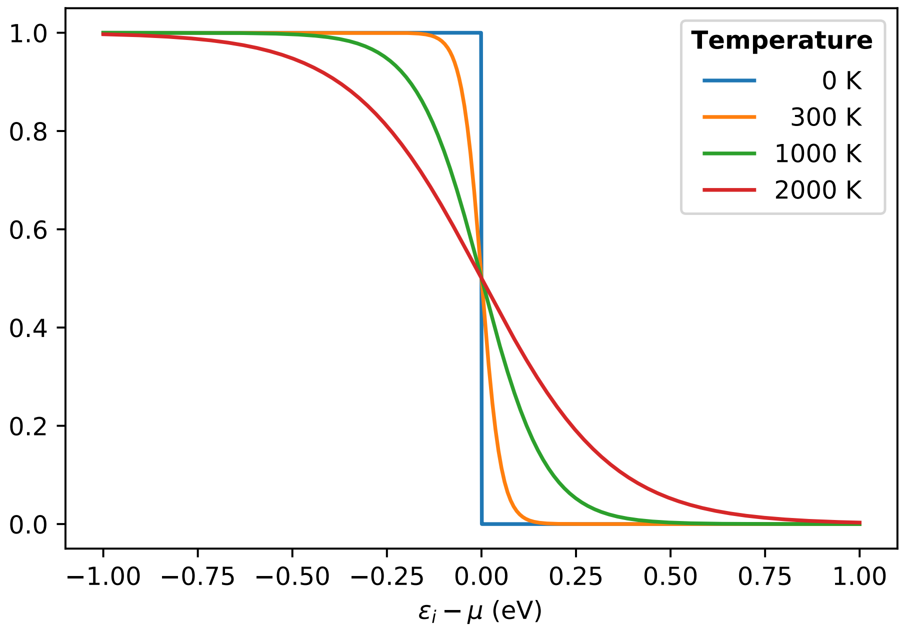

Figure 3.4: Fermi function for different temperatures. For $T \to 0$, it becomes closer and closer to the step function $\Theta(\mu - \varepsilon_i)$.

While for the moment we stick to zero temperature here, it is important to note that the definition of the density matrix changes for finite temperature, in that the Heaviside step function Θ is replaced by the Fermi function

$$W(\varepsilon_i) = \left(\exp\left(\frac{\varepsilon_i - \mu}{k_B T}\right) + 1\right)^{-1}, \tag{3.18}$$

where k_B denotes the Boltzmann constant[5] and T the temperature. Figure 3.4 depicts the Fermi function for different temperatures. For $T \to 0$, the Fermi function becomes closer and closer to the step function $\Theta(\mu - \varepsilon_i)$.

Going from the wave functions to a representation using basis functions and their coefficients, the density matrix $\boldsymbol{D}$ can be defined as [61]

$$\boldsymbol{D} = \frac{1}{2}\left(\boldsymbol{I} - \operatorname{sign}\left(\boldsymbol{S}^{-1}\boldsymbol{H} - \mu\boldsymbol{I}\right)\right)\boldsymbol{S}^{-1} \tag{3.19}$$

or

$$\boldsymbol{D} = \frac{1}{2}\boldsymbol{S}^{-1/2}\left(\boldsymbol{I} - \operatorname{sign}\left(\boldsymbol{S}^{-1/2}\boldsymbol{H}\boldsymbol{S}^{-1/2} - \mu\boldsymbol{I}\right)\right)\boldsymbol{S}^{-1/2}, \tag{3.20}$$

depending on the chosen orthogonalization scheme. In the form shown here, we assume that there is no spin polarization and therefore only electrons with one of the two possible spin quantum numbers need to be considered (see Section 3.3.2).

The electron density can directly be determined from the diagonal elements of the density matrix, i.e.,

$$n(\vec{r}) = 2 \cdot D(\vec{r}, \vec{r}). \tag{3.21}$$

[5] $k_B = 1.380\ldots \times 10^{-23}\frac{J}{K} = 8.617\ldots \times 10^{-5}\frac{eV}{K}$

Off-diagonal elements represent the covalency of the system. Also the energy of the electrons E_{elec} can be determined directly from the density matrix as

$$E_{\text{elec}} = 2 \cdot \sum_i \varepsilon_i = 2 \cdot \text{Tr}(DH). \tag{3.22}$$

Computing the density matrix directly from the Hamilton matrix (in the following also referred to as *purification*) using Equation 3.19 or 3.20, can therefore substitute explicitly solving the eigenvalue problem shown in Equation 3.16.

3.2.3 Linear Scaling DFT

The computational effort required to solve the eigenvalue problem shown in Equation 3.16 or to compute the sign function in Equation 3.19 or 3.20 scales cubically with the size of the Hamilton and density matrices. This size directly corresponds to the number of basis functions used to represent the electron configuration. For localized basis sets, the size therefore depends on the chosen basis set and the number and kinds of atoms in the system.

This computational complexity severely limits the maximum size of systems that can practically be examined using DFT. However, to reduce this complexity, a physical property referred to as the *nearsightedness of electronic matter* [62] can be exploited. It states that in a large system, the interaction of electrons that are far apart becomes negligible. Consequently, the corresponding values in the density matrix, which describes the state of that system, become negligible. Truncating very small values in the considered matrices $\boldsymbol{H}, \boldsymbol{S}$ and $\boldsymbol{D}$ makes them sparse for sufficiently large systems. Increasing the system size further then still grows the matrices accordingly but at the same time the matrices become more sparse and the number of nonzero elements in the matrices only grows linearly with the number of atoms. Exploiting this behavior using sparse linear algebra, DFT methods can be developed that scale linearly instead of cubically with the number of atoms. We refer to these methods as Linear Scaling DFT (LSDFT) [63, 64, 65].

3.3 LSDFT in the Quantum Chemistry Code CP2K

CP2K [66, A1] is an open source software package for atomistic simulations, providing support for different modeling and simulation methodologies, including Molecular Dynamics simulations. Forces between atoms can either be computed using classical force fields or using electronic structure methods such as DFT. In this work, we focus on the DFT implementation *Quickstep* [67, A1] within CP2K. In the following, we briefly describe the parts of CP2K that are relevant for this work.

3.3.1 Distributed Block Compressed Sparse Row (DBCSR) Matrix Library

Key to the implementation of LSDFT methods is to exploit the sparsity of the involved matrices. To implement efficient sparse matrix arithmetic in a distributed memory system, CP2K uses the *libDBCSR* [68] library. libDBCSR follows the idea that the sparsity patterns of the processed matrices are not random but typically show certain patterns. A matrix stored in DBCSR format is divided into a 2D grid of relatively small matrix blocks which typically contain 5–30 rows and columns. The information which blocks are zero and which contain nonzero elements is stored in Compressed Sparse Row (CSR) format. The blocks with nonzero elements are stored in a dense format. For efficient processing with MPI, libDBCSR arranges the MPI ranks in a 2D cartesian topology and creates a mapping from matrix block rows and block columns to MPI ranks that store these blocks.

libDBCSR provides routines for many matrix operations, in particular matrix-matrix multiplication which is implemented based on a modified Cannon's algorithm [69]. As part of this algorithm, many multiplications of the small DBCSR matrix blocks need to be performed. While this is generally possible using standard Basic Linear Algebra Subprograms (BLAS) implementations, these are typically not optimized for operation on such small matrices. libDBCSR therefore contains a custom library *libsmm* for small matrix-matrix multiplications. Alternatively, *libxsmm* [70] can be used for Intel-based systems and there is also a GPU-accelerated version named *libsmm_acc* [71] (formerly *libcusmm*) included in libDBCSR.

3.3.2 Basis Sets Relevant for This Work

Using localized basis functions with a limited range is key for obtaining sparse Hamilton and overlap matrices for large systems. While the number of the basis functions used for certain elements (the *basis set*) and their range is limited, they still need to cover all relevant quantum states of the electrons of that element. In this section, we briefly cover the basis sets used in this work. Before that, we describe the quantum states that need to be covered by the basis sets.

In general, the quantum state of an isolated electron in a non-relativistic case is determined by the following quantum numbers:

Principal quantum number $n \in \mathbb{N}$. This number determines the *shell* on which the electron is located and therefore the energy level of the electron. The shells are either denoted by natural numbers or by letters ($K = 1, L = 2, M = 3, N = 4, \ldots$).

Azimuthal quantum number $l \in \{0, \ldots, n-1\}$. This number determines the *subshell* or *orbital* on which the electron is located and it is also referred to as the angular momentum. The subshells are often referred to by letters ($s = 0, p = 1, d = 2, f = 3, g = 4, \ldots$). To name a specific subshell, we use the format Nl, e.g., $2s$ for the s-subshell on the L-shell.

Magnetic quantum number $m_l \in \{-l, \ldots, l\}$. It specifies the orientation of the subshell as shown in Figure 3.5. The number of possible orientations is $2l + 1$.

Spin quantum number $m_s \in \{-s, \ldots, s\}$. For electrons, $s = \frac{1}{2}$, such that for electrons $m_s \in \{-\frac{1}{2}, \frac{1}{2}\}$.

According to the Pauli exclusion principle [73], each quantum state can only be occupied by a single electron. Therefore an s-subshell can only hold two electrons, a p-subshell can hold six electrons and a d-subshell can only hold ten electrons. The subshells are filled in a specific order, following the so-called aufbau principle [74, Ch. 2.5.7].

When working with CP2K, throughout this thesis we will use the basis sets proposed by VandeVondele and Hutter [75] which are implemented in CP2K. These basis sets are all Gaussian type orbitals, and there are different variants provided. All of them are optimized to describe molecules and only cover the valence electrons, i.e., the electrons on the outermost occupied shell. The electrons on inner shells are considered bound to the core and their influence on the valence electrons is modelled

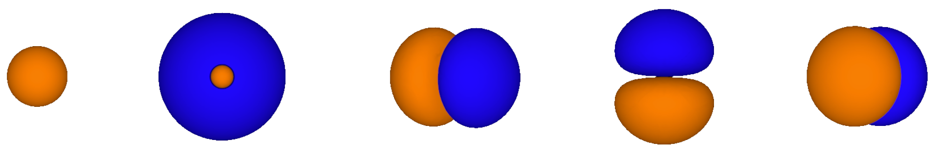

Figure 3.5: Shapes of the orbitals 1s, 2s and the three 2p orbitals 2p_x, 2p_y and 2p_z. The illustration is based on [72].

using a pseudopotential. Furthermore, no spin polarization is considered and therefore only one spin direction is computed, i.e., electrons with spin quantum number $-\frac{1}{2}$ and $\frac{1}{2}$ are not considered separately. This effectively halves the number of basis functions per quantum state. Regarding the number of basis functions, it can be chosen between the following variants:

Single-Zeta Valence (SZV) This is the smallest available basis set. Here, one, three, five, ... functions are used to cover $s, p, d, \ldots$ subshells, respectively. A hydrogen atom, which only has an electron on the K shell, is therefore represented by a single function (s). An oxygen atom, which has six valence electrons on the L shell, is represented by four functions (one for s, three for p).

Double-Zeta Valence Polarized (DZVP) In a double-zeta basis set, the number of functions used to describe the valence electrons is doubled. Additionally, there are so-called polarization functions added for higher subshells that allow to better describe chemical bonds. A hydrogen atom here is represented by five functions (two for s, three for p) and oxygen by 13 functions (two for s, six for p, five for d).

Triple-Zeta Valence Polarized (TZVP) Here, the number of functions used to describe the valence electrons is three times higher compared to SZV. Similar to DZVP, polarization functions are added.

There are further variants of basis sets available, e.g., adding more polarization functions, which are not relevant for this work. Two basis sets also come in a Short Range (SR) variant, which contain especially short ranged basis functions that are optimized for LSDFT[6]. Throughout this work, we will use these two LSDFT-optimized basis sets, namely *SZV-MOLOPT-SR-GTH* and *DZVP-MOLOPT-SR-GTH*.

3.3.3 Computational Hotspots in AIMD Simulations

So far we have discussed the fundamentals of AIMD and DFT, including the basics of the LSDFT implementation in CP2K. Before focusing more on this method throughout this thesis, we want to assess the impact that these DFT computations have on the overall run time of an AIMD simulation.

CP2K comes with an integrated timing measurement framework that automatically on each run measures the number of calls to certain functions as well as the time spent within those functions. This allows to easily spot computational hotspots,

[6] `https://github.com/cp2k/cp2k/blob/1e5aa65/data/BASIS_MOLOPT#L26`

Table 3.1: Reduced timing output of CP2K using an SZV basis set. Shown are the main hotspots of the entire AIMD simulation.

Subroutine	# Calls	Time (s)	Relative Time
CP2K	1	1058	100%
ls_scf	11	989	93.5%
density_matrix_sign	29	656	62.0%

guiding the optimization of the implemented methods as well as the implementation itself. To get an overview over the significance of different parts of an AIMD run, we simulate systems of bulk water with CP2K on a single compute node of the Noctua 1 cluster described in Section 2.1.1.

For this evaluation, we perform an MD simulation covering ten simulation steps with a timespan of 0.5 fs between each time step. We consider a microcanonical (NVE) ensemble, i.e., a constant number of particles, a constant volume and a constant total energy. For the DFT calculations, we use the LSDFT method described in Section 3.2.2, i.e., we use the sign iteration to compute the density matrix. For the SCF cycle, we set the convergence threshold $\varepsilon_{\mathrm{SCF}}$ to 10^{-6} and the maximum number of iterations to 30. In our simulation, the SCF iterations always converges with fewer iterations. Lastly, we set the $\varepsilon_{\mathrm{filter}}$ argument to 10^{-6} as well. This argument enables filtering out all matrix elements that are smaller than the given threshold after matrix operations, such that the matrices stay sparse. Furthermore, we use an easy to scale system from the CP2K benchmarks for our simulation: a cube filled with liquid water (32 H_2O molecules) that can be replicated in all three dimensions.

Timing Profile Using an SZV Basis Set

Using an SZV basis set, we replicate the cube of water four times in each dimension, leading to a system comprised of $4^3 \cdot 32 = 2048$ water molecules. The most relevant output of the integrated timing measurement framework is shown in Table 3.1. The entire program ran for over 17 minutes, of which the electronic structure computations using the SCF cycle took 93.5%. Computing the matrix sign function for the density matrix is the main hotspot of the SCF cycle and consumed overall 62.0% of the entire run time, or 66.3% of the SCF cycle.

Timing Profile Using a DZVP Basis Set

Using a DZVP basis set, the matrices become much larger compared to the previous run. To be able to simulate the system on a single node of the Noctua 1 cluster, we therefore need to reduce the system size by replicating the basic block only twice in each dimension. We therefore consider a system comprised of $2^3 \cdot 32 = 256$ water molecules. The timing results are shown in Table 3.2. The overall program ran for over 18 minutes. The iterative SCF cycle takes up 97.8% of the total run time with 75.9% of the total run time and 77.6% of the SCF run time spent computing the sign function of the density matrix.

Both for SZV and DZVP it is evident that solving the Schrödinger equation using the iterative SCF cycle is the main computational hotspot in an AIMD simulation in CP2K with the presented configuration. Within the SCF cycle, computing the matrix sign function dominates the run time and it is therefore the most promising part of the computation to optimize in order to get an overall speedup of the simulation.

Table 3.2: Reduced timing output of CP2K using a DZVP basis set. Shown are the main hotspots of the entire AIMD simulation.

Subroutine	# Calls	Time (s)	Relative Time
CP2K	1	1115	100%
ls_scf	11	1090	97.8%
density_matrix_sign	29	846	75.9%

3.4 Motivating Approximations in DFT Computations

There are multiple aspects that suggest that DFT computations, in particular in the context of AIMD, are suitable for acceleration using targeted approximations. One is that DFT computations and MD simulations already incorporate certain approximations.

Firstly, while DFT itself is an accurate theory, in practice approximations of the real world need to be performed to make computations feasible. For example, with the Born-Oppenheimer approximation we neglect any deviation from the ground state of the electrons. Moreover, any interaction between electrons is represented by a functional of which only approximations are known. Secondly, when approaching DFT numerically, we need to represent the electron wave functions by a finite set of basis functions which is not able to precisely match the original wave functions. When performing MD simulations, we simulate discrete time steps of finite length, whereas infinitely small time steps would be required to perfectly simulate real trajectories.

Next to these fundamental approximations that are already contained in the method, specific implementations further limit precision of the computed results. CP2K includes configurable thresholds to specify the targeted precision of the SCF cycle and to deliberately neglect values from matrices that are below a certain absolute value. This shows that in the practical application of DFT and AIMD, there is always a trade off between the accuracy of the results and the invested computational resources. Furthermore, AIMD simulations with certain approximations in the DFT computations can still provide qualitatively better results than classical MD based on empirically determined force fields.

Lastly, when performing AIMD simulations, we are often not interested in the exact trajectories of single particles with precisely determined initial positions, but instead want to determine ensemble averages of certain observables. Based on the fluctuation dissipation theorem [76], errors in the forces used in an MD simulation can be entirely compensated if the error in the forces resemble white noise. In related work, we have shown that even for errors that are in the same order of magnitude as the forces themselves, we can still obtain precise values for ensemble averages [A3]. Based on this observation, introducing approximations into the force calculation and therefore the underlying DFT code, is a promising approach to accelerate this computational hotspot of AIMD. In the following chapter, we follow this idea by applying low-precision computations onto two computational kernels from density matrix based DFT.

Chapter 4

Iterative Methods as Target for Approximations

Iterative schemes are an interesting target in the Approximate Computing paradigm. One reason is that naturally, iterative algorithms that converge against the sought solution can be terminated early to obtain an approximate solution. This approach has direct influence on the run time and therefore the energy consumption of the program. Another approach is to approximate the computations within each iteration. The idea here is that errors introduced within one iteration may be corrected or compensated for in a succeeding iteration and approximate computations may suffice to converge against a good solution. A third aspect that makes iterative schemes interesting is that they often can be used to refine an approximate result to obtain more precise solutions if needed.

We have seen that the iterative computation of the matrix sign function is the main computational hotspot in CP2K's LSDFT implementation and therefore using approximate arithmetic in this computation is a promising approach to accelerate the entire method. In this chapter, we therefore examine how the relevant iterative methods handle the use of low-precision computations within each iteration. In particular, we look at the computation of matrix inverse p-th roots, which cannot only be used for computation of the matrix sign function following Definition 2.17 on page 17 but which is also needed for the orthogonalization of the Hamiltonian, and the computation of the sign function following the Newton-Schulz iteration shown in Equation 2.6. Parts of this chapter have been presented at the *Approximate Computing Workshop* in 2016 [B6] and in 2018 have been published as an article in the IEEE *Embedded Systems Letters* [A5].

4.1 Computation of Inverse p-th Roots

We first focus on the computation of inverse p-th roots of matrices using the iteration scheme described in Section 2.3.2. To assess the resiliency of the given algorithm to certain errors, we simulate simple approximation techniques. First, we use custom-precision floating-point and fixed-point arithmetic for *all involved calculations* in order to get an understanding of the required data ranges and precision. Second, we restrict the use of these data types to the input data and stored intermediate results, simulating *approximate storage* or *data exchange* of the involved matrices.

4.1.1 Problem and Data Set

We simulate the use of low-precision data types for the orthogonalization of the Hamiltonian by extracting overlap matrices for different systems of liquid water

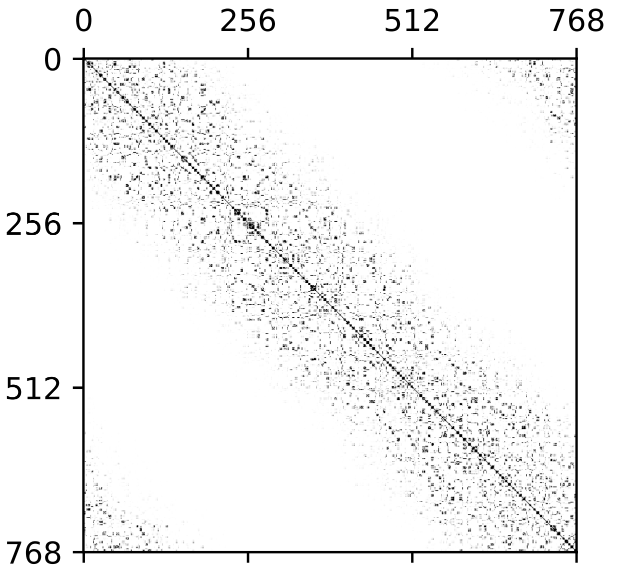

Figure 4.1: Structure of an examined overlap matrix S ($N = 768$).

from a DFT code and then iteratively calculating $S^{-1/2}$. The matrices are obtained from a Daubechies Wavelet-based DFT code [59]. Similar to the basis described in Section 3.3.2, this code uses a minimal set of localized basis functions suitable for Linear Scaling DFT.

Figure 4.1 shows one such overlap matrix S of size $N = 768$ which we used for our evaluation. It is a symmetric positive definite matrix with 25% nonzero elements in the range of $[-1, 1]$, representing a system of 128 H_2O molecules. For larger matrices density decreases to 12.4% ($N = 1536$), 6.2% ($N = 3072$) and 3.1% ($N = 6144$) nonzero elements.

4.1.2 Methodology

For the presented simulations we use Python along with NumPy and SciPy [77], which provide the required data structures and numeric operations. This allows us to define entirely custom data types, e.g. floating-point types with a custom number of bits in the exponent and mantissa, and fixed-point types with a selectable number of bits and selectable scaling factor. Implementing basic arithmetic operations like addition, subtraction and multiplication for our custom data types enables NumPy to use these data types in its own array data structures and numeric operations.

This approach allows a very flexible and fast implementation of simulators for different approximation techniques. Besides the mentioned floating-point and fixed-point data types, influences like noise or random bitflips can be easily implemented and adjusted. This flexibility comes at the cost of a performance penalty as each arithmetic operation now implies doing a function call, performing the necessary simulation steps and instantiating a return object. We deal with this performance degradation by implementing all classes in Cython [78], producing statically typed C-code which executes orders of magnitude faster than interpreted Python code, and distributing the simulations over many machines of a large compute cluster.

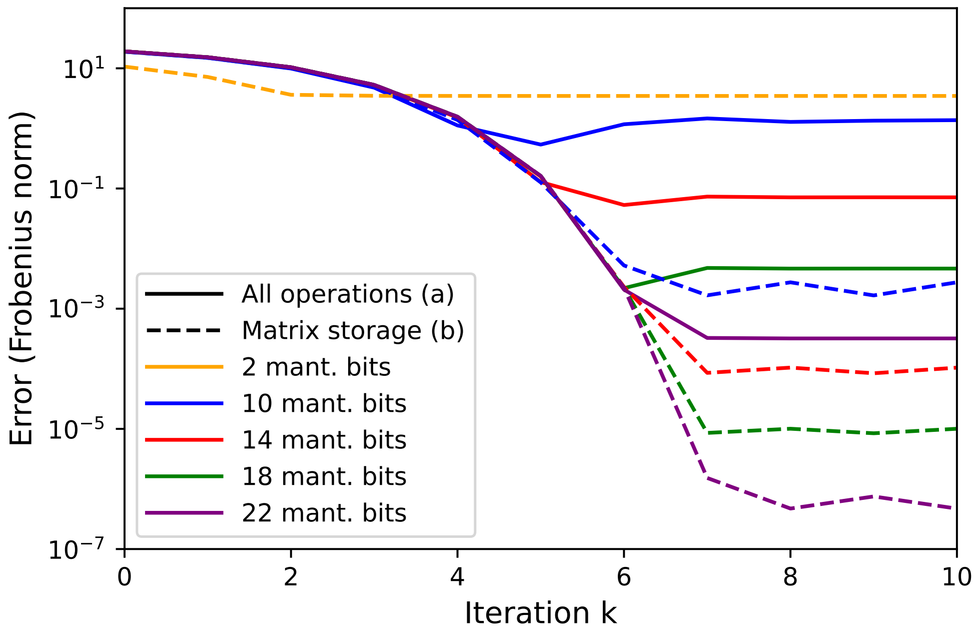

Figure 4.2: Computation of inverse p-th roots: Convergence behavior when using custom-precision floating-point for (a) all arithmetic operations and (b) only for storage of intermediate results ($N = 768, p = 2$).

4.1.3 Results

In the following, we present the results of our simulations and evaluate the influence of different parameters, such as size and condition of the input matrices and the choice of p, onto the resulting error.

Overall Error Resiliency

To assess the overall error resiliency of the algorithm, we initially choose an overlap matrix of dimension $N = 768$ and set $p = 2$ to calculate the inverse square root for these matrices. With simulation, we determine the convergence of the algorithm, depending on the given precision. The iterative algorithm shows to be rather resilient to low precision, both for storage of intermediate matrices as well as for all used arithmetic operations.

Figure 4.2 shows the error between the intermediate solutions $\boldsymbol{X}_k$ obtained from the algorithm using floating-point with custom mantissa widths and a solution that was precomputed using double-precision. As error metric we use the Frobenius norm

$$\left\|\boldsymbol{X}_k - \boldsymbol{S}^{-1/p}\right\|_F := \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N} \left|x_{ij} - s_{ij}\right|^2}, \tag{4.1}$$

where x_{ij} are the elements of $\boldsymbol{X}_k$ and s_{ij} those of $\boldsymbol{S}^{-1/p}$.

The observed convergence of the algorithm can be split into two phases: First, the error steadily decreases according to the algorithm's quadratic order of convergence [35]. In the second phase, being limited by the given precision of the data

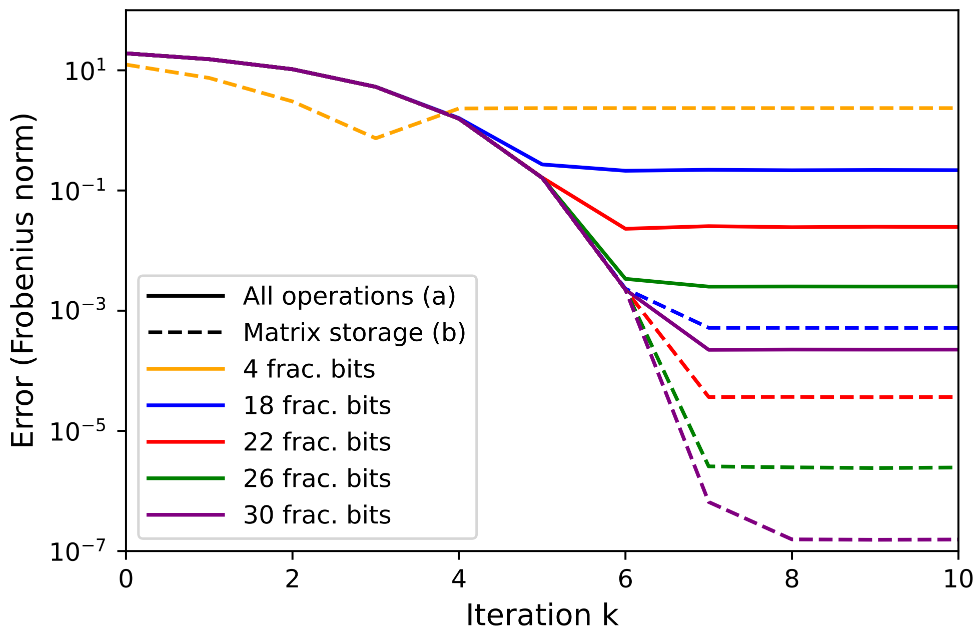

Figure 4.3: Computation of inverse p-th roots: Convergence behavior when using custom-precision fixed-point for (a) all arithmetic operations and (b) only for storage of intermediate results ($N = 768, p = 2$).

type, the algorithm does not converge further but oscillations may be observed. This shows that the convergence in the first phase is barely influenced by the introduced errors. Only for less than 10 mantissa bits the algorithm does not converge at all. Consequently, half-precision floating-point arithmetic is sufficient to retain convergence. Approximation does however increase the lower bound for the error. Therefore, the second phase of conversion starts earlier for lower precision.

Increasing the precision in later iterations allows the algorithm to further converge against a lower error, opening the possibility for dynamic precision scaling. Observing the changes introduced in each iteration, the necessity of increased precision can be detected at runtime. Note that the use of low-precision arithmetic in the first iterations does not increase the overall number of iterations, even when using higher precision in later iterations to achieve more precise results. This allows gains in performance or energy efficiency as soon as a single iteration can be executed more efficiently using approximation techniques. The gain in energy efficiency G can be estimated by

$$G = \frac{E_{\text{prec}} \cdot \#\text{iterations}_{\text{orig}}}{E_{\text{prec}} \cdot \#\text{iterations}_{\text{prec}} + E_{\text{approx}} \cdot \#\text{iterations}_{\text{approx}}} \tag{4.2}$$

where E_{prec} (E_{approx}) denotes the energy consumption of a precisely (approximately) performed arithmetic operation.

Approximating only the storage of any intermediate results allows significantly stronger approximation while achieving similar precision in the output. E.g., storing only 10 mantissa bits allows a similar error as doing all calculation using 18 mantissa bits. In our use case the algorithm still converges if only two mantissa bits are used for all stored values.

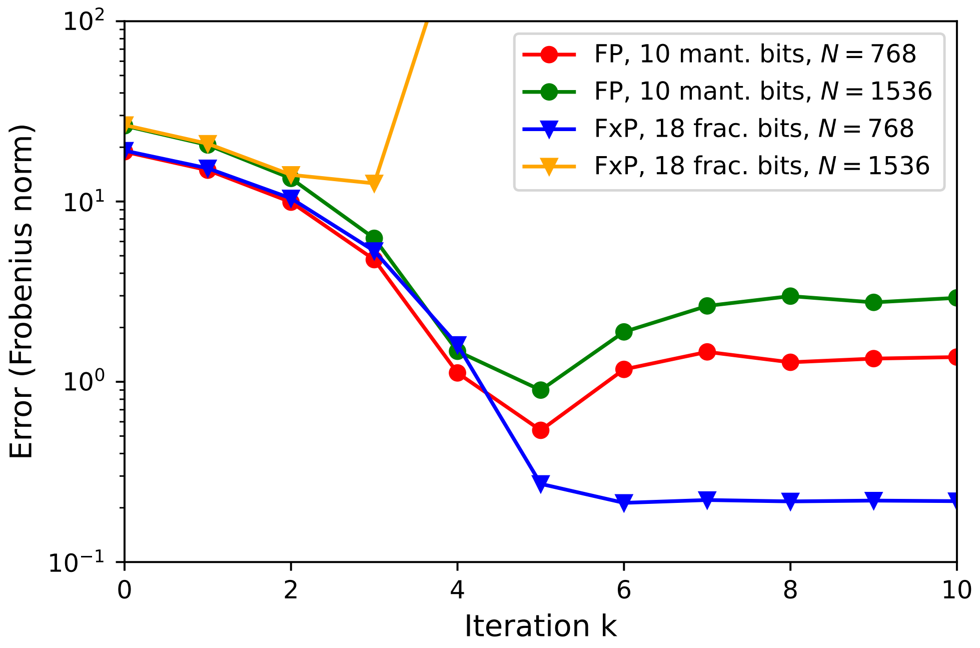

Figure 4.4: Computation of inverse p-th roots: Convergence behavior using custom-precision floating-point (FP) and fixed-point (FxP) for different matrix sizes ($p = 2$).

Figure 4.3 shows similar behavior when using fixed-point arithmetic with low precision. To retain convergence, 18 fractional bits are required for arithmetic operations. Again, restricting the approximation to stored intermediate results permits stronger approximation. In our evaluation, storing only four fractional bits showed to be sufficient to retain convergence.

Influence of the Matrix Size

Most of the results presented before apply directly to larger matrices from our problem set, in particular when only approximating the storage of intermediate results. Approximating all arithmetic operations using low-precision fixed-point arithmetic however exhibits a limitation. As depicted in Figure 4.4, using 18 fractional bits is sufficient to retain convergence for $N = 768$ but for $N = 1536$ the error eventually increases. The reason for this behavior is that larger matrices from our set are more sparse (see Section 4.1.1) and therefore their inverse contain smaller values which cannot be represented appropriately with the given number of fractional bits.

For floating-point this effect is not relevant, as depicted in Figure 4.4. The slightly larger final error for $N = 1536$ can be explained by the use of the Frobenius norm as error metric since it adds up the quadratic errors of all matrix elements and therefore is not invariant to the matrix size.

Influence of p

Calculating the inverse p-th root for $p \neq 2$ shows similar behavior as for $p = 2$, as shown in Figure 4.5 for custom-precision floating-point arithmetic and storage. With increasing p the algorithm in general needs an increasing number of iterations to converge. This effect is independent of the applied approximation.

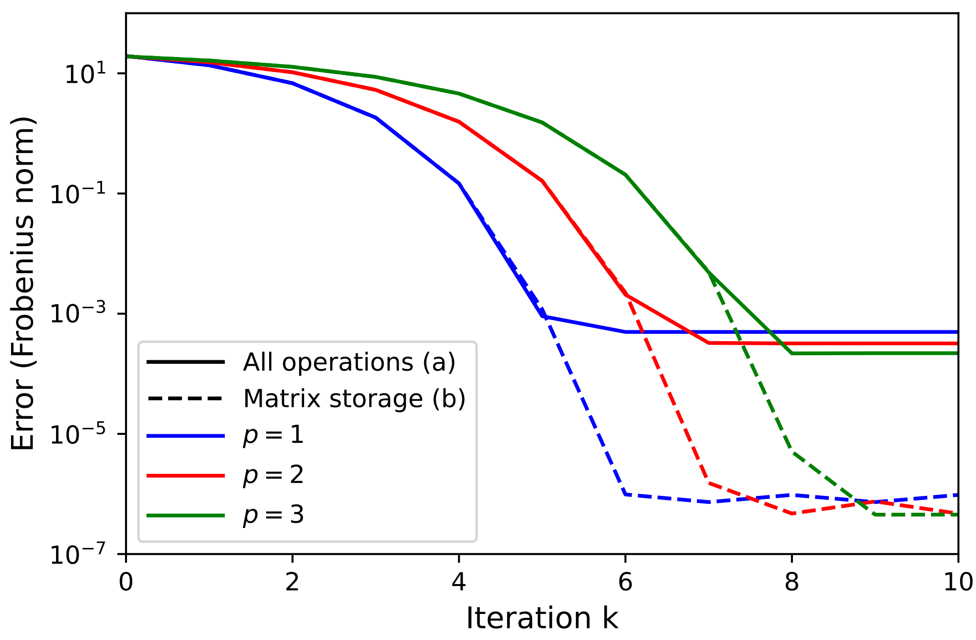

Figure 4.5: Computation of inverse p-th roots: Convergence behavior for different p using custom-precision floating-point with 22 mantissa bits ($N = 768$).

Influence of the Matrix Condition

The condition of the overlap matrices $\boldsymbol{S}$ depends on the system, in particular the element, that is simulated. The matrices for systems of H_2O molecules used in our evaluation have condition numbers around $\kappa = 1.5$. Calculating the inverse p-th root of matrices with larger condition numbers requires overall more iterations, as we discuss in related work [A4]. Additionally, the resulting matrix is expected to be full so that exploiting sparsity of the matrix becomes more difficult.

4.2 Iterative Computation of the Sign Function

We now take a look at the computation of the matrix sign function using custom-precision floating-point and fixed-point calculations. The definition of the sign function given in Definition 2.17 on page 17 shows that it can in principle be computed based on matrix powers and inverse matrix square roots. Based on the results in the previous section, it can therefore be effectively computed using low-precision arithmetic as well.

In this section we will however consider the iterative 2nd order Newton Schulz approach from Equation 2.6. Also we use slightly different input data than in the previous section. While the overlap matrix $\boldsymbol{S}$ was a realistic input set for the calculation of inverse p-th roots, the sign function in the LSDFT scenario is computed for the orthogonalized Hamilton matrix $\tilde{\boldsymbol{H}}$, as shown in Equations 3.19 and 3.20. We therefore extracted this matrix from the same wavelet-based DFT code, simulating the same system of liquid water.

The orthogonalized Hamilton matrix is more dense than the overlap matrix and has 93.6% nonzero values for 128 H_2O molecules ($N = 768$). However, the density decreases with increasing system size as for the overlap matrix. For $N = 1536$ there

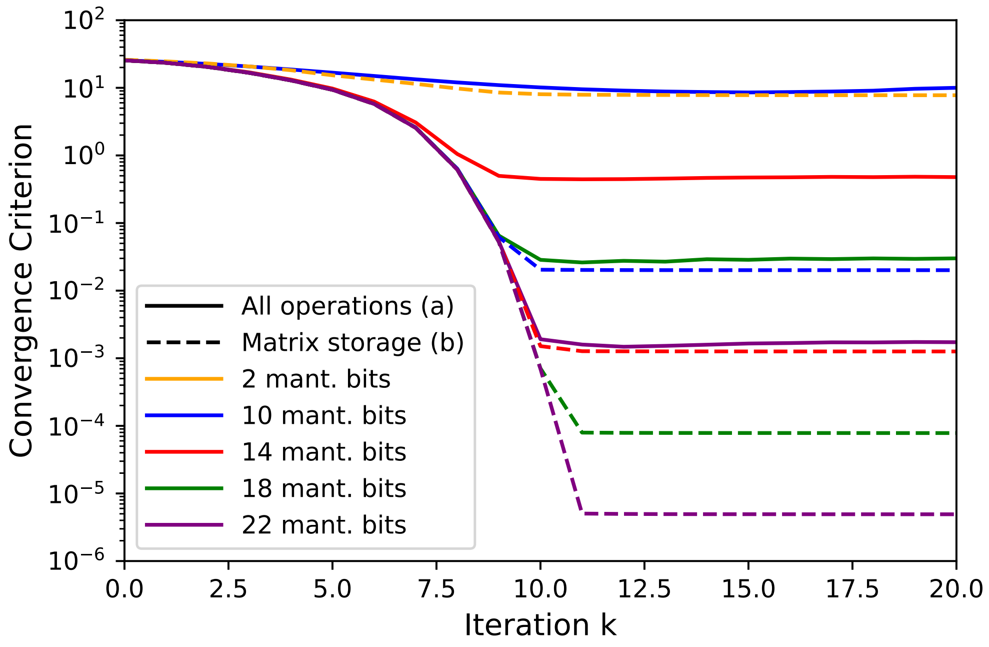

Figure 4.6: Computation of sign function: Convergence behavior when using custom-precision floating-point for (a) all arithmetic operations and (b) only for storage of intermediate results ($N = 768$).

are 63.3%, for $N = 3072$ there are 32.4% and for $N = 6144$ there are 16.1% nonzero values. In the following evaluation, we will use the matrix representing 128 H_2O molecules as input data. Its eigenvalues range from -0.83 to 0.58 and its condition number is $\kappa = 10.1$.

4.2.1 Rate of Convergence

We first look at the convergence criterion of the Newton Schulz iteration. Since the sign functions maps all eigenvalues towards ± 1, the resulting matrix is involutory, i.e., for any matrix $\boldsymbol{A}$ it holds that

$$\boldsymbol{X} = \operatorname{sign}(\boldsymbol{A}) \Rightarrow \boldsymbol{X}^2 = \boldsymbol{I}. \tag{4.3}$$

This property is typically used to determine convergence of the iterative algorithm. Similar to before, we use the criterion for all intermediate results $\boldsymbol{X}_k$ in combination with the Frobenius norm to determine a single value $\left\|\boldsymbol{X}_k^2 - \boldsymbol{I}\right\|_F$ that we use as convergence metric.

The results are shown in Figures 4.6 and 4.7. Overall we see a very similar behavior as for the inverse p-th root iteration. The method converges also when using low-precision floating-point and fixed-point arithmetic and the convergence rate in the first iterations is very similar among all precisions.

4.2.2 Error Accumulation

While the convergence behavior of the Newton-Schulz iteration suggests robustness against errors induced by low-precision arithmetic, these results needs to be verified by comparing the computed solution against a golden result computed

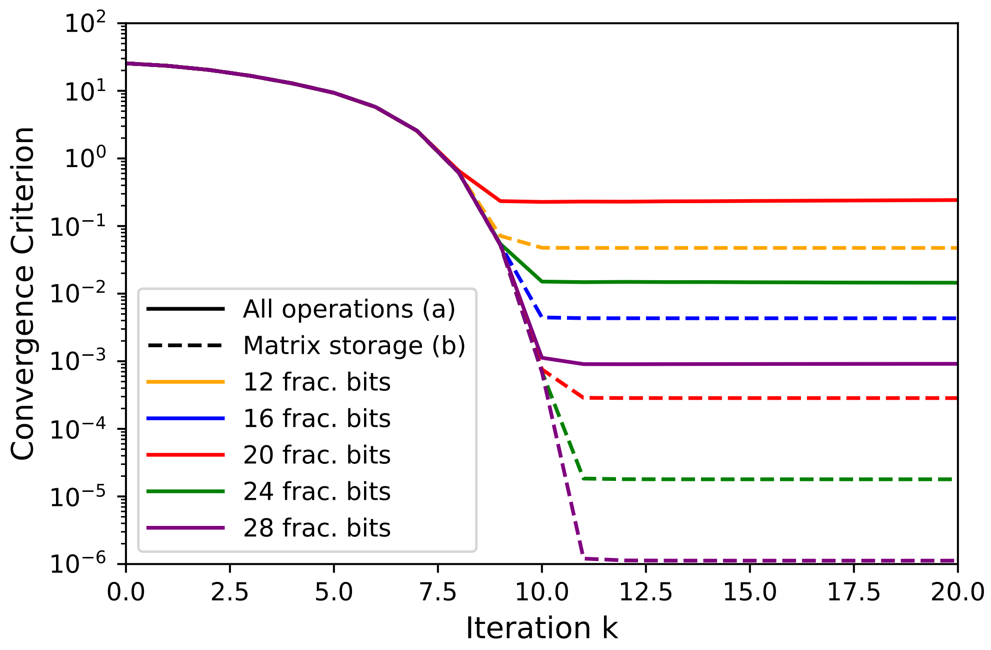

Figure 4.7: Computation of sign function: Convergence behavior when using custom-precision fixed-point for (a) all arithmetic operations and (b) only for storage of intermediate results ($N = 768$).

using double-precision arithmetic. We therefore now look at the absolute error in the computed result, again using the Frobenius norm to derive a single value $\left\|\boldsymbol{X}_k - \mathrm{sign}(\tilde{\boldsymbol{H}})\right\|_F$ as error metric.

The results are shown in Figures 4.8 and 4.9. We observe that after the algorithm has converged, the quality of the result degrades with further iterations when using low-precision arithmetic for the calculations. The reason is that the Newton-Schulz iteration within the iteration step does not take the original matrix into account. This allows small errors to accumulate over time. While the algorithm still produces an involutory matrix after each additional iteration, this result moves further and further away from the sign function of the initial input matrix. When using low precision only for the storage of intermediate results, this effect showed to be neglectable for the scenario evaluated here.

4.3 Summary of Findings

The presented results demonstrate the overall resiliency of the examined algorithms against errors introduced due to low-precision arithmetic and storage. It stands out that the number of iterations required to reach a certain precision does not significantly increase with the amount of approximation. This sets the examined algorithms apart from other iterative methods like the preconditioned conjugate gradient method which was modified to run on approximate hardware by Schöll et al. [79] and showed to require additional iterations when using approximation.

This opens up great opportunities for the acceleration of applications that require the calculation of inverse p-th roots of matrices or the matrix sign function, such as the LSDFT method discussed in this work. The availability of high-performance

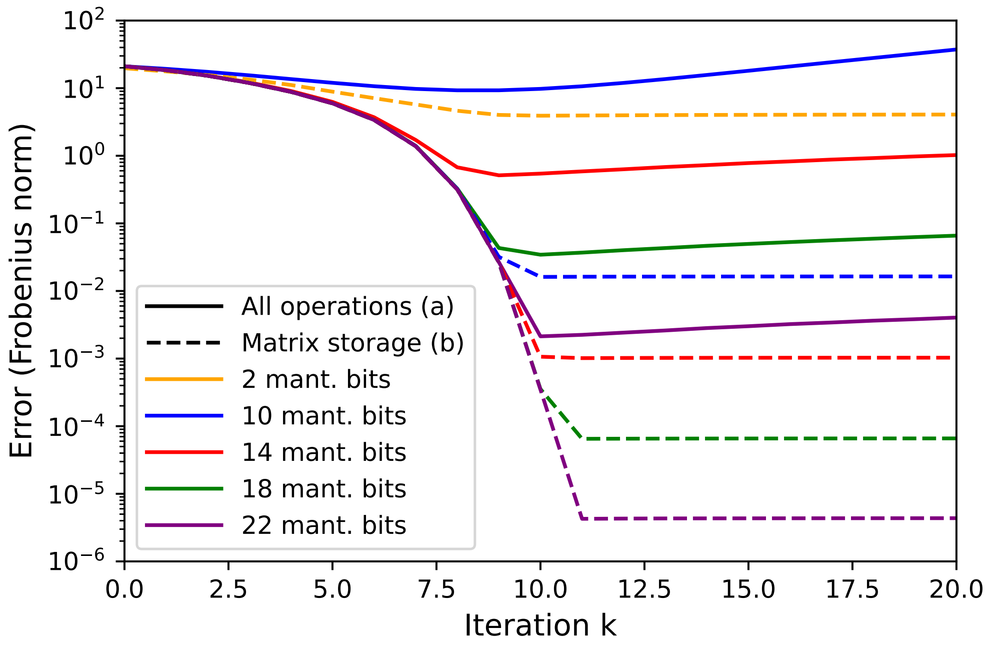

Figure 4.8: Computation of sign function: Error accumulation when using custom-precision floating-point for (a) all arithmetic operations and (b) only for storage of intermediate results ($N = 768$).

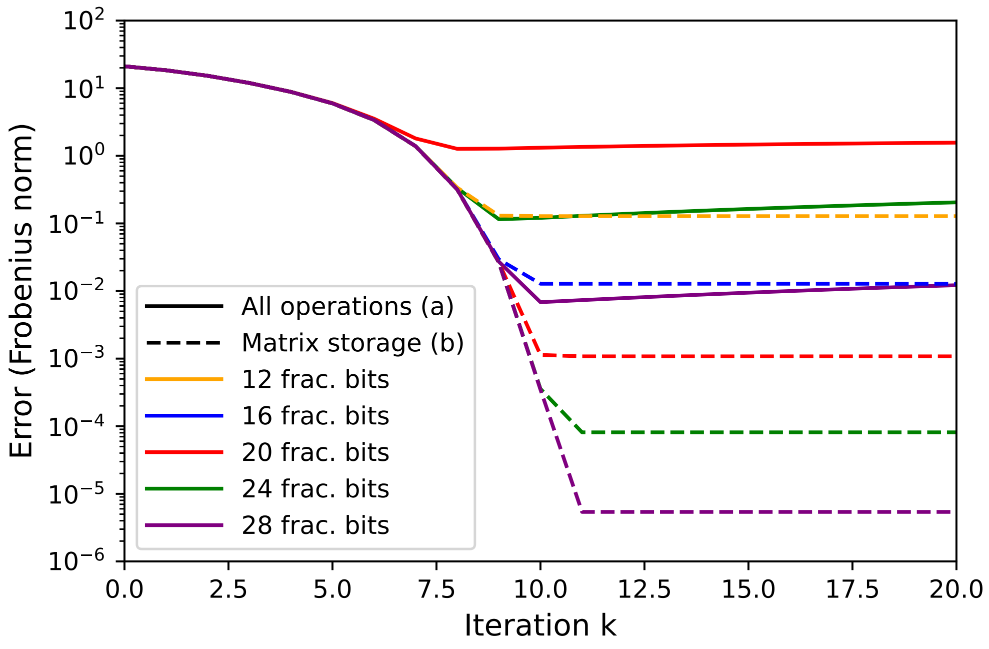

Figure 4.9: Computation of sign function: Error accumulation when using custom-precision fixed-point for (a) all arithmetic operations and (b) only for storage of intermediate results ($N = 768$).

low-precision arithmetic in modern hardware accelerators like GPUs and FPGAs allows to run these algorithms with reduced precision. Since both discussed methods are based around matrix multiplications, this directly increases the achievable performance. The possibility to store approximate results without reducing the result quality too much allows the reduction of required memory space and bandwidth on these devices.

For computation of the matrix sign function using the Newton Schulz method, we have seen that errors introduced by using low precision accumulate instead of being corrected in later iterations, because the original matrix is not considered by all iterations. It is therefore important to avoid any superfluous iterations. In contrast, when computing the inverse p-th root, errors can be corrected in later iterations. In particular, results obtained using low precision can be refined into a precise solution in very few additional iterations using higher precision arithmetic.

Chapter 5

Submatrix Method: Algorithmic Approximation of Matrix Functions

After having examined the effect of low-precision arithmetic on selected iterative schemes, we now introduce a second level of approximation. In certain applications, matrix functions need to be computed on very large matrices which are not fully populated but sparse. In particular, this holds for the matrices handled in Linear Scaling DFT. To efficiently deal with these matrices, it is essential to exploit the sparsity of the matrices in all demanding computations. One particular challenge is fill-in generated when matrix functions are applied to a sparse input matrix, causing the result becoming more densely populated.

In this chapter, we present a new method to compute approximate solutions for unary matrix functions of large, sparse, symmetric matrices. It enforces the sparsity pattern of the input matrix to the result, avoiding any fill-in. At the same time the method allows massive parallelization of the required computations. We call the method proposed in this work the *Submatrix Method*.

The method has originally been thought of by Stephan Mohr [80, Section 5.1.7.2] for the computation of inverse square roots and later been generalized and evaluated by the author of this work. Large parts of this chapter have been presented at the *Platform for Advanced Scientific Computing Conference (PASC)* in 2018 and are part of the proceedings published by ACM [A6]. Parts of this chapter have also been published at the *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)* in 2020 [A2].

This chapter is structured as follows: In Section 5.1, the fundamental idea of the Submatrix Method as well as an algorithmic description are presented. In Section 5.2 we evaluate the practical impact of using the Submatrix Method onto the results. We look at performance and scaling of the method on a theoretical basis in Section 5.3 and on a practical basis in Section 5.4.

5.1 Algorithm Description

The fundamental idea of the Submatrix Method is to transform a matrix operation f on a large, sparse $n \times n$ matrix $\boldsymbol{A}$ into n operations on smaller dense matrices. The overall scheme is shown in Figure 5.1 and can be summarized as follows:

1. For each column $i \in 1 \ldots n$, a principal submatrix $\boldsymbol{a}_i$ is assembled by removing all rows and columns j from the original matrix, where $A_{ji} = 0$. The size of the submatrix $\boldsymbol{a}_i$ is therefore determined by the number of nonzero elements in the i-th column of $\boldsymbol{A}$.

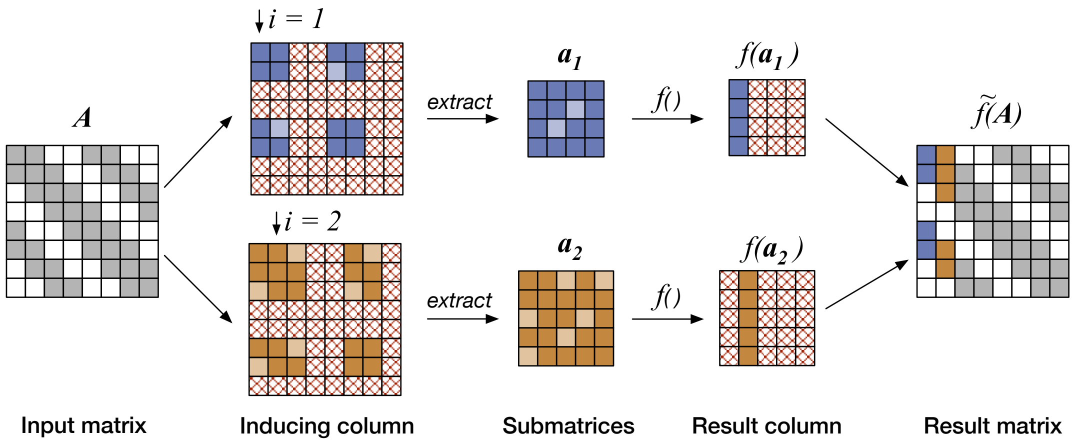

Figure 5.1: Submatrix Method: Overview.

2. The matrix operation of interest is performed on all submatrices $\boldsymbol{a}_i$, resulting in result submatrices $f(\boldsymbol{a}_i)$.

3. Let k be the column within $\boldsymbol{a}_i$ that contains the values originating from the i-th column of $\boldsymbol{A}$. Then the values from the k-th column of $f(\boldsymbol{a}_i)$ are used to assemble the i-th column of the approximate result matrix $\tilde{f}(\boldsymbol{A})$, while retaining the sparsity pattern of the original input matrix $\boldsymbol{A}$.

In the following, we describe the single steps of the method in more detail.

5.1.1 Building the Submatrices

Implementing the Submatrix Method typically follows a constructive approach (i.e., copying elements of the original matrix) instead of the destructive approach (i.e., removing rows and columns of the original matrix) described above. An procedure to build all submatrices constructively from the original input matrix is described in Algorithm 5.1. To simplify comprehension of the algorithm, all matrices have a dense representation in our pseudocode. As will be discussed in Section 5.1.4, in practice a sparse representation should be used for the sparse input and output matrices. Also note that, although we only discuss a column-based approach for building the submatrices, the method can as well be applied in a row-based manner since we are dealing with symmetric matrices.

To construct the j-th submatrix, the j-th column of the input matrix $\boldsymbol{A}$ is evaluated. We determine the set R of row indices i for which $A_{ij} \neq 0$. The submatrix is then constructed by taking all values A_{xy} from the input matrix where $x, y \in R$. For an input matrix of size $n \times n$ we obtain a set of n submatrices. The size of each submatrix is determined by the number of nonzero elements in the corresponding column of the input matrix.

5.1.2 Performing Submatrix Operations

For all of the submatrices, we now perform the operation which should originally be performed on the input matrix, i.e., we either invert all submatrices, calculate their inverse p-th roots or compute the matrix sign function. Note that the method and implementation for these submatrix operations can be freely selected and this choice is entirely orthogonal to the Submatrix Method.

Algorithm 5.1 Construction of submatrices.

```
$n \leftarrow$ number of rows/columns of input matrix
$A[1 \ldots n][1 \ldots n] \leftarrow$ input matrix
**for** $j \leftarrow 1 \ldots n$ **do**
    $R \leftarrow \{j\}$
    **for** $i \leftarrow 1 \ldots n$ **do**                      ▷ locate nonzero elements in column $j$
        **if** $A[i][j] \neq 0$ **then**
            $R \leftarrow R \cup \{i\}$
        **end if**
    **end for**
    $m \leftarrow R$.length()                           ▷ $m$: dimension of submatrix $j$
    **for** $k \leftarrow 1 \ldots m$ **do**                           ▷ assemble submatrix $j$
        **for** $l \leftarrow 1 \ldots m$ **do**
            submatrices$[j][k][l] \leftarrow A[R[k]][R[l]]$
        **end for**
    **end for**
    indices$[j] \leftarrow R$                 ▷ store indices required for result assembly
**end for**
```

Algorithm 5.2 Assembly of result matrix.

```
$n \leftarrow$ number of rows/columns of input matrix
indices$[1 \ldots n] \leftarrow$ from submatrix generation stage
submatrices$[1 \ldots n][1 \ldots ?][1 \ldots ?] \leftarrow$ result matrices
$X \leftarrow$ zeros($n \times n$)
**for** $j \leftarrow 1 \ldots n$ **do**
    $R \leftarrow$ indices$[j]$
    $m \leftarrow R$.length()                           ▷ $m$: dimension of submatrix $j$
    **for** $i \leftarrow 1 \ldots m$ **do**                       ▷ fill column $j$ of result matrix
        $X[R[i]][j] \leftarrow$ submatrices$[j][i][R$.indexof$(j)]$
    **end for**
**end for**
```

5.1.3 Assembling the Result Matrix

After having applied the matrix operation of interest to each submatrix, we have n result submatrices. From these result submatrices we assemble an approximate solution $\boldsymbol{X}$ for the whole matrix. This procedure is shown in Algorithm 5.2. Similar to the construction of the submatrices, the j-th column of the final result matrix is determined by the j-th result submatrix. We take the values from the column of the result submatrix which was originally filled with values from the j-th column of the input matrix, and copy them back to their original position in the original matrix.

5.1.4 Implementation Notes

Although the inverse (root) of a sparse matrix is typically not sparse, the approximate solution provided by the Submatrix Method exhibits the exact same sparsity pattern as the input matrix. This allows for efficient implementation of the method based on matrices in the Compressed Sparse Column (CSC) format which consists of a value array (`val`), a list of row indices (`row_ind`) and a list of column pointers (`col_ptr`). In particular, the result assembly stage can be implemented by concatenating the corresponding columns of all result submatrices to obtain the value array `val` for the approximate result matrix. `row_ind` and `col_ptr` from the input matrix can be reused for the output matrix without any changes. If the method is applied in a row-based manner, the same holds for matrices in the CSR format.

5.2 Applicability and Approximation Error

The result obtained by the Submatrix Method is only an approximation of the correct result. Whether this result is still of use for an application depends on three aspects: Is the application able to deal with results that contain a certain error? How large can this error be to be still tolerable? And how large is the error introduced into results by using the Submatrix Method?

In this section, we demonstrate the error caused by using the Submatrix Method in different scenarios. For computing the inverse p-th root, we look at randomly generated inputs as well as real-world inputs from different applications. For the matrix sign function, we test the applicability with real-world input data taken from a Linear Scaling DFT code.

5.2.1 Computation of Inverse p-th Roots

We first evaluate using the Submatrix Method for the computation of inverse p-th roots.

Error for Random Input Matrices

To get an impression about the error introduced for arbitrary symmetric positive-definite matrices, we generate random matrices $\boldsymbol{A}$ using the `sprandsym`[1] function in Matlab. This allows us to sweep over different sizes n, densities d and condition numbers κ and assess the influence of these matrix properties onto the error. For each set of these parameters, we generate ten different matrices. For all of these matrices we then use the Submatrix Method to obtain an approximate solution $\boldsymbol{X}$ for the inverse p-th root $\boldsymbol{A}^{-1/p}$. To assess the error of these results, we calculate the spectral norm of the residuals

$$\|\boldsymbol{R}\|_2 = \|\boldsymbol{X}^p\boldsymbol{A} - \boldsymbol{I}\|_2 . \tag{5.1}$$

Since for a precise solution it should hold that $\boldsymbol{X}^p\boldsymbol{A} = \boldsymbol{I}$, $\boldsymbol{R}$ is a good indicator for the introduced error. We choose the spectral norm of $\boldsymbol{R}$ as a metric because in contrast to other matrix norms like the Frobenius norm it is relatively invariant of the matrix size.

Our initial evaluation has not shown a significant influence of the density of the randomly generated matrices onto the precision of the result. We therefore neglect this parameter in the evaluation presented here, focus on matrices with density $d = 0.05$ and discuss the influence of the size and the condition number of the matrices. Figure 5.2 shows the relationship between these matrix properties and the calculated residual for $p = 1$. It shows that the error increases for matrices with larger size and larger condition numbers. For small matrices, the error stays relatively low even for higher condition numbers. Similarly, for well-conditioned matrices, the error stays low even for large matrices.

To demonstrate the latter, we now focus on well-conditioned matrices with $\kappa = 2$ and $d = 0.05$, varying only their size. Results are shown in Figure 5.3. It shows that

[1]`sprandsym(size,density,1/condition,kind)` with `kind=1`

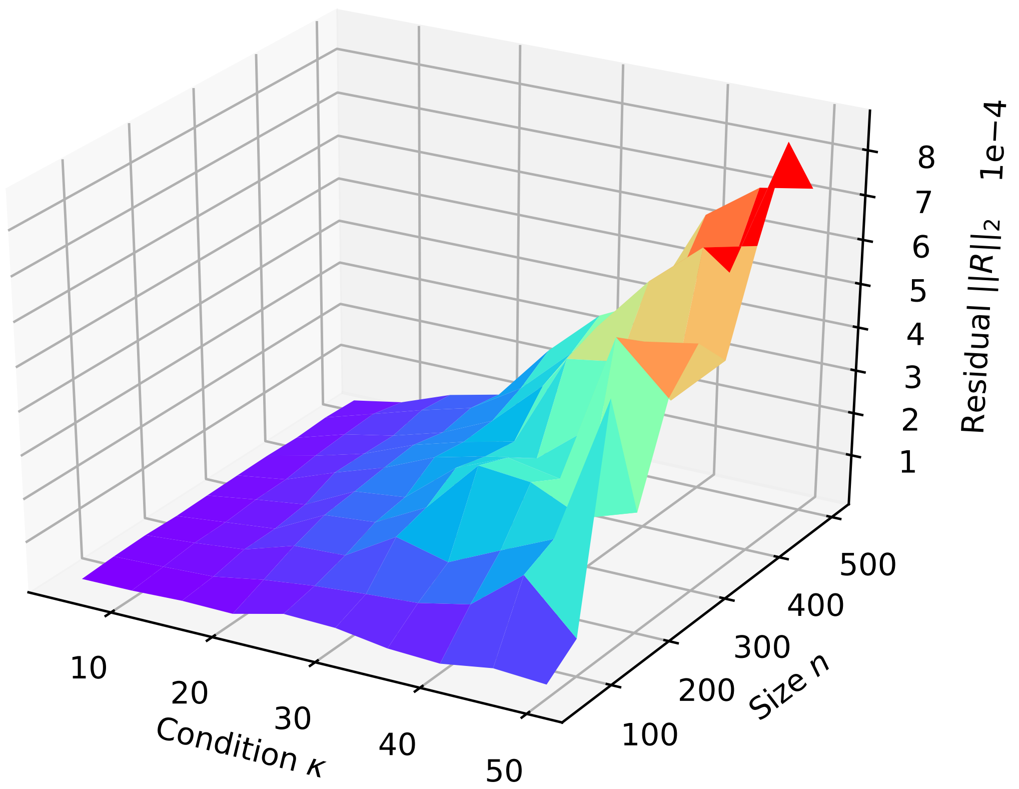

Figure 5.2: Residual for approximately calculated inverses of random matrices using Submatrix Method, for different sizes and condition numbers.

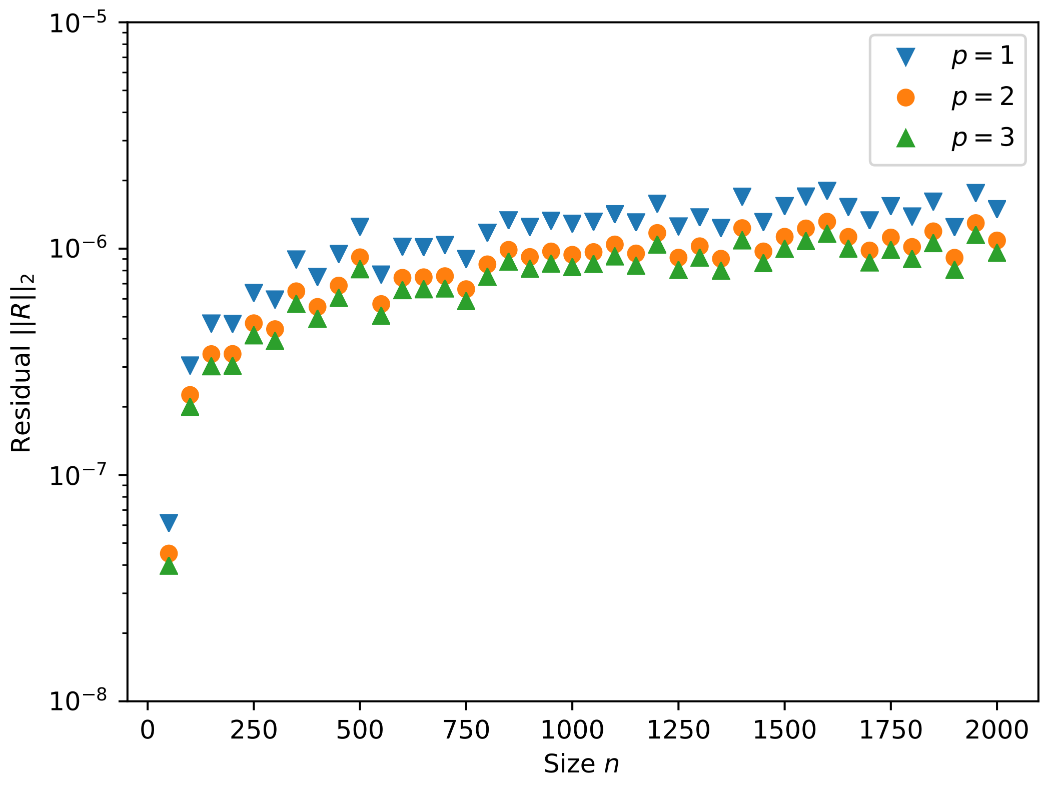

Figure 5.3: Residual for approximately calculated inverses of random matrices with $\kappa = 2$ using Submatrix Method, in relation to size of input matrix.

for a fixed condition number, the error introduced by using the Submatrix Method is limited even when further increasing the matrix size. As shown, this not only holds for calculating the inverse of a matrix but also for calculating inverse p-th roots where $p > 1$.

Orthogonalization Within Electronic Structure Codes

We demonstrated that using the submatrix method for well-conditioned matrices yields results very similar to a precisely calculated solution. However, whether errors are acceptable in an application depends on the the kind of matrices used in the application and the effect that small deviations have on the final result. In this section, we show a specific application of the Submatrix Method and demonstrate its limited influence on the final result.

We look at using the Submatrix Method for orthogonalization of the basis functions used in DFT. As described in Section 3.2, the inverse or inverse square root of the overlap matrix $\boldsymbol{S}$ is required to transform the generalized eigenvalue problem into a conventional eigenvalue problem. To evaluate the effect of using the Submatrix Method in this scenario, we use the same matrices as in Chapter 4, obtained from a Daubechies Wavelet-based DFT code [59]. The extracted overlap matrices all have condition numbers around $\kappa = 1.5$ whereas the density decreases with increasing system size as described in Section 4.1.1.

In addition to the overlap matrices $\boldsymbol{S}$, we also extract the density matrix $\boldsymbol{D}$ as well as the Hamilton matrix $\boldsymbol{H}$ from our DFT code. This allows us to calculate the

Table 5.1: Influence of using the Submatrix Method for orthogonalization of basis functions onto electronic band structure energy calculations.

Matrix size n	Density($\boldsymbol{S}$)	E_{BS}	$E_{\text{BS}}^{\text{sm}}$	ΔE_{rel}
768	0.25	-372.83597	-372.83600	6.96×10^{-8}
1536	0.12	-747.13928	-747.13933	7.60×10^{-8}
3072	0.06	-1492.25282	-1492.25297	1.01×10^{-7}
6144	0.03	-2986.25656	-2986.25683	8.76×10^{-8}
12288	0.02	-5976.31525	-5976.31576	8.64×10^{-8}
24576	0.01	-11951.19504	-11951.19598	7.85×10^{-8}

band-structure energy as

$$E_{\text{BS}} = \text{Tr}(\boldsymbol{DH}). \tag{5.2}$$

To orthogonalize the Hamilton matrix, we calculate

$$\tilde{\boldsymbol{H}} = \boldsymbol{H}\boldsymbol{S}^{-1}, \tag{5.3}$$

where $\boldsymbol{S}^{-1}$ is computed using the Submatrix Method. From this, we again calculate the band-structure energy as

$$E_{\text{BS}}^{\text{sm}} = \text{Tr}(\boldsymbol{SD}\tilde{\boldsymbol{H}}), \tag{5.4}$$

and evaluate the relative error caused by the approximate inversion of $\boldsymbol{S}$ as

$$\Delta E_{\text{rel}} = \left| \frac{E_{\text{BS}} - E_{\text{BS}}^{\text{sm}}}{E_{\text{BS}}} \right|. \tag{5.5}$$

Results are shown in Table 5.1. For all evaluated matrix sizes, the relative error caused by using the Submatrix Method for orthogonalization is rather small and throughout below or around 10^{-7}.

The band structure of the considered overlap matrices may suggest that the low deviation between E_{BS} and $E_{\text{BS}}^{\text{sm}}$ comes from the similarity between $\boldsymbol{S}$ and the identity matrix $\boldsymbol{I}$, and therefore $\boldsymbol{I}$ could be used in Equation (5.3) as an approximation for $\boldsymbol{S}^{-1}$. However, doing so leads to relative errors between 1.15×10^{-2} and 1.21×10^{-2}, i.e., five orders of magnitude higher than when using the approximate inverse provided by the Submatrix Method.

Application as Preconditioner

While we focus on LSDFT as the application of interest throughout this work, it is interesting to note that the Submatrix Method can be used in entirely different applications as well. One example, where the computation of approximate inverse (roots) of matrices is of interest, is preconditioning. We therefore now demonstrate using the Submatrix Method to process ill-conditioned matrices in order to obtain a preconditioner that can be used to iteratively solve systems of linear equations.

To demonstrate this application scenario, we obtain sparse, symmetric, positive definite matrices from the SuiteSparse matrix library [81]. We select all matrices $\boldsymbol{A}$ with size $1000 \leq n \leq 5000$ that fulfill these requirements. For these matrices we

Table 5.2: Number of iterations required to solve Equation (5.6) for different matrices $\boldsymbol{A}$ using Conjugate Gradient with different preconditioners.

Matrix	n	κ	None	SM	ILU(0)
1138_bus	1138	8.5×10^{06}	2120	151	139
bcsstk08	1074	2.6×10^{07}	—	41	27
bcsstk09	1083	9.5×10^{03}	194	56	—
bcsstk10	1086	5.2×10^{05}	—	85	182
bcsstk11	1473	2.2×10^{08}	—	273	477
bcsstk12	1473	2.2×10^{08}	—	273	477
bcsstk13	2003	1.1×10^{10}	—	409	—
bcsstk14	1806	1.2×10^{10}	—	54	262
bcsstk15	3948	6.5×10^{09}	—	177	591
bcsstk16	4884	4.9×10^{09}	464	32	35
bcsstk21	3600	1.8×10^{07}	—	224	—
bcsstk23	3134	2.7×10^{12}	—	1269	—
bcsstk24	3562	2.0×10^{11}	—	300	244
bcsstk26	1922	1.7×10^{08}	—	325	337
bcsstk27	1224	2.4×10^{04}	907	66	19
bcsstk28	4410	9.5×10^{08}	—	668	755
bcsstm12	1473	6.3×10^{05}	2790	7	12
Chem97ZtZ	2541	2.5×10^{02}	86	10	1
crystm01	4875	2.3×10^{02}	70	8	2
ex10hs	2548	5.5×10^{11}	—	—	—
ex10	2410	9.1×10^{11}	—	—	—
ex13	2568	1.1×10^{15}	—	—	—
ex33	1733	7.0×10^{12}	—	1052	—
ex3	1821	1.7×10^{10}	—	—	—
ex9	3363	1.2×10^{13}	—	—	—
mhd3200b	3200	1.6×10^{13}	—	6	3
mhd4800b	4800	8.2×10^{13}	—	6	2
msc01050	1050	4.6×10^{15}	—	—	—
msc01440	1440	3.3×10^{06}	—	89	155
msc04515	4515	2.3×10^{06}	4411	357	—
nasa1824	1824	1.9×10^{06}	—	275	264
nasa2146	2146	1.7×10^{03}	282	67	12
nasa2910	2910	6.0×10^{06}	—	282	760
nasa4704	4704	4.2×10^{07}	—	1100	570
plat1919	1919	1.2×10^{17}	—	—	—
plbuckle	1282	1.3×10^{06}	1965	76	69
sts4098	4098	2.2×10^{08}	—	67	119
Trefethen_2000	2000	1.6×10^{04}	435	6	5

Table 5.3: Influence of using the Submatrix Method for computation of the matrix sign function onto electronic band structure energy calculations. After symmetric orthogonalization of the Hamiltonian, values with an absolute value below 10^{-7} have been truncated to obtain a sparse input matrix to the matrix sign computation.

Matrix size n	Density($\tilde{\boldsymbol{H}}$)	E_{BS}	$E_{\text{BS}}^{\text{sm}}$	ΔE_{rel}
768	0.67	-372,83598	-372,83609	2.96×10^{-7}
1536	0.36	-747,13928	-747,13925	3.41×10^{-8}
3072	0.18	-1492,25283	-1492,25272	7.42×10^{-8}
6144	0.09	-2986,25658	-2986,25458	6.70×10^{-7}
12288	0.05	-5976,31522	-5976,31929	6.80×10^{-7}
24576	0.02	-11951,19484	-11951,19065	3.51×10^{-7}

solve the system

$$\boldsymbol{A}\vec{x} = \vec{b}, \quad \vec{b} = [1,1,\dots,1]^{\mathsf{T}} \tag{5.6}$$

using the Conjugate Gradient (CG) method. We set the threshold for the residual to 10^{-6} and limit the number of iterations by $2n$. Table 5.2 shows the number of iterations required for CG to converge towards a solution. For preconditioning, we use the Submatrix Method to obtain an approximate solution for

$$\boldsymbol{K} \approx \boldsymbol{A}^{-1/2}. \tag{5.7}$$

Instead of solving Equation (5.6), we now solve the system given by

$$\boldsymbol{K}^{\mathsf{T}}\boldsymbol{A}\boldsymbol{K}\vec{y} = \boldsymbol{K}^{\mathsf{T}}\vec{b} \tag{5.8}$$

using the CG method. The solution $\vec{x}$ for Equation (5.6) can then be computed as

$$\vec{x} = \boldsymbol{K}\vec{y}. \tag{5.9}$$

Again, results are shown in Table 5.2. For comparison, we also include the number of iterations required when using an ILU(0)[2] preconditioner. The results show that using the Submatrix Method for preconditioning is not only competitive to the use of ILU(0) but enables CG to converge in more of the cases.

5.2.2 Computation of the Matrix Sign Function

In addition to the computation of inverse p-th roots, we now consider using the Submatrix Method for computation of the matrix sign function. As shown in Definition 2.17, $\text{sign}(\boldsymbol{A})$ can be constructed from $\boldsymbol{A}$ and the inverse square-root of $\boldsymbol{A}^2$, suggesting that the sign function is a sufficiently related operation that is also suitable for use with the Submatrix Method. To practically evaluate if the Submatrix Method can be applied to the computation of the matrix sign function as part of a LSDFT method, we use it to purify the Hamiltonian into a density matrix and then compute the resulting band structure energy. However, since the Submatrix Method relies on a symmetric input matrix, we need to use the symmetric orthogonalization

[2]incomplete LU decomposition with zero fill-in

scheme from Equation 3.20. Hence, we compute

$$\tilde{\boldsymbol{H}} = \boldsymbol{S}^{-1/2}\boldsymbol{H}\boldsymbol{S}^{-1/2} \tag{5.10}$$

and set all matrix elements with an absolute value below 10^{-7} to zero to obtain a sparse matrix. We then recompute the density matrix $\boldsymbol{D}$ as

$$\boldsymbol{D} = \boldsymbol{S}^{-1/2}\left(\boldsymbol{I} - \operatorname{sign}\left(\tilde{\boldsymbol{H}} - \mu\boldsymbol{I}\right)\right)\boldsymbol{S}^{-1/2} \tag{5.11}$$

and compute the band structure energy as

$$E_{\mathrm{BS}} = \operatorname{tr}(\boldsymbol{D}\boldsymbol{H}). \tag{5.12}$$

We repeat these computations using the Submatrix Method when computing the matrix sign function to calculate a corresponding $E_{\mathrm{BS}}^{\mathrm{sm}}$ and evaluate the relative error. The results are shown in Table 5.3. The values for E_{BS} slightly differ from those in Table 5.1 due to the different orthogonalization scheme and the truncation of small matrix elements. $E_{\mathrm{BS}}^{\mathrm{sm}}$ exhibits a relative error smaller than 10^{-6} for all considered inputs. As part of electronic structure methods, the Submatrix Method therefore shows to be applicable not only to the computation of inverse roots as part of the orthogonalization but also to the computation of the matrix sign function when computing the density matrix.

5.2.3 Controlling the Approximation Error

We have demonstrated the use of the Submatrix Method for computation of inverse p-roots and the matrix sign function in different parts of LSDFT methods as well as preconditioning, all of which can highly benefit from the speedup and the additional parallelism and still yield good results. However, there may be applications that are less tolerant to errors but still can benefit from using the submatrix method.

If an application requires a lower error than what is provided by the solution calculated using the Submatrix Method, it may be possible to use iterative methods to refine the solution obtained from the Submatrix Method, as discussed in Chapter 4. The result obtained by using the Submatrix Method then acts as an initial guess for these iterative methods. While we validated that such a refinement of a solution generated by the Submatrix Method works in principle and converges within very few iterations, a detailed evaluation of combining the Submatrix Method with iterative methods remains for future work.

In the contrary case, if the application has a particularly high resiliency against errors in the inverse matrix, the Submatrix Method can also be combined with other approximation techniques to achieve further performance gains. Since using the Submatrix Method is orthogonal to the implementation of the operations performed on the single submatrices, these submatrix calculations can be performed in an approximate manner as well. Using an iterative method, precision can be scaled by the number of iterations. Additionally, calculations can be performed using low-precision arithmetic as has been shown in Chapter 4.

5.3 Complexity and Scalibility

We now want to discuss the time complexity and scalability of the submatrix method and show that, although for an $n \times n$ matrix n submatrices need to be processed, it can still provide a significant reduction in time required for determining a matrix inverse, its inverse p-th root or its sign function. Note that for simplicity the following sections only discuss matrix inversion. However, the results apply to the computation of inverse p-th roots and the matrix sign function as well.

The considerations in this section only hold if the nonzero elements are spread relatively evenly over all columns (or rows) of the input matrix. An obvious counterexample are arrowhead matrices, where using the Submatrix Method cannot provide any speedup since the first submatrix has the same size as the original input matrix.

5.3.1 Single-Threaded Scenario

We first want to discuss the general complexity of matrix inversion, both using conventional methods and using our proposed Submatrix Method. While from a theoretical standpoint, inversion of matrices is not harder than multiplication, and therefore $\mathcal{O}(n^{2.81})$ using Strassen's algorithm [82, Ch. 4.2], or even $\mathcal{O}(n^{2.373})$ using Coppersmith and Winograd's algorithm [83], in practice methods such as Gaussian elimination or building and using the LU decomposition for inversion which have time complexity $\mathcal{O}(n^3)$ are commonly used. In the following, we define $I(n)$ as the time required for a precise matrix inversion, abstracting from a concrete implementation.

For a sparse $n \times n$ input matrix, using the Submatrix Method requires performing n matrix inversions for smaller but dense matrices. To be more efficient in a single-threaded application scenario, these submatrices have to be significantly smaller than the original input matrix. In the following, we assume a uniformly filled, sparse input matrix. Let d be the density of this matrix, then the average size $m \times m$ of the submatrices is determined by $m = d \cdot n$. If the density d is small enough, such that

$$n \cdot I(d \cdot n) < I(n), \tag{5.13}$$

then the Submatrix Method has lower run time than a precise inversion, even in a single-threaded scenario.

We now want to determine, by what rate the density d has to decrease so that for increasing matrix sizes the asymptotic run time does not grow faster than using conventional methods for matrix inversion. We therefore assume that matrix inversion has at least time complexity n^2, i.e., $I(n) = \Omega(n^2)$. To fulfill Equation (5.13), it then needs to hold that for sufficiently large n

$$\begin{aligned} n \cdot (d \cdot n)^2 &< n^2 \\ d &< n^{-0.5}. \end{aligned} \tag{5.14}$$

From this we can deduce the following asymptotic relation:

$$d = \mathcal{O}(n^{-0.5}) \Rightarrow S(n,d) = \mathcal{O}(I(n)), \tag{5.15}$$

where $S(n,d)$ is the time required to calculate an approximate inverse using the Submatrix Method. If d decreases faster than with rate $n^{-0.5}$, then $S(n,d)$ increases slower than $I(n)$ for larger n. Using methods where $I(n) = \Omega(n^3)$ further relaxes the requirements on d, in that $d = \mathcal{O}(n^{-1/3})$ suffices to fulfill Equation (5.13) for large n.

Note that we neglect the time required for building the submatrices and assembling the final result matrix, as their influence on execution time is negligible compared to the involved matrix inversions in asymptotic considerations.

5.3.2 Parallel Execution of Submatrix Operations

Although using the Submatrix Method can reduce execution time even in a single-threaded environment for very sparse matrices, its strength is to allow massively parallel execution. All submatrix operations are entirely independent from each other such that the inversion of an $n \times n$ matrix can be distributed over n compute nodes. Each compute node can construct its own submatrix from the input matrix. The final result matrix has to be assembled on a single node but as described in Section 5.1.4, this step consists of a simple concatenation of n arrays. Communication between nodes is only required for initial data distribution and for the final collection of all results. Provided that n compute nodes can be used for execution of the algorithm, a speedup is already achievable if all submatrices are significantly smaller than the original input matrix, i.e., each column of the original matrix contains a significant fraction of zero-elements.

5.3.3 Application to Electronic Structure Methods

A major target for our method are Linear Scaling DFT methods that rely on the nearsightedness of electronic matter. With respect to the matrices for which an inverse (root) or the sign function needs to be calculated, this means that while for growing systems the total number of matrix elements increases with n^2 where n is the matrix dimension, the density of the matrix decreases linearly with n^{-1}. Consequently, the number of nonzero elements in the matrix increases only linearly with n.

Based on this fact, the Submatrix Method is particularly suitable for solving these problems. In particular, since for $n > 1$

$$n^{-1} < n^{-0.5} \tag{5.16}$$

holds, the density of matrices decreases faster than required in Equation (5.15). From that it follows that the asymptotic run time of the Submatrix Method in a single-threaded environment is limited by that of a precise inversion for the applications discussed here.

Again, the strong advantage of the Submatrix Method is the possibility of parallel execution on many compute nodes. In the case of linear scaling methods in density functional theory, this means that for growing systems the execution can be parallelized onto more and more nodes while the size of the single submatrices stays constant. As long as the number of compute nodes can be scaled with n as well, the overall execution time can even be held constant.

5.4 Performance Evaluation

To evaluate the performance and scalability of the proposed method, we built a distributed implementation using MPI and OpenMP. We run this implementation on 65 compute nodes of the OCuLUS cluster described in Section 2.1.1. We use one node as a control node, leaving the remaining 64 nodes with a total of 1024 CPU cores for handling the workload. In the following, we first describe details of our implementation, and then present results obtained from running our implementation on our compute cluster.

5.4.1 Implementation Details

Our implementation makes use of Intel MPI [84] to distribute work over a large number of compute nodes and to collect all results in order to build up the final result matrix. The MPI rank 0, in the following called main process, reads the input matrix stored in CSC format from persistent storage into memory. Metadata such as an identifier for the matrix, as well as its total size and number of nonzero elements, are then sent via `MPI_Bcast` to all nodes.

Data Distribution and Work Assignment

There are different possible ways to make the input matrix available to all other MPI ranks, which we call worker processes in the following. In principal, it would be sufficient to send single submatrices to the workers which then perform the inversion. In this case, all submatrices would have to be constructed within the main process, which would clearly present a bottleneck. Instead, we make the whole input matrix available to all worker processes which then autonomously construct their submatrices. In our environment all systems have access to a shared file system which allows all processes to read in the input matrix from persistent storage. Since this scenario cannot generally be assumed, we additionally implemented distribution of data via `MPI_Bcast` to all worker processes. We found that, for the data we use in our evaluation, both variants provide comparable performance.

Assuming we have w workers, each worker needs to process $x = n/w$ submatrices. In our implementation, each worker processes a contiguous set of submatrices, i.e., the worker with rank k is responsible for submatrices $(k-1)x$ to $kx-1$. Therefore, depending on its rank and the total number of ranks, each worker process can determine autonomously, which of the submatrices it has to process.[3] It builds the submatrix according to Algorithm 5.1 and calls the Linear Algebra PACKage (LAPACK) [85] functions `dgetrf` to obtain an LU decomposition and `dgetri` to calculate the inverse of the submatrix. In our evaluation we use Intel MKL [86] as a highly optimized implementation for these LAPACK routines. After inversion, the worker selects the section of the result matrix which is relevant for the final result matrix and stores it in a buffer. Since the main process just needs to concatenate these buffers to create the final result matrix, a single call to `MPI_Gatherv` is sufficient to perform data collection and assembly after all submatrices have been processed.

Multi-Threading using OpenMP

The described implementation already allows to distribute the load over many nodes and CPU cores. To utilize multiple CPU cores on a single node, multiple MPI ranks could be placed on a node, or multiple cores could be used for processing a single submatrix by using a multi-threaded LAPACK implementation. Having multiple MPI ranks on the same node comes at the cost of data duplication in memory and overall increased MPI communication load. Using multiple cores for processing a single submatrix operation has shown to provide lower speedup than using these additional cores to process more submatrices in parallel.

In our implementation, we therefore use OpenMP to process c submatrices in parallel on a node featuring c CPU cores. We do so by calling all submatrix operations within an OpenMP parallel for loop:

[3]Note that if the number of worker processes does not divide the size of the matrix, some workers need to process one additional submatrix.

```
#pragma omp parallel for schedule(dynamic)
```

To allow OpenMP to fully utilize the available CPU cores, we explicitly disable the multi-threading functionality provided by Intel MKL by performing a call to `mkl_set_num_threads(1)`.

Discussion of our Implementation

As described, each worker process is responsible for a contiguous set of submatrices and all workers are responsible for the same number (± 1) of submatrices. This can lead to workload imbalance between the different workers, if the input matrix exhibits a pattern such that certain sets of columns contain significantly more or significantly fewer nonzero values than other sets of columns. It is important to note that this is a limitation of our implementation and not a conceptual issue of the proposed Submatrix Method. In practice, there are different ways to deal with this issue in order to create an optimized implementation that does not exhibit this load imbalance:

Shuffling input matrices: To balance the load between all worker processes, the mapping between submatrices and workers can be shuffled randomly. Clusters of full columns which result in larger submatrices would then not be assigned to a single worker but distributed over all workers. This can, for example, be implemented using a pseudo-random but deterministic permutation, so that each worker can still autonomously determine the submatrices it is responsible for.

In our implementation, each worker concatenates the results of its submatrix operations in a buffer which is then sent as a whole to the main process. The main process therefore only needs to collect and concatenate w arrays for w worker processes. If submatrices are shuffled, collection and concatenation of n arrays would be required in the main process instead. Apart from this, there is no additional computational effort required for this load balancing technique.

Dynamic work scheduling: Instead of assigning a fixed set of submatrices to a worker process, work can be scheduled dynamically. Each worker could request work packages from the main process using MPI. The size of these work packages can be chosen in the range from one single submatrix up to n/w submatrices in order to trade off load balancing and additional communication effort. Concepts similar to OpenMP's *guided* scheduling could also be implemented to minimize scheduling overhead. Note that in our implementation, we already use dynamic work scheduling for the parallel processing of multiple submatrices on a single node by using OpenMP's *dynamic* scheduler.

Availability

The described prototypic implementation of the Submatrix Method has been made available under MIT license [B5]. The archive also contains the scripts that have been used in the following evaluation.

5.4.2 Results

We now evaluate our implementation wrt. scalability and overall performance.

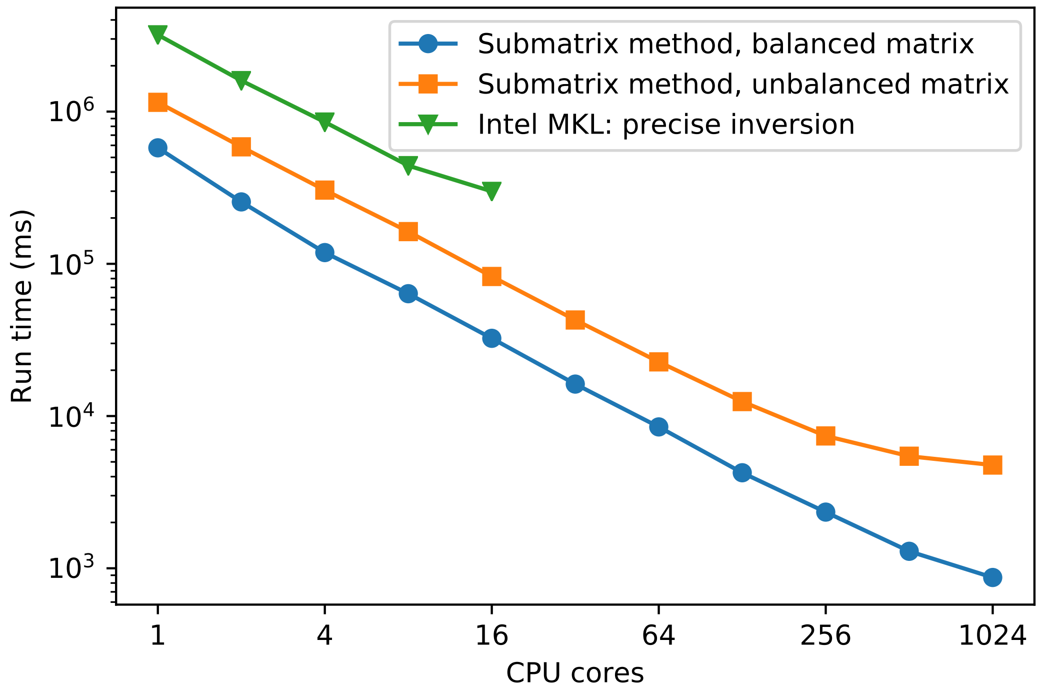

Figure 5.4: Scalability of the Submatrix Method for a random matrix of size $n = 32768$ and density $d = 0.01$.

Scalability with the Number of CPU Cores

We use our implementation of the Submatrix Method to calculate an approximate inverse of multiple random matrices with size $n = 32768$, condition number $\kappa = 2$ and density $d = 0.01$. We vary the number of utilized CPU cores in the range from 1 to 1024 and measure the total wall clock time required to obtain a result. We consider a set of balanced matrices whose columns have roughly the same number of nonzero elements[4] and a set of unbalanced matrices which exhibit visible patterns in the distribution of values[5]. Results are shown in Table 5.4 and Figure 5.4. As a reference, we also show the time required for a precise matrix inversion using Intel MKL's implementation of the `dgetrf` and `dgetri` routines, utilizing up to 16 CPU cores on a single node.

The results show that the Submatrix Method overall scales well over a large number of processors. Comparing the data for balanced and unbalanced matrices, two distinct effects can be observed:

1. Even for a low number of CPU cores, the Submatrix Method performs better for balanced matrices. The reason for this is that if some columns contain significantly more nonzero elements than others, the resulting submatrices are larger in size and the time to process them increases cubically with their size.

2. For the imbalanced matrices in our scenario, the curve starts to flatten at around 256 cores and scaling beyond 512 cores provides diminishing returns. The reason for this is that the number of submatrices per worker becomes small enough such that load imbalance between workers has an increasing effect.

[4]generated using Matlab's `sprandsym(size,density,1/condition,kind)` with `kind=2`
[5]generated using Matlab's `sprandsym(size,density,1/condition,kind)` with `kind=1`

Table 5.4: Time in ms required for inversion of a matrix with size $n = 32768$ and density $d = 0.01$ using the Submatrix Method on 1–64 nodes (1–1024 cores).

	Balanced matrix		**Unbalanced matrix**	
Cores	Wall time	Speedup	Wall time	Speedup
1	578,140	1.0	1,150,366	1.0
2	255,081	2.3	586,778	2.0
4	118,534	4.9	304,941	3.8
8	63,644	9.1	162,792	7.1
16	32,405	17.8	82,571	13.9
32	16,216	35.7	42,760	26.9
64	8,485	68.1	22,692	50.7
128	4,242	136.3	12,447	92.4
256	2,339	247.2	7,402	155.4
512	1,293	447.1	5,447	211.2
1024	870	664.5	4,765	241.4

This effect could be countered by implementing some form of load balancing, as discussed in Section 5.4.1.

For over 512 cores, even for the balanced matrices in our scenario the additional speedup is limited. This is caused by the overall short run time of the algorithm and therefore increased influence of communication time (around 32% of the wall time).

On a single node, Intel MKL as well nearly scales linearly with the number of CPU cores. Only for 16 cores there is a slight efficiency drop, likely caused by the NUMA architecture of our compute nodes. Using ScaLAPACK [87] to distribute execution of the utilized library functions over multiple nodes may allow to further increase the number of CPU cores. However, due to increasing communication overhead, the potential for scaling is limited in this case. Related work that uses ScaLAPACK for matrix inversion, describes decreasing performance for execution on more than 64 CPU cores [88].

Run Time for Growing Matrices

We now evaluate how the total execution time develops for increasing matrix sizes, given a fixed number of CPU cores. We consider two different scenarios: a fixed density of $d = 0.01$ and matrix sizes ranging from 2^{11} to 2^{18} and a density that decreases linearly with n as encountered in LSDFT. For the latter, we set $d = 0.16 \cdot 1024/n$ and consider sizes from 2^{10} to 2^{20}.

The results of this evaluation are shown in Table 5.5 and Figure 5.5. In the table we also show the fraction of the total wall clock time spent on communication and the fraction of compute time spent on building the submatrices. Note that the assembly of the result matrix is performed implicitly by `MPI_Gatherv` and therefore accounted as communication time. It clearly shows that for matrices with linearly decreasing density, the required run time only increases linearly with the matrix size, as expected based on the discussion in Section 5.3.3. Combining this result with the possibility for linear performance scaling with the number of CPU cores, the run time can be held constant by increasing the number of cores with n for growing matrices. The data also shows that for increasing size of the submatrices, as shown in the upper half of Table 5.5, the overhead required for communication and for

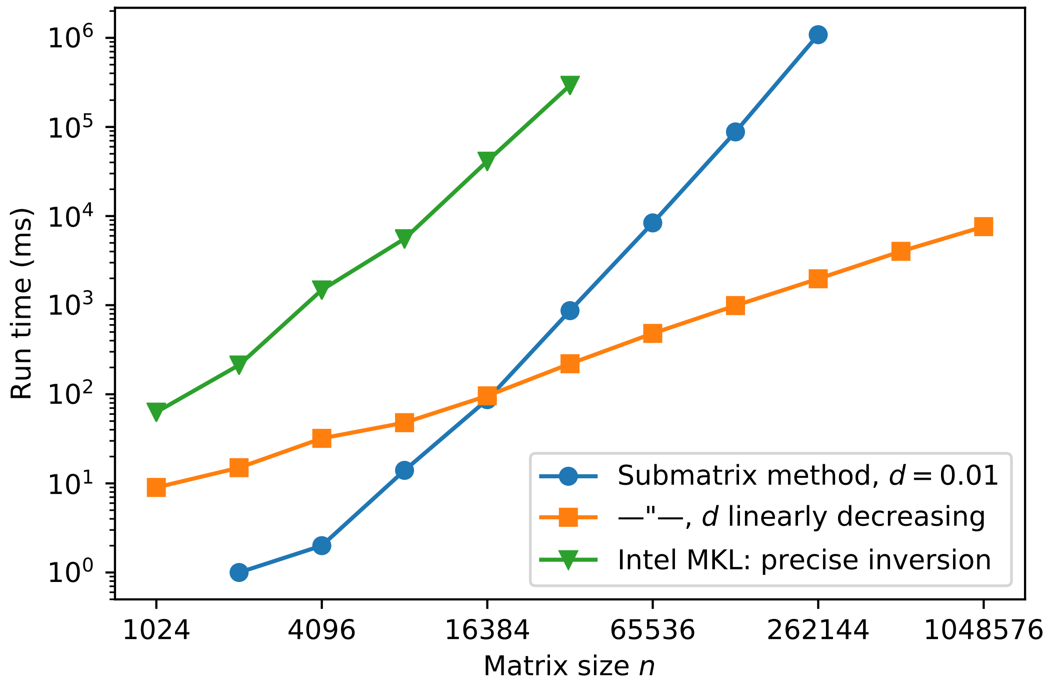

Figure 5.5: Required time for inversion of matrices using the Submatrix Method on 64 nodes (1024 cores) and Intel MKL on a single node (16 cores).

Table 5.5: Time in ms required for inversion of a matrix using the Submatrix Method on 64 nodes (1024 cores).

Size	Density	Wall time	MPI Comm.	Submatrix Construction
2,048	1.0×10^{-2}	1	—	—
4,096	1.0×10^{-2}	2	—	—
8,192	1.0×10^{-2}	14	57.1%	58.8%
16,384	1.0×10^{-2}	87	39.1%	60.5%
32,768	1.0×10^{-2}	868	31.7%	57.8%
65,536	1.0×10^{-2}	8,380	13.5%	56.8%
131,072	1.0×10^{-2}	87,977	5.0%	47.0%
262,144	1.0×10^{-2}	1,085,176	1.5%	36.8%
1,024	1.6×10^{-1}	9	22.2%	61.3%
2,048	8.0×10^{-2}	15	26.7%	61.0%
4,096	4.0×10^{-2}	32	37.5%	60.5%
8,192	2.0×10^{-2}	48	33.3%	60.3%
16,384	1.0×10^{-2}	96	38.5%	60.4%
32,768	5.0×10^{-3}	220	51.7%	60.9%
65,536	2.5×10^{-3}	482	54.4%	61.4%
131,072	1.3×10^{-3}	990	54.7%	62.2%
262,144	6.3×10^{-4}	1977	55.2%	63.0%
524,288	3.1×10^{-4}	4020	56.8%	63.7%
1,048,576	1.6×10^{-4}	7609	49.9%	64.0%

building the submatrices decreases. For fixed-size submatrices, the overhead stays relatively constant.

5.5 Summary of Findings

In this chapter we presented the Submatrix Method, which can be used to calculate an approximate solution for unary matrix functions, such as the inverse of matrices, as well as inverse p-th roots and the matrix sign function, for large sparse matrices. Following the idea of Approximate Computing, it allows the result to deviate from an exactly calculated solution in order to utilize the sparsity of the input matrix and to allow massively parallel execution of the involved calculations. For an $n \times n$ matrix, the workload can be distributed over n nodes. The method is particularly interesting for LSDFT where for growing matrices their density decreases linearly at the same time. In this case, the Submatrix Method exhibits a linear increase in execution time for growing systems. As long as the number of available CPU cores can be scaled with the same rate, execution time can even be held constant. The method however also shows to be suitable for entirely different areas of application, such as preconditioning, and is likely to be applicable to different unary matrix functions as well. It therefore should be considered as a general approximation method beyond the scope of this work.

Chapter 6

Integration of the Submatrix Method into CP2K

After we have demonstrated the suitability of the Submatrix Method for different computational problems based on test inputs and we have evaluated the performance and scaling based on a prototype using MPI and OpenMP in Chapter 5, we now discuss its integration into CP2K and thereby evaluate its behavior in a realistic application scenario. More specifically, we show how the Submatrix Method can be applied to the calculation of the matrix sign function within density matrix based LSDFT to yield a novel massively-parallel Linear Scaling DFT method.

In Section 3.3.3 we have seen that the dominant computational hotspot in CP2K's LSDFT code is the computation of the matrix sign function. Although inverse square roots are explicitly computed in CP2K for orthogonalization of the Hamiltonian, this orthogonalization is performed only once before running the SCF cycle and therefore not repeated as often as the sign computation. We therefore focus on computing the matrix sign function using the Newton Schulz method throughout this chapter.

We begin by describing a necessary extension of the matrix sign definition used in CP2K in Section 6.1, followed by a detailed description of the specific requirements posed by CP2K and the resulting implementation of the Submatrix Method as part of the LSDFT method in Section 6.2. The accuracy and parallel scaling properties of the new method for representative benchmark cases are then evaluated in Section 6.3. The contents of this chapter have been presented at the *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)* in 2020 and have been published by ACM and IEEE as part of the corresponding proceedings [A2].

6.1 Extension of the Matrix Sign Function Definition in CP2K

When using the Submatrix Method to compute the matrix sign function, we need to compute the sign function for all of the generated submatrices. We therefore need to make sure that the sign function is actually defined for all submatrices.

The matrix sign function can be calculated for square matrices which have no eigenvalues on the imaginary axis. In CP2K, the sign function is only applied to square matrices and by construction all submatrices generated as part of the Submatrix Method are also square. However, the requirement that no eigenvalues are on the imaginary axis, cannot be guaranteed in CP2K, since the chemical potential μ can be an arbitrarily chosen in a grand-canonical ensemble and it directly influences all eigenvalues. In CP2K, the definition of the sign function is therefore extended, such that in addition to Equation 2.5

$$\lambda_i(\operatorname{sign}(\boldsymbol{A})) = \begin{cases} +1, & \text{if } \operatorname{Re}(\lambda_i(\boldsymbol{A})) > 0 \\ -1, & \text{if } \operatorname{Re}(\lambda_i(\boldsymbol{A})) < 0 \end{cases} \tag{2.5 revisited}$$

we also handle the case

$$\lambda_i(\operatorname{sign}(\boldsymbol{A})) = 0, \quad \text{if } \operatorname{Re}(\lambda_i(\boldsymbol{A})) = 0. \tag{6.1}$$

When using the sign function for computation of the density matrix, this extension is consistent with the physically underlying Fermi function (see Equation 3.18), since

$$\lim_{\varepsilon \to \mu} \left(\exp\left(\frac{\varepsilon - \mu}{k_B T} \right) + 1 \right)^{-1} = \frac{1}{2}. \tag{6.2}$$

The modification allows us to compute the sign function for all square matrices and therefore also for all submatrices.

6.2 Implementation of the Submatrix Method Within CP2K

A basic implementation of the Submatrix Method has already been described in Section 5.4. This implementation however makes several simplifying assumptions and therefore can only serve as a reference for an implementation within CP2K.

One of these assumptions is that the input matrix is known to all MPI ranks so all of them can create their own submatrices independently. In contrast, in CP2K the matrices are stored in the DBCSR format and therefore in a distributed fashion. Ranks only know about their own blocks of the data. Another difference coming from the DBCSR storage format is that the sparsity of the matrix is only exploited at the level of blocks and not single elements of the matrix. Lastly, the matrices in CP2K have a certain sparsity pattern that depends on the represented chemical system. This pattern needs to be taken into account to minimize data transfers and required floating-point operations and to balance the load between all ranks.

In the following Sections 6.2.1–6.2.5, we discuss all of these implementation details. Afterwards, we discuss the operation performed on all submatrices in Sections 6.2.6 and 6.2.7.

6.2.1 Overview

To enable all ranks to assemble their submatrices, a couple of initialization steps need to be performed. Major steps are the following:

Create Global View on the Sparsity Pattern of the Matrix

For the input matrix in DBCSR format, each rank only knows which rank is responsible for which blocks of the matrix. However, whether a block is zero or if it contains data is only known to the rank holding that block. To assemble submatrices, each rank needs to know the sparsity pattern of the entire matrix. We achieve this by creating a list of nonzero blocks in a COOrdinate format (COO) representation, which stores row and column of each nonzero block. This list is deterministically sorted by columns and rows such that it is identical on all ranks. This way, the position of a nonzero block in this COO representation also serves as a unique ID for the block throughout our implementation.

Create a Mapping Between Ranks and Submatrices

The responsibility for creating and processing the submatrices needs to be distributed among all ranks. This happens in a deterministic fashion such that all ranks know which submatrices are solved by which rank. The details of this mapping will be discussed later on.

Determine Required Matrix Block Transfers

To assemble a submatrix, the corresponding rank needs a copy of all nonzero blocks that are part of this submatrix. Therefore, we iterate through all blocks of the locally processed submatrices, determine the origin rank and collect the IDs of blocks to be transferred. Additionally, we store a list of all blocks that will be filled with the results calculated locally. These blocks need to be copied back to their origin after finishing the computations.

6.2.2 Data Transfers

To keep overall communication time low, any unnecessary data transfers need to be avoided. We do so by ensuring that matrix blocks are transferred only once between ranks and that the number of blocks required by a rank is minimized.

Deduplication of Data Transfers

In general, data exchange could be implemented as part of the submatrix assembly, such that only blocks required for the currently processed submatrix need to be exchanged and stored. However, we know that blocks are included in multiple submatrices and these blocks would have to be transferred multiple times between the same ranks. To avoid any duplicate transfers, we make sure that blocks are only transferred once between a pair of nodes by exchanging all required blocks already during the initialization. Each rank stores all blocks that are required for its own submatrices in a local buffer such that submatrices can be assembled without further communication. With this approach we avoid duplicate data transfers and make sure that submatrix assembly becomes a purely local operation.

Minimization of Memory Use and Data Transfers

To minimize the amount of data that needs to be held in memory and to further reduce the amount of data transfers, ranks should process a set of *similar* submatrices, such that reuse of locally buffered blocks is maximized. Submatrices $\boldsymbol{a}_i$ and $\boldsymbol{a}_j$ are similar if they share many blocks. This is the case, when columns i and j of the original sparse matrix exhibit a similar sparsity pattern, which depends on the index order of atoms. A similar sparsity pattern of neighboring columns can for example be achieved by indexing the atoms in an order that minimizes the real-space distances between adjacent indices. For a system constructed of smaller cells of atoms as building blocks, the orthogonalized Kohn-Sham matrix usually exhibits a banded structure if the indexing is consecutive in the building blocks. Figure 6.1 shows an example for the orthogonalized Kohn-Sham matrix for a system of 864 water molecules built up of blocks of 32 water molecules. Thus, in this case a minimization of

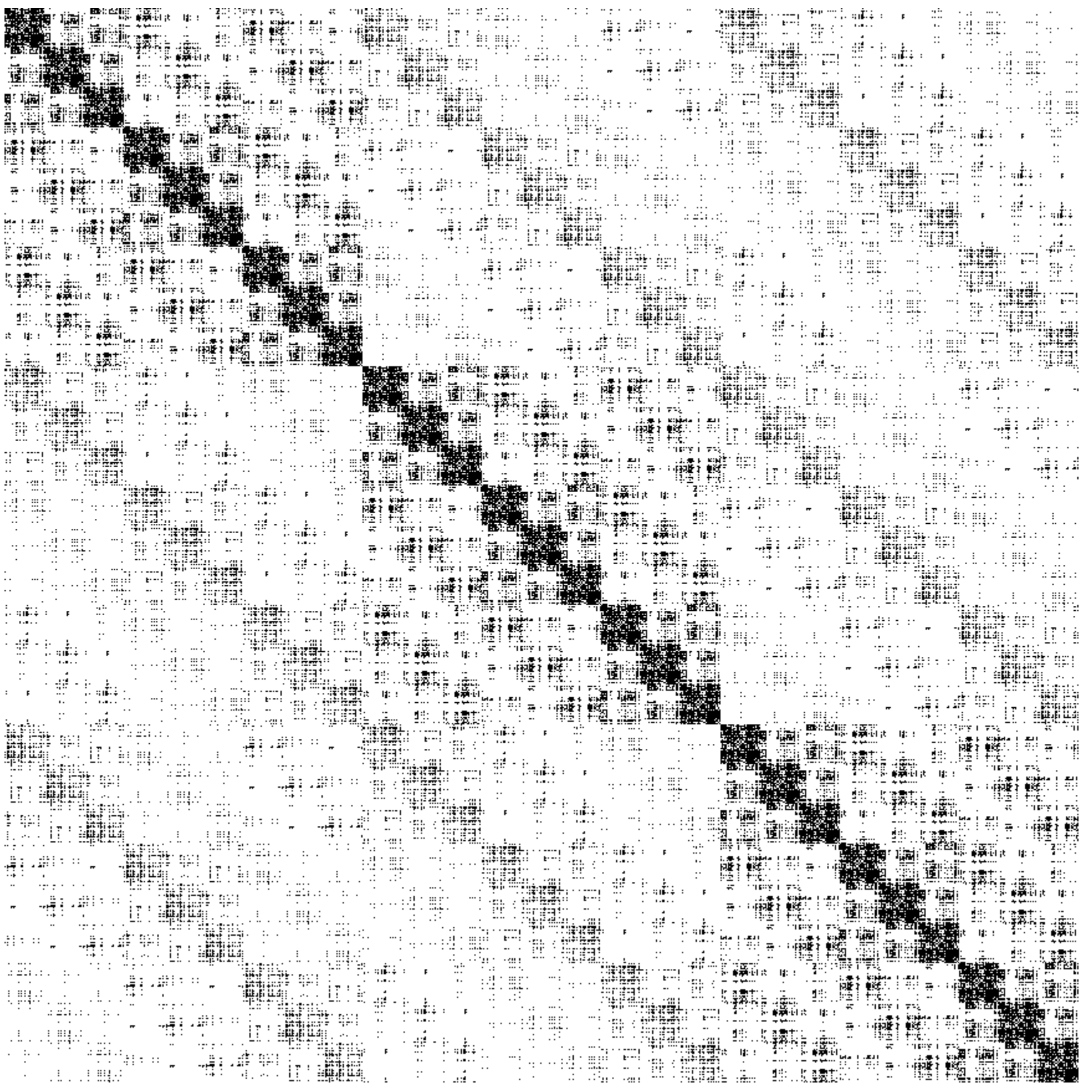
Figure 6.1: Block-based sparsity pattern of an orthogonalized Kohn-Sham DBCSR matrix $\tilde{\boldsymbol{H}}$ for 864 H_2O molecules, using the SZV basis set and a cutoff value of 10^{-5}, exported from CP2K. Each column corresponds to a water molecule and each black area corresponds to a block that contains at least one nonzero matrix element.

memory use can be achieved by assigning a consecutive sequence of submatrices to each rank.

6.2.3 Minimization of Floating-Point Operations

To maximize throughput, we also want to minimize the floating-point operations required to obtain a result in addition to data transfer and reuse optimizations. The Submatrix Method allows a certain trade-off here: While the original idea of the Submatrix Method is to generate a submatrix for each column of the original matrix, there is also the possibility to generate submatrices from multiple columns.

So far, we apply the Submatrix Method at the level of DBCSR blocks. This way, we automatically generate submatrices from b consecutive columns, where b is the block width of the corresponding column of the DBCSR matrix. However, there is still the possibility to further split up submatrices to get submatrices for single columns or to combine submatrices built from single block columns.

Splitting up Submatrices

After assembling a submatrix at the level of DBCSR block columns, it is a regular, densely stored matrix which however still may be sparse. The Submatrix Method can be applied a second time at the level of single columns to split the submatrix into even smaller, more dense sub-submatrices. Note that it is not required to generate sub-submatrices for all columns of the submatrix. Since submatrix $\boldsymbol{a}_i$ only provides values to the overall solution that originate from block column i from the original matrix, it is sufficient to build and solve sub-submatrices for columns originating from block column i.

Generating Submatrices From Multiple Block Columns

We can also go the other way and combine even more block columns to lower the total number of submatrices N_S that need to be assembled and processed. In contrast to splitting up submatrices, combination of multiple submatrices already needs to be taken into account during the initial assembly of submatrices. Different strategies can be followed to combine block columns. Assuming a relatively homogenous system where all submatrices are of roughly equal size, one can follow a simple heuristic. Given that the operation performed on all submatrices requires $\mathcal{O}(n^3)$ floating-point operations, the total number of floating-point operations can be estimated from the size of the first submatrix as

$$\text{operations} \sim n_i^3 \cdot N_b / i, \tag{6.3}$$

where n_i is the dimension of the submatrix generated from the first i block columns and N_b is the dimension of the original matrix in blocks. During the initialization of the Submatrix Method, one can determine all possible n_i and then choose the i for which the model provides the lowest number of floating-point operations.

This simple heuristic is however prone to produce unfavorable results when the system does not conform to the assumptions taken, e.g., when the system is not homogeneous and therefore the sparsity of different block columns varies significantly. In our publication [A2], we describe a more sophisticated heuristic based on a clustering of atoms based on their position in real space. This method also allows generating submatrices for a block columns that are not directly adjacent to each other in the matrix.

In CP2K, the block size of the DBCSR matrix is chosen based on physical properties of the system, e.g., one block of the DBCSR matrix corresponds to an atom or a molecule. Based on this property, we can assume single blocks to be relatively dense, such that splitting up submatrices provides no benefit. Still, combining multiple block columns to create a submatrix can generate a benefit in the amount of required arithmetic operations. For the rest of this work, we will stick to a simple clustering of consecutive block columns.

6.2.4 Shared-Memory Parallelism

CP2K supports both distributed memory parallelism and shared memory parallelism. To make use of shared memory parallelism, we use OpenMP to parallelize parts of the initialization of the Submatrix Method. Routines for the generation of a specific submatrix and for extracting results from a result submatrix are implemented in a thread-safe way, such that these steps as well as solving the submatrices can be implemented using thread-parallelism in the calling code. In our use of the Submatrix Method for solving the sign function, we distribute the work among all available threads using OpenMP work sharing clauses.

6.2.5 Load Balancing

Depending on the chemical system, block sizes and sparsity pattern of the DBCSR matrix, the submatrix dimensions can vary between different columns of the matrix. For example, a large molecule in solution may have different atom species and exhibit different interactions between its atoms than within the solvent. The matrix columns containing the atoms of the large molecule will therefore induce much larger submatrices. For achieving a good load balance, we therefore cannot just assign the same number of submatrices to each rank but need to consider the estimated computing time to reduce the deviation in execution time between different ranks.

We employ a greedy algorithm to assign submatrices to ranks such that they have similar load. As discussed in Section 6.2.2, we need to find a mapping that assigns one consecutive chunk of submatrices to each rank. Our approach computes the expected number of floating-point operations assuming that processing a submatrix takes $\mathcal{O}(n^3)$ FLOP and assigns submatrices to ranks as long as their load is expected to be lower than $\text{FLOP}_{\text{total}}/\#\text{ranks}$. Additionally, we make sure that each rank obtains at least one submatrix.

6.2.6 Sign Calculation Based on Diagonalization

So far, we have described the implementation of the Submatrix Method in CP2K. The submatrices can generally be processed using the same mechanism as originally performed on the orthogonalized Kohn-Sham matrix, e.g., by applying a Newton-Schulz iteration scheme. An alternative approach is to use diagonalization to compute the sign function of all submatrices.

To guarantee that the input to the sign function is diagonalizable, we require it to be symmetric. However, although both $\boldsymbol{S}^{-1}$ as well as $\boldsymbol{H}$ in Equation 3.19 are symmetric, their product and therefore the input to the sign function is not. We therefore follow the same approach as in Section 5.2.2 and modify the orthogonalization scheme in CP2K to use the symmetric Löwdin orthogonalization from Equation 3.20.

The density matrix $\boldsymbol{D}$ can then be computed as

$$\boldsymbol{D} = \frac{1}{2}\boldsymbol{S}^{-1/2}\left(\boldsymbol{I} - \operatorname{sign}\left(\boldsymbol{S}^{-1/2}\boldsymbol{H}\boldsymbol{S}^{-1/2} - \mu\boldsymbol{I}\right)\right)\boldsymbol{S}^{-1/2} \tag{3.20 revisited}$$

As described in Section 2.3.3, the matrix sign function can be conceived as an application of the scalar sign function to all eigenvalues. Instead of using iterative schemes, it therefore can be computed using an eigendecomposition of the matrix, for which we use the BLAS routine `dsyevd` in our implementation. We use the extended definition of the scalar sign function, i.e., including

$$\operatorname{sign}(0) = 0, \tag{6.4}$$

and utilize the fact that all submatrices are symmetric:

$$\begin{aligned} \boldsymbol{A} &= \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\mathsf{T}} \\ \Lambda'_{i,i} &= \operatorname{sign}(\Lambda_{i,i}) \\ \operatorname{sign}(\boldsymbol{A}) &= \boldsymbol{Q}\boldsymbol{\Lambda}'\boldsymbol{Q}^{\mathsf{T}}. \end{aligned} \tag{6.5}$$

For computing the sign function of our dense submatrices, we found this approach to be superior to iterative approaches on CPUs. Also, it allows to easily apply our method to systems at finite temperature by replacing the sign function in Equation 6.5 by the Fermi function.

6.2.7 Adaptation of the Method to Canonical Ensembles

As described so far, the Submatrix Method is a method for grand canonical computations where the chemical potential μ is fixed, as is the original Newton-Schulz approach. However, solving the submatrices using eigendecompositions allows us to adapt the method also for canonical ensembles, where μ needs to be dynamically adjusted to compute a density matrix that matches a certain fixed number of electrons.

Using a grand canonical method for a canonical ensemble requires us to compute the total number of electrons as

$$n_{\text{elec}} = \operatorname{Tr}\left(\frac{1}{2}\left(\boldsymbol{I} - \operatorname{sign}\left(\boldsymbol{S}^{-1/2}\boldsymbol{H}\boldsymbol{S}^{-1/2} - \mu\boldsymbol{I}\right)\right)\right) \tag{6.6}$$

and to compare it against the actual number of electrons of the underlying system. If the number of electrons deviate, μ needs to be adjusted, e.g., using a simple bisection algorithm. Normally this would require recalculation of the sign function in each bisection step, leading to massively increased run times depending on how many bisection steps are required. Having computed the eigendecomposition of all submatrices allows us to perform this adjustment of μ without recomputing the sign function or the eigendecomposition in each step, as shown in Algorithm 6.1.

Based on the determined value for μ, the sign function for all submatrices can be computed following the scheme from Equation 6.5 while adjusting all $\Lambda_{i,i}$ according to the new μ.

In practice, storing all eigendecompositions may be infeasible due to the high memory requirements. However, as shown in Algorithm 6.1, calculating μ only requires certain rows from the matrix of eigenvectors $\boldsymbol{Q}$. Most of the additional memory requirements can be saved by only keeping these rows in memory. The downside of this approach is that after determination of the correct value for μ, the

Algorithm 6.1 Adjustment of μ based on eigendecompositions of all submatrices. The single decompositions $\boldsymbol{Q\Lambda Q}^{\mathsf{T}}$ only need to be computed once.

```
$\mu_{\text{corr}} \leftarrow 0$
repeat
    $n_{\text{elec}} \leftarrow 0$
    for all submatrices $\boldsymbol{a}^{n\times n} = \boldsymbol{Q\Lambda Q}^{\mathsf{T}}$ do
        for all diagonal elements $\lambda_i$ of $\boldsymbol{\Lambda}$ do
            $\lambda'_i \leftarrow \operatorname{sign}(\lambda_i - \mu_{\text{corr}})$
        end for
        for all columns $k$ of $\boldsymbol{a}$ that contribute to the
                sparse result matrix do
            $n_{\text{elec}} \leftarrow n_{\text{elec}} + \frac{1}{2} - \frac{1}{2}\sum_{l=1\ldots n} {Q_{k,l}}^2 \cdot \lambda'_l$
        end for
    end for
    update $\mu_{\text{corr}}$ based on error of $n_{\text{elec}}$
until error of $n_{\text{elec}}$ is sufficiently small
$\mu \leftarrow \mu + \mu_{\text{corr}}$
```

submatrices need to be decomposed again in order to compute the final result for the sign function. Still, this approach is superior to recomputing the decomposition in each step of the μ-bisection and we consider it a practical compromise.

6.2.8 Availability

The implementation described in this chapter has been included in the official CP2K repository and is therefore available to all of its users. A copy of the exact implementation used in the following evaluation has been archived [B4]. Additionally, scripts and input data used in the evaluation are publicly available [B3].

6.3 Evaluation

To evaluate our method, we use it within CP2K to compute the density matrix from the Kohn-Sham matrix, following Equation 3.20 with a fixed value for μ, i.e., we consider a grand canonical ensemble. The submatrices are solved using our diagonalization approach, as described in Section 6.2.6. For comparison, we look at the default alternative for grand canonical computations which is a 2nd-order Newton-Schulz scheme to compute the sign function of the sparse DBCSR matrix. To make results comparable, we use the same symmetric orthogonalization approach from Equation 3.20, also when using Newton-Schulz iterations.

We perform all computations on a typical benchmark system, which contains liquid water, and use a single-zeta valence basis set (*SZV-MOLOPT-SR-GTH*). The benchmark systems are generated from a fixed-size region containing 32 H_2O molecules that is repeated in each dimension by a certain factor *NREP*. The total number of atoms in the system therefore increases with NREP^3. Due to the fact that the heuristic described in Section 6.2.3 has not been integrated into CP2K yet, submatrices have instead been combined based on a simple greedy heuristic that only considers using a single block column or combining multiples of these basic regions.

For evaluation of the Submatrix Method, all measurements have been run using a single thread per MPI rank. For measurements of the standard Newton-Schulz

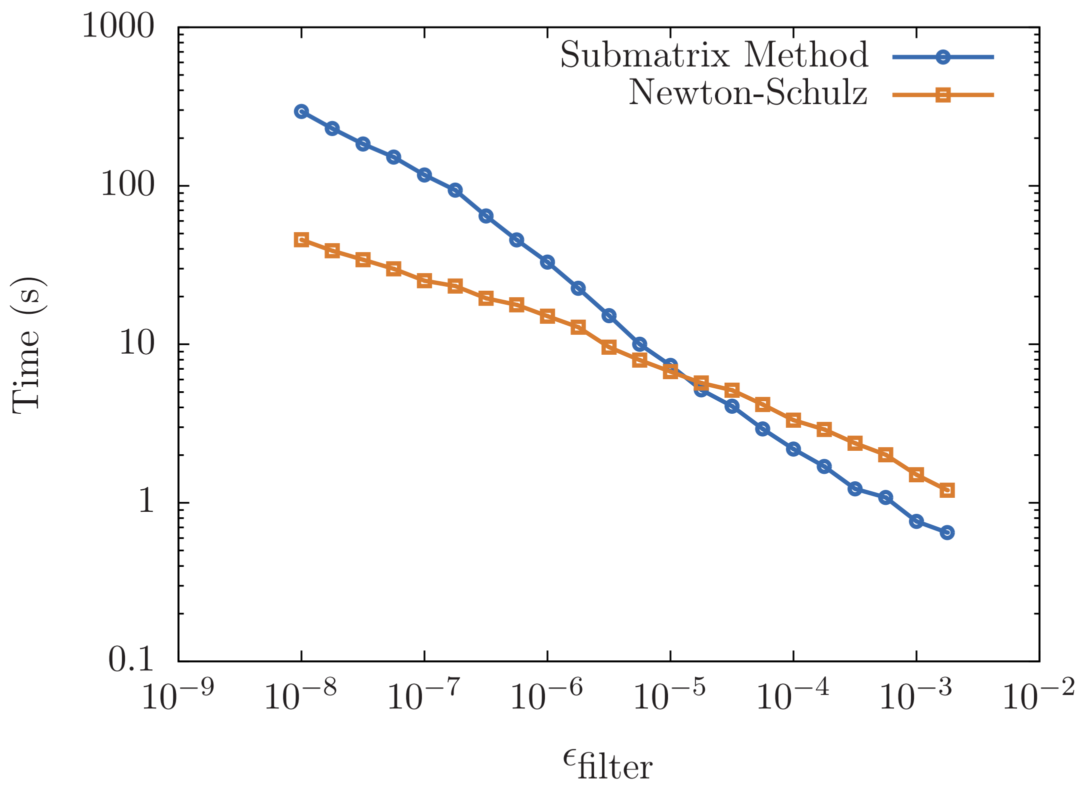

Figure 6.2: Run time of Submatrix Method and 2nd-order Newton-Schulz for various $\varepsilon_{\text{filter}}$ on 80 cores for a system of 20,736 atoms.

method, we used eight ranks per node and five threads per rank, as we found this combination to scale well on our infrastructure. All measurements in this evaluation have been performed on compute nodes of the Noctua 1 cluster described in Section 2.1.1.

6.3.1 Performance and Error for Various $\varepsilon_{\text{filter}}$ Thresholds

To exploit the nearsightedness in quantum mechanics, to obtain sparse matrices and therefore enable linear scaling methods, values below a certain threshold need to be neglected. In CP2K this threshold is called $\varepsilon_{\text{filter}}$ and it is configurable in the input file. For the Newton-Schulz iteration scheme, $\varepsilon_{\text{filter}}$ also determines the convergence criterion. Figure 6.2 shows the time required for computation of the density matrix based on different values for $\varepsilon_{\text{filter}}$ for a system where NREP $= 6$ (20,736 atoms) on two compute nodes (80 cores). The chosen value for $\varepsilon_{\text{filter}}$ significantly influences the run time as for higher values the matrices become more sparse. This effect is even more emphasized for the Submatrix Method which strongly benefits from the sparsity of the input matrix. For $\varepsilon_{\text{filter}} > 10^{-5}$, we observe that the Submatrix Method becomes quicker than the default Newton-Schulz approach.

Of course, also the resulting error of the cutoff needs to be taken into account. For that we compute the band-structure energy as $\text{Tr}(\boldsymbol{DH})$ (see Equation 3.22) after computation of the density matrix and compare it against a reference value computed with $\varepsilon_{\text{filter}} = 10^{-15}$. Results are shown in Figure 6.3. The Submatrix Method overall shows a resulting error similar to Newton-Schulz which means that the approximation inherent to the Submatrix Method does not negatively impact the results too much.

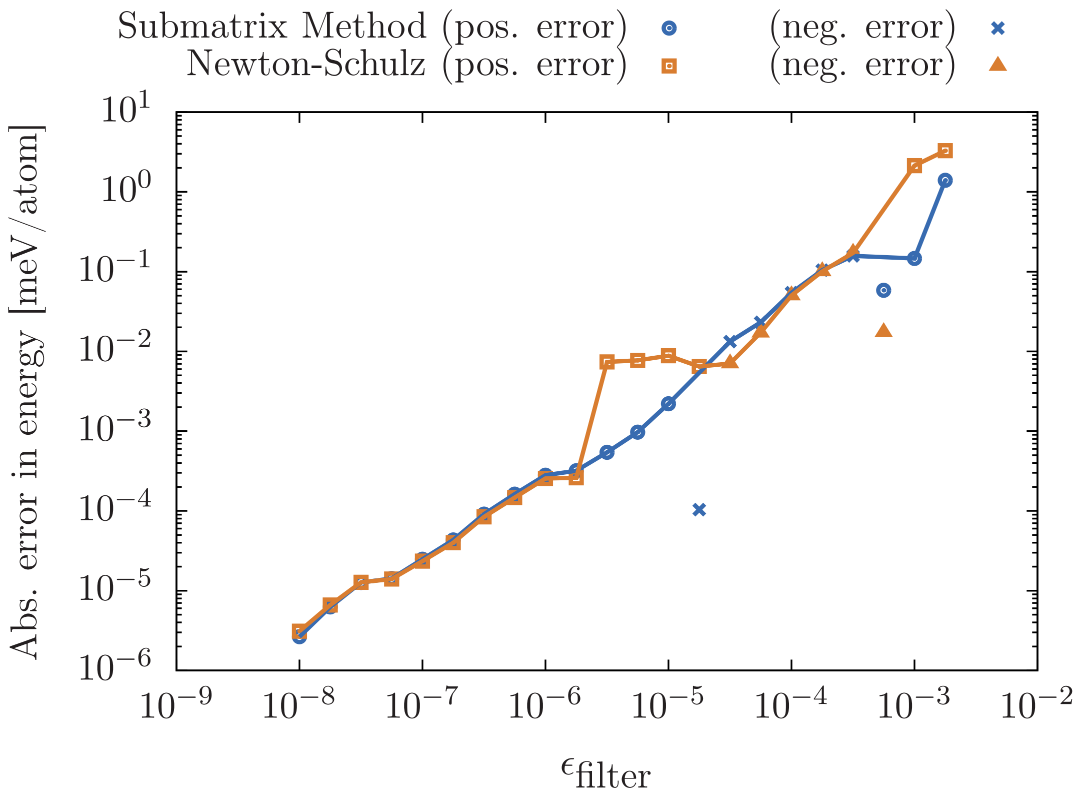

Figure 6.3: Error in energy computed by the Submatrix Method and 2nd-order Newton-Schulz for different $\varepsilon_{\text{filter}}$ for a system of 20,736 atoms. The error can be positive or negative, as shown by different markers. The plotted line serves as visual guidance, leaving out data points that show a lower absolute error due to the error transitioning between positive and negative values.

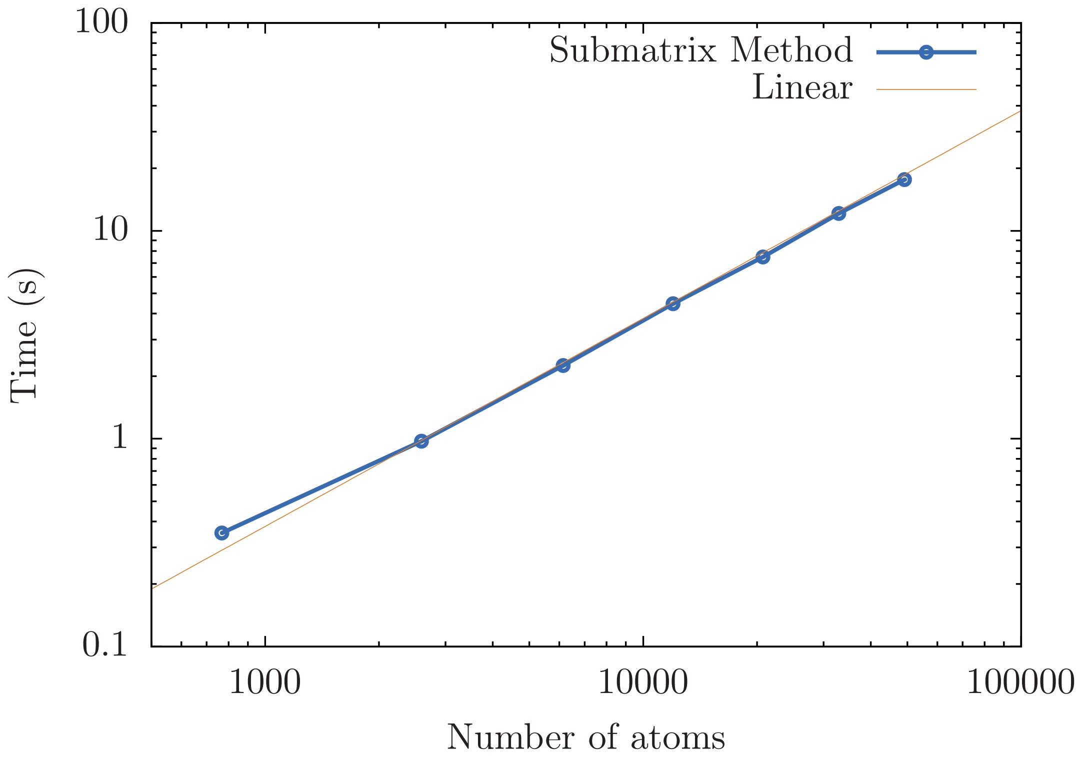

Figure 6.4: Run time of Submatrix Method for increasing system sizes on 80 CPU cores and $\varepsilon_{\text{filter}} = 10^{-5}$.

6.3.2 Scaling

We verify the linear scaling behavior of the Submatrix Method by scaling up our benchmark system from NREP = 2 (768 atoms) to NREP = 8 (49,152 atoms) while keeping the amount of computing resources constant at two nodes (80 cores). Results are shown in Figure 6.4 and match very well with a linear function.

To evaluate the strong-scaling behavior of the Submatrix Method, we take the opposite approach and scale the amount of computing resources between two nodes (80 cores) and eight nodes (320 cores) while keeping the system size fixed at NREP = 7 (32,928 atoms). Results are shown in Figure 6.5. For comparison, we also show a hypothetical perfect scaling based on the time required on two nodes and the number of nodes used. Going from two to eight nodes, we retain an efficiency of 83%.

Finally, we evaluate weak scaling, where we increase system size and the amount of compute resources at the same time. To allow more fine-grained control over the system size, we do not replicate the system in all three dimensions but instead use a sufficiently large system of 12,000 atoms (NREP = 5) as basis and further replicate it in only one dimension while increasing the number of nodes.

To put the weak-scaling efficiency into perspective, we repeat the same measurements using the standard Newton-Schulz approach, which relies on libDBCSR to scale well over many nodes. Results are shown in Figure 6.6. While there is certainly a loss in efficiency when scaling from one to 32 nodes, we see that weak-scaling efficiency is generally higher than for the default Newton-Schulz method.

6.3.3 Larger Basis Sets

To obtain a Linear Scaling DFT method as shown in the evaluation, the number of nonzero blocks in a block column needs to stay constant when growing systems.

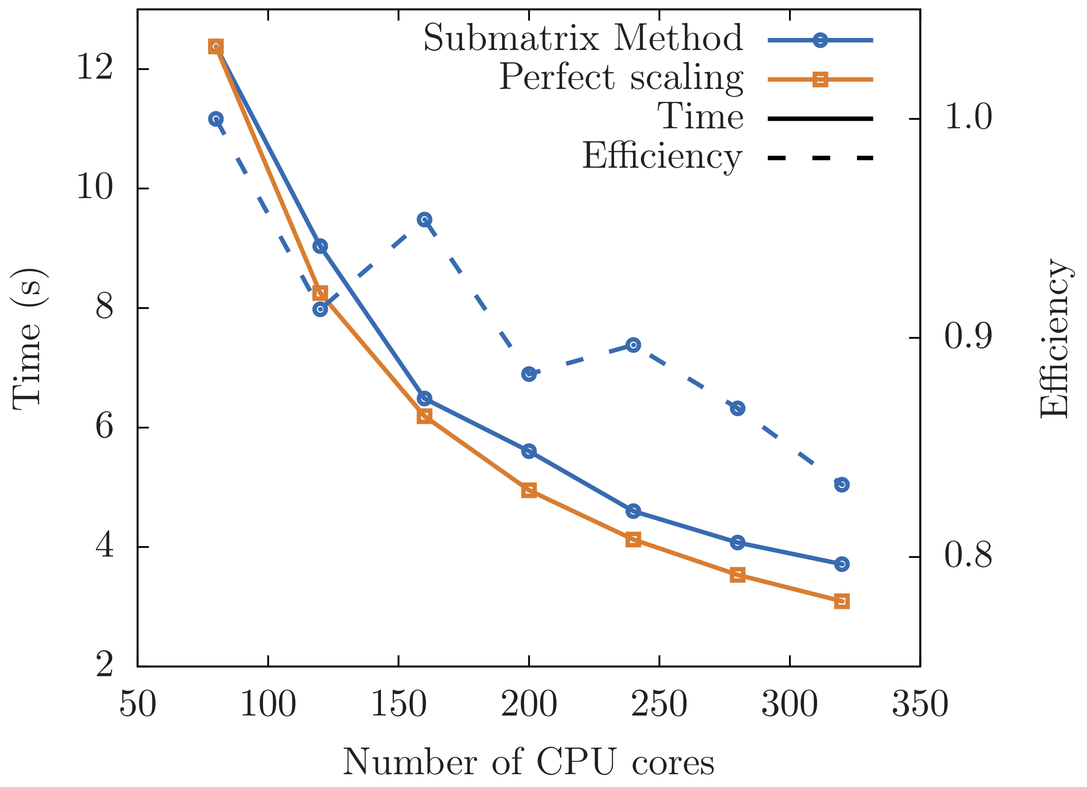

Figure 6.5: Strong scaling of Submatrix Method for 32,928 atoms and $\varepsilon_{\text{filter}} = 10^{-5}$.

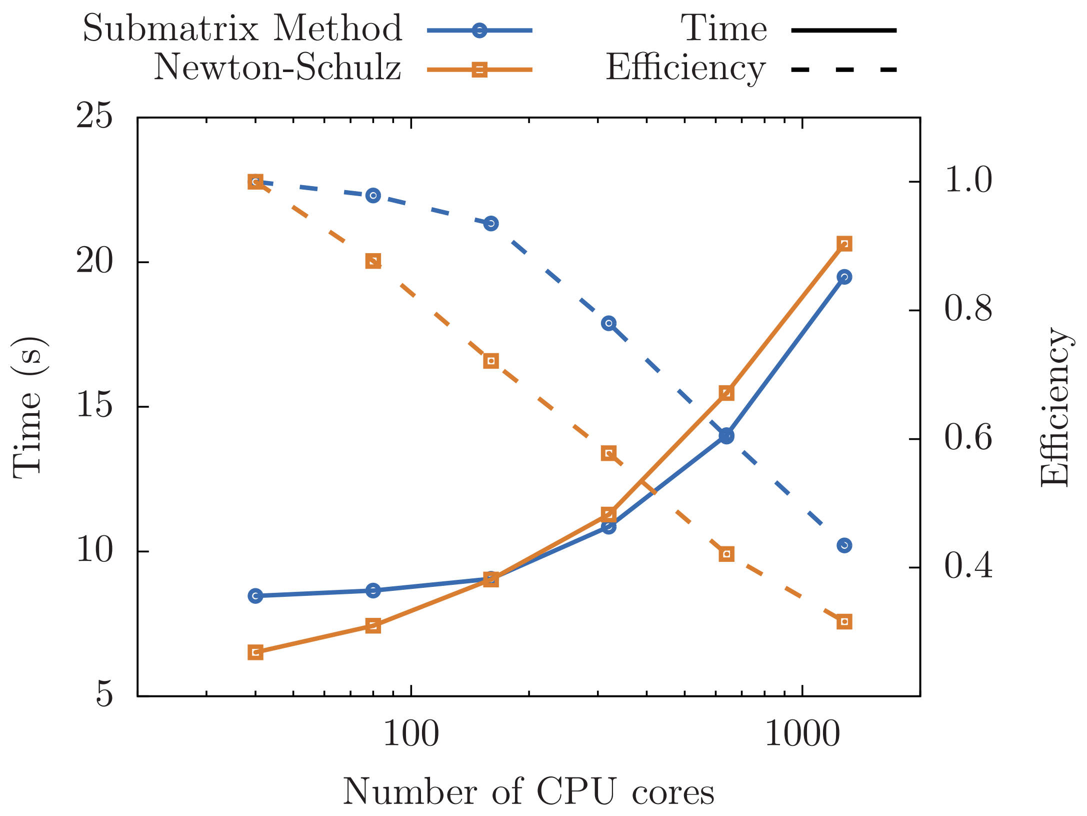

Figure 6.6: Weak scaling of Submatrix Method and 2nd-order Newton-Schulz for 12,000–384,000 atoms and 40–1280 CPU cores and $\varepsilon_{\text{filter}} = 10^{-5}$.

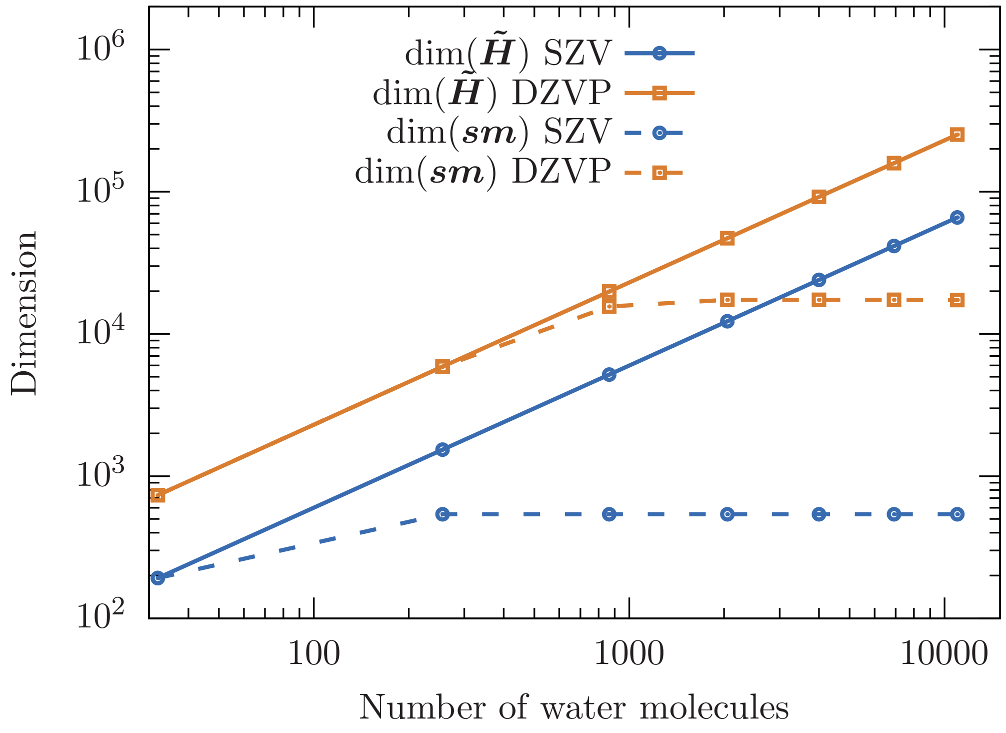

Figure 6.7: Dimension of submatrices dim(**sm**) (block-based, dashed lines) compared to the overall dimension of the orthogonalized Kohn-Sham matrix $\dim(\tilde{\boldsymbol{H}})$ (solid lines) for a cube of liquid water with periodic boundary conditions described in an SZV (blue, circles) and a DZVP (orange, cubes) basis set and a cutoff value of 10^{-5} for the matrix elements.

We obtain this behavior already for relatively small systems due to our use of short-range basis sets where matrix elements between basis functions decay rapidly with the distance of atoms they belong to. The minimum system size for which this linear scaling comes becomes effective, i.e., reaching the linear scaling regime, heavily depends on the used basis set and the chosen cutoff value.

Figure 6.7 compares the dimension of the Kohn-Sham matrix to the dimension of the submatrices for different sizes of our benchmark system using two different basis sets and a cutoff value of 10^{-5}. Going from a single-zeta valence basis (*SZV-MOLOPT-SR-GTH*) to a double-zeta valence basis (*DZVP-MOLOPT-SR-GTH*), the Kohn-Sham matrix grows due to the increased number of basis functions per atom. At the same time we observe that a larger number of water molecules is required before reaching the linear scaling regime at around 1000 water molecules, whereas for SZV-MOLOPT-SR-GTH the submatrices stop growing already at around 200 water molecules. Additionally, the obtained submatrices are significantly larger, showing dimensions above 10^4 whereas submatrices for SZV-MOLOPT-SR-GTH in this scenario have dimensions below 10^3.

In addition to the size of the submatrices, the sparsity of the generated submatrices depends on the chosen basis set as well. This correlation is shown in Figure 6.8 which for the same scenario plots the block-wise sparsity of the Kohn-Sham matrix and the resulting submatrices and additionally the element-wise sparsity of the generated submatrices. While on a block level the sparsity of the generated submatrices is very similar when using SZV-MOLOPT-SR-GTH or DZVP-MOLOPT-SR-GTH, we can observe that element-wise we obtain much more sparse submatrices when using

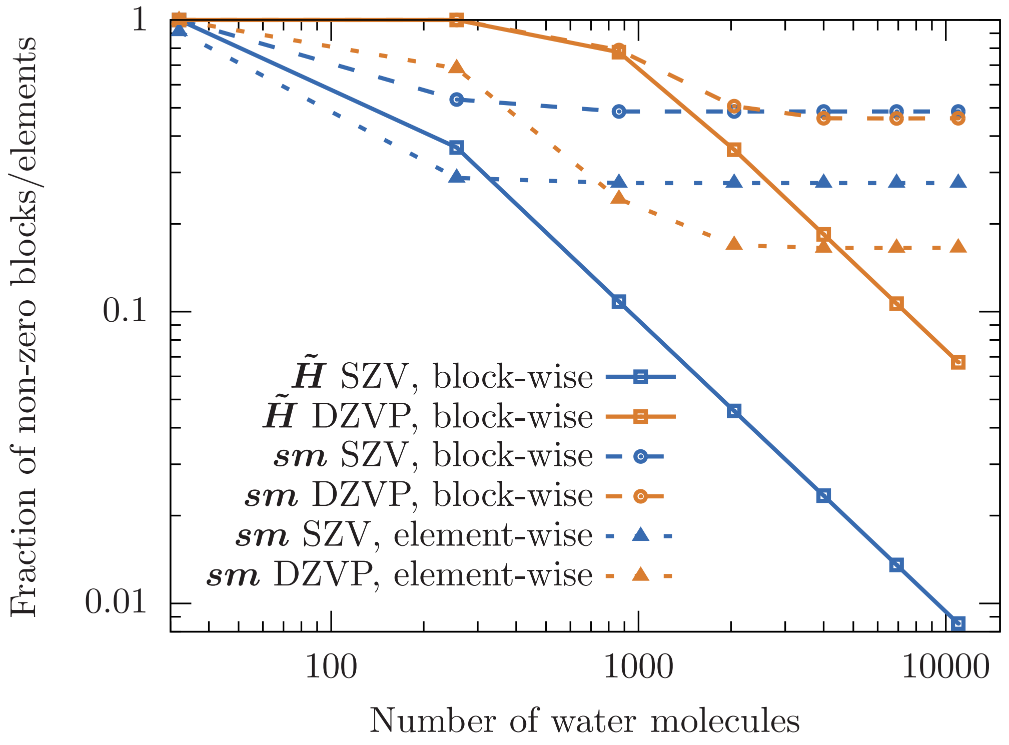

Figure 6.8: Block-wise (dashed lines with circles) and element-wise (dashed lines with triangles) sparsity of submatrices **sm** compared to block-wise sparsity the orthogonalized Kohn-Sham matrix $\tilde{\boldsymbol{H}}$ (solid lines with circles) for a cube of liquid water with periodic boundary conditions described in an SZV (blue lines) and a DZVP (orange lines) basis set and a cutoff value of 10^{-5} for the matrix elements.

DZVP. The reason is that blocks are considered nonzero as soon as a single element in that block is above the cutoff value. For the overall larger blocks using DZVP this causes more sparse blocks being included in the submatrices.

In cases where a minimal basis such as SZV-MOLOPT-SR-GTH is not sufficient to achieve the desired chemical accuracy, the Submatrix Method can also be combined with larger basis sets. However, due to the increased sparsity of the generated submatrices, sparse linear algebra algorithms may be favorable for processing the submatrices. Alternatively, the Submatrix Method could be applied in an element-wise manner to the generated submatrices to obtain smaller and more dense matrices, even when using larger basis sets.

6.4 Summary of Findings

In this chapter, we have demonstrated that the Submatrix Method is a promising new method to realize Linear Scaling DFT computations. Using it in CP2K to compute the density matrix from the Kohn Sham matrix via the matrix sign function, the Submatrix Method outperforms traditional, iterative approaches if the matrices are sufficiently sparse. At the same time, it exhibits better weak-scaling properties due to the fact that operations on different submatrices are embarrassingly parallel and do not require any communication.

Although the Submatrix Method is an inherently approximate method, the quality of the results is comparable to that of the originally implemented method which directly applies the Newton-Schulz iteration to the distributed sparse Hamiltonian. We have also seen that the mechanism in CP2K to trade off output quality against computational effort via an $\varepsilon_{\text{filter}}$ cutoff value applies well to the Submatrix Method where the cutoff value directly affects the size of the submatrices. By now, our implementation of the Submatrix Method including its use for computing the density matrix has been included in CP2K and can be easily accessed through corresponding configuration flags in the input file.

Chapter 7

Hardware Acceleration of Submatrix Operations

The Submatrix Method transforms the original operation on a large sparse input matrix into many independent operations on smaller more dense matrices. In the case of large HPC applications it can also be used to transform an operation on a distributed input matrix into operations on locally stored matrices. In the case of the CP2K implementation presented in Chapter 6, the original input matrix is a distributed block sparse DBCSR matrix. Using the Submatrix Method, the operation can be performed on locally stored dense matrices whose size is limited by the short range of the used basis functions. All this makes the processing of the generated submatrices an ideal target for hardware offloading.

In this chapter we describe different ways to offload the iterative computation of the sign function for the generated submatrices to GPUs and FPGAs. For GPU offloading, we will combine the Submatrix Method with the use of low-precision arithmetic as evaluated in Chapter 4 to be able to fully exploit the performance of the GPUs. For FPGA offloading, we will consider two approaches: First, only matrix multiplications are offloaded to the FPGA, allowing use of the most performant matrix multiplication kernel. In the second approach, all computations required for the sign iteration are offloaded to the FPGA, allowing to design a more general FPGA offloading library for iterative schemes.

Parts of this chapter have been published at the *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)* in 2020 and are part of the corresponding proceedings published by ACM and IEEE [A2]. The GPU accelerator described in Section 7.1 and the first FPGA acceleration approach described in Section 7.2 have mainly been developed and evaluated by the second author of our publication, Robert Schade. The second FPGA acceleration approach described in Section 7.3 has been developed and evaluated by the author of this work.

7.1 GPU Acceleration Using Tensor Cores

As presented in Section 2.1.2, GPUs provide high floating-point performance and are therefore well suited for compute-limited matrix operations. With their specialized tensor cores, current NVIDIA GPUs provide support for low-prevision arithmetic, for example using half-precision or mixed-precision arithmetic. Here, we demonstrate the use of consumer-grade NVIDIA RTX 2080 Ti cards to accelerate the matrix sign computation of submatrices generated as part of CP2K's modified LSDFT method.

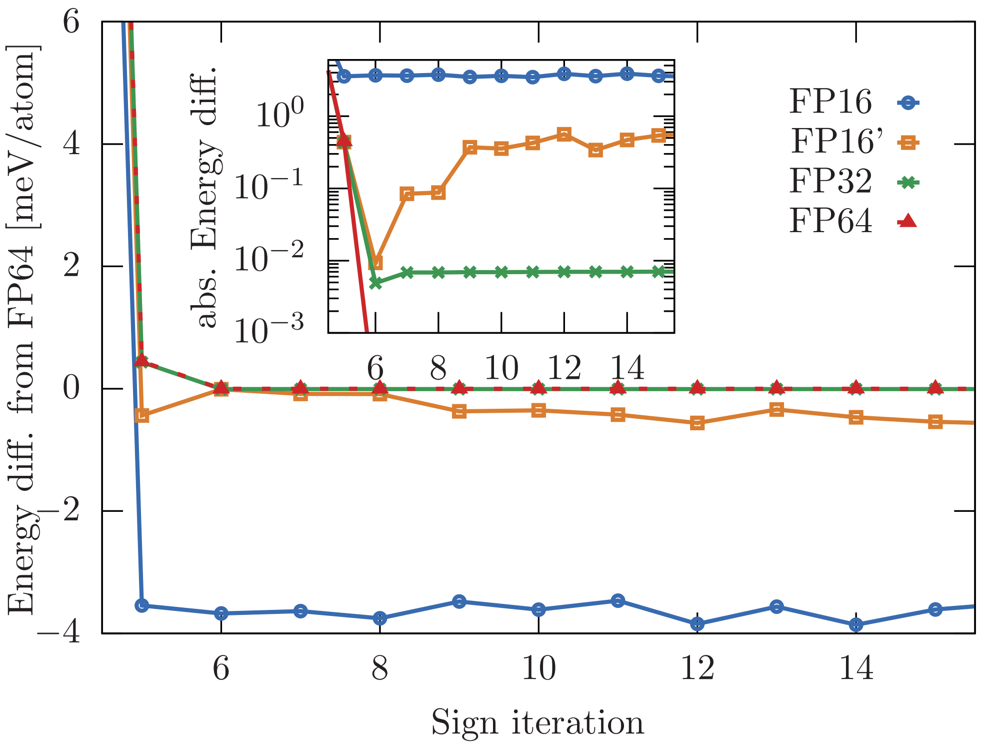

Figure 7.1: Convergence of the third-order sign iteration in different precisions on a NVIDIA RTX 2080 Ti for the combined submatrix of 32 water molecules in a system of 4000 water molecules described in an SZV basis. The large graph shows the energy difference for the 32 water molecules from the converged FP64 result. The inset shows the absolute energy difference on a logarithmic scale.

To compute the sign function, we implement the sign iteration based on the third-order Padé-approximation from Equation 2.7

$$\begin{aligned} \boldsymbol{X}_0 = \boldsymbol{A}, \quad \boldsymbol{X}_{k+1} &= \frac{1}{8}\boldsymbol{X}_k(15\boldsymbol{I} - 10\boldsymbol{X}_k^2 + 3\boldsymbol{X}_k^4) \\ \operatorname{sign}(\boldsymbol{A}) &= \lim_{k\to\infty} \boldsymbol{X}_k. \end{aligned} \tag{2.7 revisited}$$

using the NVIDIA cuBLAS [14] library. We differentiate between four different variants of this implementation: double-precision (FP64), single-precision (FP32), half-precision (FP16) and mixed precision (FP16′) where multiplication is performed using half-precision arithmetic and accumulation is performed using single-precision. All steps of the iterative scheme have been implemented on the GPU using CUDA 10.2. The input matrix is only transferred once to the GPU and the result matrix is only transferred once back to the host.

We evaluate the convergence behavior and the performance of our GPU accelerator on a GPU node of the OCuLUS cluster introduced in Section 2.1.1 using a submatrix that has been generated for multiple block columns such that it covers 32 water molecules. The underlying system is the same as used in Section 6.3 with NREP set to 5, such that the entire system contains 4000 water molecules. SZV-MOLOPT-SR-GTH was used as basis set. The implementation of the GPU accelerator, as well as scripts and inputs that have been used in this evaluation, are publicly available [B3].

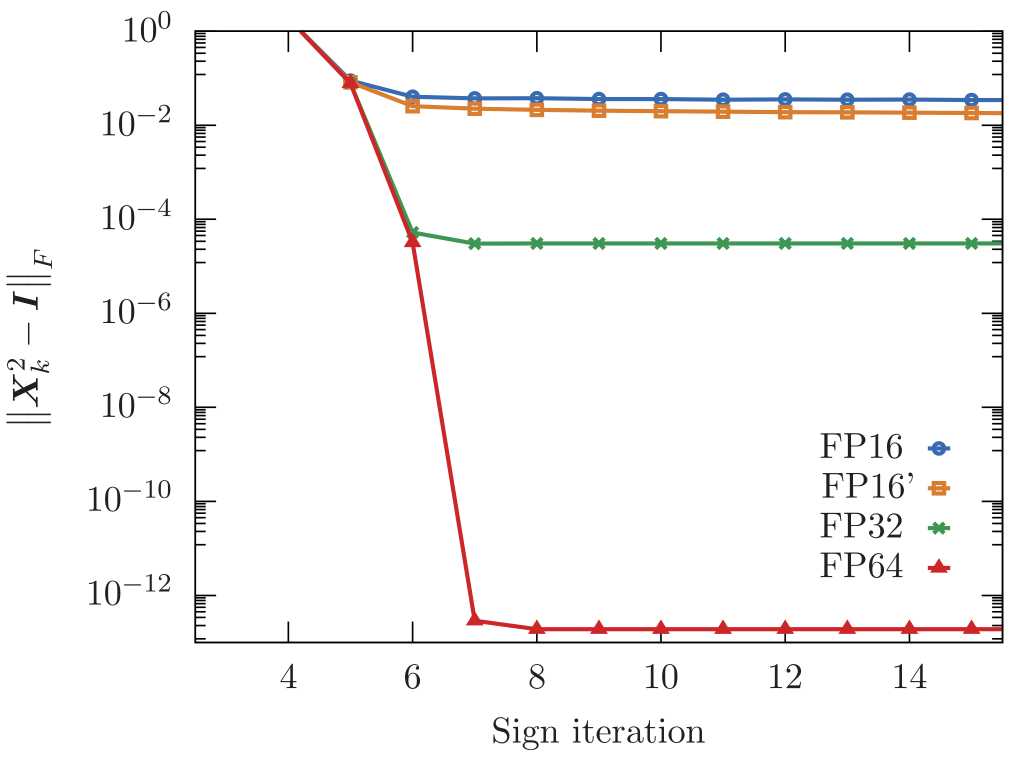

Figure 7.2: Deviation from the involutority condition $X_k^2 = I$ in every step of the third-order sign iteration in different precisions on a NVIDIA RTX 2080 Ti for the same situation as in Figure 7.1.

The convergence is shown in Figure 7.1. All curves depict the difference of the energy computed using a certain precision and a certain number of iterations and a result precomputed using double-precision arithmetic. For all of the different variants, we observe convergence within 6–8 iterations. This matches our findings presented in Section 4.2 in that the initial rate of convergence does not significantly decrease when using low-precision computations. Overall the resulting energies are within 5 meV/atom of the double-precision result. In particular, the mixed-precision approach provides results close to single-precision, if the iterations are stopped before errors start to accumulate. Using the computed energy itself as convergence criterion, i.e., running the iteration until the energy is minimized, shows not to be suitable, as for example in the FP16′ case the energy tends to drift downwards with additional iterations. Instead, using the involutority as convergence criterion shows to be reliable also in the low-precision cases, as shown in Figure 7.2. Again this confirms our initial findings in Section 4.2 in that error accumulation can become an issue when too many iterations are performed using low-precision arithmetic.

Performance results are shown in Table 7.1. We distinguish between three values: the *peak performance* that is theoretically achievable on a NVIDIA RTX 2080 Ti for the given precision, the measured performance for *matrix multiplications* which was measured for submatrices of size 3972 that have been constructed as described above, and the overall *sign algorithm* performance which includes all auxiliary operations as well as data transfers from and to the host and necessary type conversions. With a theoretical peak performance of 108 TFLOP/s at 250 W, the RTX 2080 Ti provides a peak energy efficiency of 432 GFLOP/J when using half-precision arithmetic. For matrix multiplications, we achieve around half of the peak performance

Table 7.1: Peak performance of a NVIDIA RTX 2080 Ti, practical matrix-matrix multiply performance for the given matrix size 3972 and overall performance of the sign algorithm including type conversions, data transfer and convergence tests for the different precision modes.

Precision	Peak Performance	Matrix Multiplications	Sign Algorithm
FP16	108 TFLOP/s	56.4 TFLOP/s	35.2 TFLOP/s
FP16′	56 TFLOP/s	38.2 TFLOP/s	27.8 TFLOP/s
FP32	13 TFLOP/s	12.2 TFLOP/s	10.4 TFLOP/s
FP64	0.5 TFLOP/s	0.5 TFLOP/s	0.5 TFLOP/s

at 56.4 TFLOP/s (225 GFLOP/J). For the entire algorithm, we achieve around one third of the peak performance at around 35 TFLOP/s (140 GFLOP/J). Looking at both performance and accuracy of the results, we see that mixed-precision provides a good compromise at around 28 TFLOP/s (112 GFLOP/J) and an accuracy much closer to single-precision arithmetic compared to half-precision.

7.2 FPGA Acceleration of Matrix Multiplications

Next to GPUs, FPGAs pose a promising architecture for offloading of computationally intensive hotspots. Since FPGAs are a relatively new emerging technology in HPC environments, the range and capabilities of available libraries is much smaller. Furthermore, since the compute logic is directly mapped to resources on the FPGA and the selection and amount of resources greatly varies between different FPGA vendors and models, it is difficult to design universal libraries. For the sign iteration, the computationally demanding operation is the matrix-matrix multiplication. As a first step, we therefore focus on offloading this multiplication to the FPGA and implement the rest of the iterative scheme on the host CPU. The implementation of this FPGA acceleration approach, as well as scripts and inputs that have been used in this evaluation, are publicly available [B3].

Our target hardware is the BittWare 520N board which is contained in the FPGA nodes of the Noctua 1 cluster introduced in Section 2.1.1. The board contains an Intel Stratix 10 GX 2800 FPGA and 32 GiB of DDR4 memory. It is connected to the host via a PCI-E 3.0 x8 interface. Since this FPGA contains DSP units that are optimized towards single-precision floating-point operations, we stick to single-precision arithmetic. Instead of developing a new matrix multiplication kernel, we use the example provided as part of the Intel FPGA Software Development Kit (SDK) for OpenCL 19.2 [89]. We were able to place two instances of this kernel on the FPGA with both kernels performing a matrix multiplication $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ of size $2048 \times m \times 2048$. The design reaches a frequency of 424 MHz and uses 71% of the available DSP resources. The peak performance is measured at 3.4 TFLOP/s. At 110 W measured power consumption, this corresponds to 31 GFLOP/J. However, this does only take the power consumption of the FPGA into account.

The designs uses a significant portion of the available resources and with its 424 MHz it achieves a relatively high clock frequency. However, it has some drawbacks that limit its energy efficiency and its practicality. The first disadvantage is that the kernels need to be fed with matrix blocks by the host in a very specific fashion. In fact, this requires constant block transformations on the host, utilizing one of the two host CPUs. Taking into account the 129 W measured power consumption of this

CPU as well, the theoretically achievable energy efficiency drops down to around 14 GFLOP/J. A second disadvantage is that the matrices are constantly transferred between the host and the FPGA. Moving the entire iteration scheme to the FPGA is impossible due to the need for block transformations that require complex index operations and need to be performed on the host.

For the submatrices of dimension 3972, we observe a practical performance of the matrix multiplications of 2.7 TFLOP/s, corresponding to nearly 80% of the theoretical peak performance. However, considering the entire sign iteration, this drops down to 1.75 TFLOP/s, caused by the overhead of block transformations and data transfers. With that, energy efficiency drops to about 16 GFLOP/J when only considering the FPGA or 7.3 GFLOP/J when considering both FPGA and host CPU. While this is significantly lower than the 41 GFLOP/J measured for the GPU implementation in single-precision, we see that an FPGA would be able to achieve similar energy efficiency, if overheads could be significantly reduced and the entire iteration scheme could be offloaded to the FPGA.

7.3 FPGA Accelerator for Iteration Schemes

The first FPGA acceleration approach provides good performance but has two major drawbacks. Firstly, only the matrix multiplication is implemented on the FPGA. Therefore all other parts of the iteration need to be performed on the host CPU, requiring regular transfers of updated matrices from and to the FPGA. Secondly, the used matrix multiplication kernel that is provided by Intel requires compute intensive preparation of matrix blocks. This pre- and post-processing of matrix blocks utilizes multiple CPU cores, increasing the overall workload of the system. We now present a more general design for an FPGA accelerator that allows not only offloading matrix multiplication but entire iteration schemes to the FPGA. This avoids the need to copy matrices or matrix blocks from and to the FPGA during the iteration and relieves the host CPU from all compute intensive tasks.

The accelerator is designed with the following requirements in mind:

1. Support for the Newton-Schulz sign iteration and related operations such as the computation of inverse p-th roots of dense (sub-) matrices.
2. Input matrices shall only be copied once to the FPGA before starting the iteration scheme. Output matrices shall only be copied once back to the host after all computations have finished.
3. The accelerator should be able to utilize an Intel Stratix 10 FPGA and large parts of the available DSP resources.
4. The accelerator should be easily accessible from any HPC application.

7.3.1 Required Kernels

One major design constraint is that there should be no necessity to copy matrices back and forth between host and FPGA more than once. That implies that the FPGA accelerator needs to be able to perform all numeric operations involved in the iterative computational scheme. The Newton-Schulz iteration for computing the matrix sign function is given by

$$\begin{aligned} \boldsymbol{X}_0 = \boldsymbol{A}, \quad \boldsymbol{X}_{k+1} &= \frac{1}{2}\boldsymbol{X}_k(3\boldsymbol{I} - \boldsymbol{X}_k^2) \\ \operatorname{sign}(\boldsymbol{A}) &= \lim_{k\to\infty} \boldsymbol{X}_k. \end{aligned} \tag{2.6 revisited}$$

Hence, the following operations need to be performed on the FPGA:

1. Matrix multiplication,
2. scaling of matrices by constants,
3. addition of matrices.

Looking at the iterative step required for computation of inverse p-th roots, given by

$$\boldsymbol{X}_{k+1} = \frac{1}{p}\left((p+1)\boldsymbol{X}_k - \boldsymbol{X}_k^{p+1}\boldsymbol{A}\right), \tag{2.1 revisited}$$

it shows that this iteration scheme can be implemented as well using the exact same kernels. In fact, likely many more iterative schemes can be implemented using these fundamental operations.

In addition to the iteration itself, also the convergence criterion must be computed on the FPGA to be able to terminate the algorithm as soon as a suitable result has been found. For that, a suitable matrix norm must be implemented on the FPGA, leading to a fourth required kernel in our design:

4. Computation of a matrix's Frobenius norm.

In the following, all of these kernels including their optimization are briefly described. All kernels operate on single-precision floating-point numbers.

Matrix Multiplication

The matrix multiplication is the only compute limited kernel. It needs to be designed in a way to utilize large parts of the FPGA, in particular the DSP resources, make effective use of them in most clock cycles and at the same time allow high clock rates. This makes the development of well-performing matrix multiplication for large FPGAs such as the Intel Stratix 10 challenging. Instead of designing a custom kernel, we use the design proposed and published by Gorlani et al. [90]. It does not require a complex blocking scheme like the kernel used in Section 7.2 and its OpenCL code is well readable and can be customized if necessary.

The kernel is based on a pipelined implementation of Cannon's algorithm [69] that performs multiplications of two 8×8 matrix blocks with an Initiation Interval (II) of one, i.e., it performs 1024 FLOP in each clock cycle. An outer blocking of configurable size (dimension: 360–1024) serves caching purposes and increases the arithmetic intensity of the entire kernel such that it becomes mainly compute bound.

The kernel is available as open source [91] and its performance as well as area requirements can be controlled by (1) changing the size of matrix blocks that are cached and (2) replicating the entire compute kernel to process multiple blocks in parallel. Caching larger blocks allows increasing the arithmetic intensity and therefore reduces the time spent for data transfers from and to DRAM.

To fulfill the needs for implementation of the iterative schemes, the kernel's functionality needs to be extended. These extensions are described in the following.

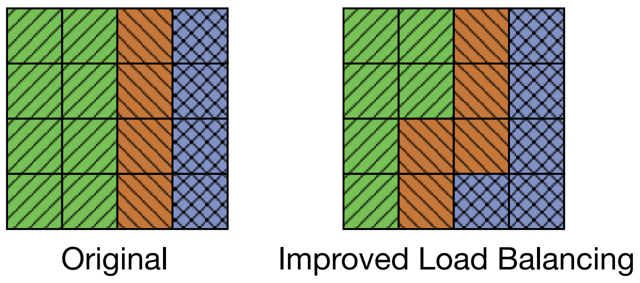

Figure 7.3: Original and improved load balancing between matrix multiplication kernels. Instead of entire block columns, individual blocks are assigned.

Performing a Full SGEMM Operation The kernel originally was designed to perform the operation

$$\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}, \tag{7.1}$$

where A, B and C are square matrices. To allow a more flexible use of the kernel and avoid any unnecessary scaling and addition operations, we extend the functionality to a full SGEMM operation, i.e.,

$$\boldsymbol{C} = \alpha \boldsymbol{A} \cdot \boldsymbol{B} + \beta \boldsymbol{C}, \tag{7.2}$$

where α and β are scalar values. This extension requires two modifications to the kernel code:

1. The output blocks for matrix $\boldsymbol{C}$ need to be properly initialized by the block currently held in memory and all elements need to be multiplied by β. This requires introducing another data loading loop that is executed when starting to process a new block.

2. The computation itself needs to be extended to scale the multiplication result by α and to accumulate on top of the previously loaded value.

Although these modifications introduce additional logic, they do not significantly impact the synthesis result in terms of achieved clock rates and consumed resources. While loading more data from DRAM does take additional run time, this time is negligible compared to the overall run time of the kernel, in particular if block sizes are rather large, e.g., 512×512 elements.

Optimized Load Balancing The original kernel supports an arbitrary number of kernel replicas. Blocks are then distributed column-wise to kernels. As an example, let us assume three kernels that jointly process a matrix composed of $4 \times 4 = 16$ blocks. This example is shown in Figure 7.3. The first kernel gets assigned two block columns, i.e., eight blocks. The other kernels get assigned one block column, i.e., four blocks each. Consequently, the first kernel would take double the execution time of the other kernels.

We modify this logic in that the load balancing does not operate on block columns but on single blocks. In the above example, this leads to the first kernel processing six blocks and all other kernels processing five blocks. In this specific scenario, this optimization provides a speedup of $8/6 = 1.3\times$. However, the impact greatly varies for different number of kernels and different matrix sizes.

Listing 7.1: OpenCL copy kernel.

```
#pragma unroll 16
for (uint i = 0; i < len; i++) {
        B[i] = A[i];
}
```

Listing 7.2: OpenCL scaling and addition kernel.

```
#pragma unroll 16
for (uint i = 0; i < len; i++) {
        C[i] = A[i]*alpha + B[i]*beta;
}
```

Allowing Use of Seperate Memory Channels The matrix multiplications kernel performs best when matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ are placed in different memory banks and automatic interleaving of memory channels is turned off. To be able to fulfill this constraint at all times, we need to be able to copy over matrices from one memory bank to another. This functionality is provided by an additional *copy* helper kernel that is implemented in OpenCL by a simple loop as shown in Listing 7.1. The unrolling factor of 16 gives a total data width of 512 bits that is read and written in a single clock cycle. With that, the kernel is able to fully exploit the available memory bandwidth.

Scaling and Addition

Although the extension of the matrix kernel allows performing scaling and addition during a matrix multiplication, it is still reasonable to provide a kernel that allows to perform these operations independent of matrix multiplications. Both of these operations can be combined in a single kernel that performs the operation

$$\vec{c} = \alpha\vec{a} + \beta\vec{b}, \tag{7.3}$$

where α and β are scalars and $\vec{a}, \vec{b}$ and $\vec{c}$ are vectors. As matrices are stored consecutively in memory, they can be passed in and out as vectors as well. Similar to the copy kernel, this operation is purely memory bound such that the only optimization goal is to fully utilize the available memory bandwidth. Again this is achieved using an unrolling factor of 16 as shown in Listing 7.2.

Frobenius Norm Computation

The decision when to terminate the iteration is usually based on some distance norm. In case of the iterative sign computation, in addition to $||\boldsymbol{A} - \boldsymbol{I}||_F$ for some matrix $\boldsymbol{A}$, codes often additionally require $||\boldsymbol{A}||_F$ to normalize the convergence criterion wrt. size and contents of the matrix. The Frobenius norm kernel is designed to compute both of these values, such that they can be transferred back to the host where the decision for or against continuation of the iteration is taken.

The Frobenius norm from Definition 2.12 is given by

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}. \tag{7.4}$$

Computing the square root is an expensive operation to perform on an FPGA which takes up valuable resources. At the same time, computing the square root of two scalar values can be quickly done on the host CPU. Therefore the kernel does not compute the Frobenius norm but $\|\boldsymbol{A}\|_F^2$ and $\|\boldsymbol{A} - \boldsymbol{I}\|_F^2$, leaving proper interpretation of these values to the host code.

Computing the Frobenius norm is a purely memory bandwidth limited operation. In general, the implementation approach therefore closely resembles the formerly described kernels. However, computation of $\|\boldsymbol{A} - \boldsymbol{I}\|_F^2$ requires to identify elements on the matrix diagonal and handling them differently by subtracting one before squaring. Performing this distinction between elements based on their position and performing the corresponding arithmetic operation turned out to be challenging for the OpenCL High-Level Synthesis tools. Another operation that could not automatically be handled by the tools is the accumulation of products into a single scalar value. Synthesis tools tend to introduce high Initiation Intervals into loops that contain this operation. However, to work close to the available memory bandwidth, an II of one is required.

To cope with these issues, shift registers have been introduced at multiple places to introduce delay cycles and shorten critical paths within single clock cycles. At the same time, the loop needs to be unrolled by a factor of 16 to fully utilize the available memory bandwidth. Therefore all shift registers need to be replicated by 16 as well. Overall, the kernel now consists of the following elements, replicated 16 times:

1. Upcoming diagonal index computation: If one of the 16 parallel loop iterations has dealt with a diagonal element, compute the index of the next upcoming diagonal element.

2. Diagonal index identification: If the currently processed matrix element corresponds to the awaited diagonal element, emit a corresponding signal to trigger computation of the next upcoming diagonal element. This signal is passed through a shift register to introduce delay cycles.

3. Element computation: Square current matrix element. If the currently processed matrix element corresponds to the awaited diagonal element, subtract one before squaring when computing $\|\boldsymbol{A} - \boldsymbol{I}\|_F^2$.

4. Accumulation: Pop leftmost value from accumulation shift register, add computed element and push result from the right.

This concept is visualized in Figure 7.4 which shows the part of the kernel which computes $\|\boldsymbol{A} - \boldsymbol{I}\|_F^2$. $\|\boldsymbol{A}\|_F^2$ can easily be computed by bypassing the logic that distinguishes between diagonal and non-diagonal elements and accumulating into a second set of shift registers.

After the entire matrix has been processed, the results are spread over 16 shift registers each, where each shift register contains twelve elements, i.e., 192 values need to be accumulated to get the final result. This operation is performed in a loop with an II larger than one. Allowing a non-optimal II has no significant performance impact due to the low number of iterations but allows to save resources on the FPGA.

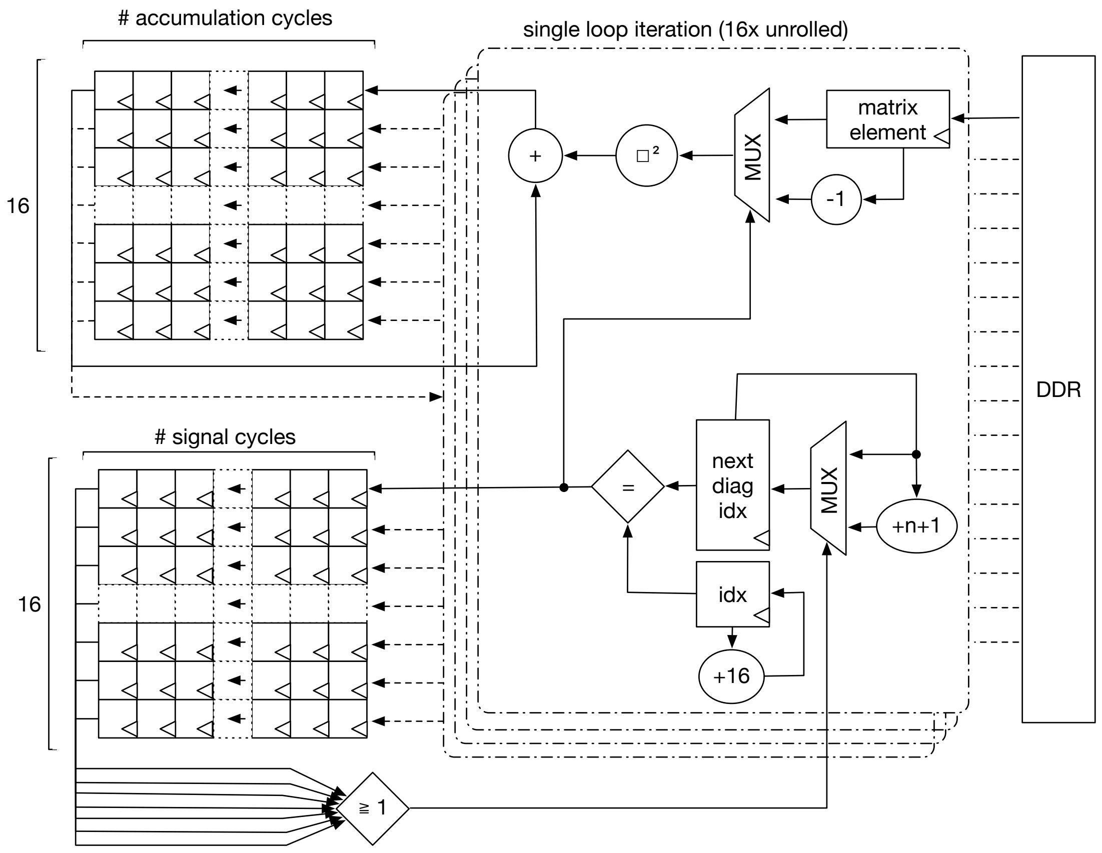

Figure 7.4: Visualization of the computation of $\|\boldsymbol{A} - \boldsymbol{I}\|_F^2$ within the Frobenius kernel. After completion, the upper set of shift registers holds fractional results which need to be accumulated before returning to the host. In practice, 12 accumulation cycles and two signal cycles are used.

7.3.2 Host Code Design

While all involved matrix operations are implemented on the FPGA such that matrices do not need to be transferred repeatedly between the FPGA and the host, the main control flow remains on the host. While in general also the control flow can be implemented as part of the FPGA accelerator, such that it performs the entire iteration scheme autonomously, this requires resynthesis of the entire design when changes to the algorithm are required or new algorithms should be implemented. In contrast, when the main control flow remains on the host, new methods can easily be implemented based on a fixed FPGA design that contains the kernels discussed above.

Next to this extensibility, the host code needs to provide an easy interface to scientific software that wants to take advantage of the FPGA accelerator. It is therefore wrapped in a library that provides the following functionality:

1. Allocation of available FPGA devices. Using environment variables, the set of available devices can be restricted. This allows integration into workload managers that manage the available resources, including accelerator devices.

2. Programming of the FPGA using presynthesized bitstreams. This step is automatically performed if an accelerator routine is called that requires a different bitstream than what is currently programmed on the FPGA.

3. Running an accelerator routine, e.g., to compute the matrix sign function. All OpenCL-specific code is transparently handled, such that the application does not need to be aware of the concrete accelerator platform or the kernels implemented on this accelerator.

Each accelerator routine needs to abstract away data transfers from and to the FPGA as well as the kernel calls required to implement the offloaded operation. For the sign function computation, the routine performs the following steps:

1. Data preparation:

   (a) To guarantee convergence, the matrix elements need to be scaled. This operation, i.e., determining a suitable scaling factor and scaling each element of the matrix, are performed in software.

   (b) The size of the input matrix needs to be a multiple of the block size supported by the matrix multiplication kernel. A suitable size is determined and the padded area is filled with ones on the diagonal and zeros on all off-diagonal positions.

2. Setup of OpenCL buffers required as input and output memory for the different kernels.

3. Transfer of input data to the corresponding buffer.

4. Iteration loop:

   (a) Perform kernel calls to compute the result of a single iteration step.

   (b) Perform kernel calls to compute $\|\boldsymbol{A}\|_F^2$ and $\|\boldsymbol{A} - \boldsymbol{I}\|_F^2$ for $\boldsymbol{A} = \boldsymbol{X}_k^2$.

   (c) Transfer these two scalar values back to the host.

   (d) Post-process returned values to compensate for the padding of the input matrix and compute the square root to obtain the Frobenius norm.

Table 7.2: Resources available on the Intel Stratix 10 GX 2800 FPGA and resources consumed by the static portion of the BittWare BSP. Shown are the numbers of *Adaptive Logic Modules (ALMs)*, *Flip Flops (FFs)*, on-chip memory blocks *(RAMs)* and *DSPs*.

	ALMs	FFs	RAMs	DSPs
Static Portion	227,620 (24%)	910,480 (24%)	2,627 (22%)	1,047 (18%)
Available	705,500 (76%)	2,822,000 (76%)	9,094 (78%)	4,713 (82%)
Total	933,120 (100%)	3,732,480 (100%)	11,721 (100%)	5,760 (100%)

(e) Repeat step 4 until convergence.

5. Remove the initially added padding from the result matrix.

The accelerator routine to compute inverse p-th roots can be programmed analogously.

Availability

The described FPGA accelerator, including kernels, host code and an abstraction library are available under MIT license [B1].

7.3.3 Evaluation

The used matrix multiplication kernel generally provides lower floating-point peak performance than the kernel used in Section 7.2 and the performance depends on the block size used by the kernel, the achieved clock frequency and the size of the input matrix.

There are elaborate performance measurements for the used matrix multiplication kernel in the original publication [90]. On the BittWare 520N board with an Intel Stratix 10 FPGA, a design consisting of five kernels and a block size of 512×512 elements performs best with around 1.3 TFLOP/s. Placing six kernels is only possible when reducing the block size to 360×360 which increases the impact of data transfers onto the overall run time. At the same time, the achievable clock frequency is reduced from 294 MHz to 224 MHz, reducing the overall performance to 1.12 TFLOP/s. Therefore, we use the configuration of five kernels and 512×512 block sizes for the FPGA accelerator design.

Combining five instances of the modified matrix multiplication kernel with the additional kernels described above, a final accelerator design has been synthesized for the BittWare 520N board using the Intel FPGA SDK for OpenCL in version 20.2.0 and Quartus in version 19.4.0. In total, 20 designs have been synthesized using different seeds in order to obtain a design that comes close to the best achievable clock frequency. With `seed=7`, a design running at 298.24 MHz could be synthesized, which is even slightly faster than what has been presented before.

Resource Consumption

A significant portion of the FPGA's resources is already taken up by the Board Support Package (BSP), or the *static portion* of the design, as shown in Table 7.2. The resource consumption of the kernels is given as percentage of the remaining usable resources in Table 7.3.

Table 7.3: Resources consumed by the different kernels. Percentages are given wrt. the available resources. Shown are the numbers of *Adaptive Logic Modules (ALMs)*, *Flip Flops (FFs)*, on-chip memory blocks *(RAMs)* and *DSPs*.

Kernel	ALMs	FFs	RAMs	DSPs
copy	2,510 (0.36%)	6,205 (0.02%)	34 (0.37%)	0 (0.00%)
frobenius	28,028 (3.98%)	62,513 (2.22%)	16 (0.18%)	52 (1.10%)
sgemm_0	66,517 (9.43%)	212,633 (7.53%)	1,665 (18.3%)	669 (14.2%)
sgemm_1	66,659 (9.45%)	212,255 (7.52%)	1,665 (18.3%)	669 (14.2%)
sgemm_2	66,877 (9.48%)	213,030 (7.55%)	1,665 (18.3%)	669 (14.2%)
sgemm_3	66,824 (9.47%)	231,630 (8.21%)	1,665 (18.3%)	669 (14.2%)
sgemm_4	66,699 (9.45%)	214,223 (7.59%)	1,665 (18.3%)	669 (14.2%)
scale_add	3,071 (0.44%)	7,912 (0.28%)	48 (0.53%)	32 (0.68%)
Total	395,664 (56.1%)	1,248,313 (44.3%)	8,497 (93.4%)	3,429 (72.8%)

The combined resource consumption of the SGEMM kernels is close to the values reported by the original kernel author [90]. Only few additional DSP blocks (in total: 3345 instead of 3223) are required to implement the additional functionality provided by the proposed kernel extensions. It can also be seen that five kernels with block size 512×512 take up 91.5% of the available on-chip memory blocks, i.e., instantiating additional kernels would require lowering the block size.

The additional kernels require only little additional resources. Most noteworthy is the consumption of adaptive logic modules and flip-flops by the Frobenius kernel which is caused by the extensive use of shift registers and the accumulation logic that is replicated for each unrolled loop instance.

Performance Evaluation

The performance of all kernels is evaluated for matrix sizes that are multiples of the block size 512×512. Since other matrix sizes are padded up to the next multiple of this block size, performance for those matrix sizes can be determined from these measurements.

Due to the deterministic behavior of all kernels and the fixed clock frequency, the content of the matrices does not have any influence on the performance. For performance evaluation, therefore identity matrices of the corresponding size are generated on the host and transferred to the device before running the kernels. All kernels are executed 50 times to obtain reliable measurements. Data transfer times are not included in this performance evaluation since all kernels are designed to be executed as part of an iterative scheme without the need of large data transfers between kernel invocations.

Floating-Point Performance The only compute limited kernel is the matrix multiplication kernel. Due to multiple kernels operating on blocks of a fixed size of 512, optimal performance will only be achieved if the input matrix sizes are a multiple of this block size. Otherwise, matrices will be padded and the kernels will spend additional time on the appended rows and columns. In addition, there will be load

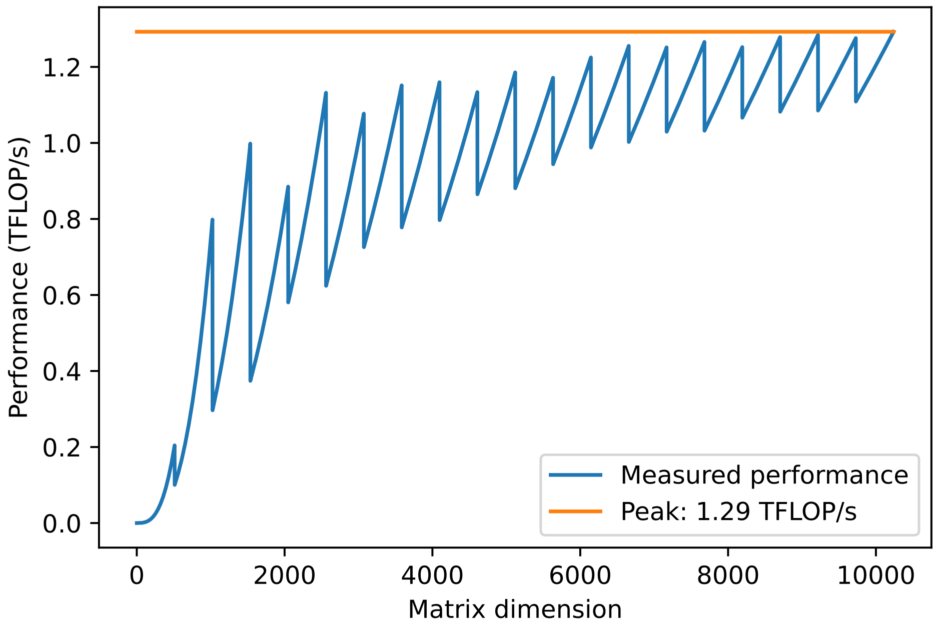

Figure 7.5: Performance of the matrix multiplication kernel for different input matrix sizes. Required padding and non-optimal load balancing show in the visible sawtooth pattern and the fluctuating height of its peaks.

imbalance between the kernels unless

$$\left\lceil \frac{S}{s_B} \right\rceil^2 \mod n_K = 0, \tag{7.5}$$

where S is the matrix dimension, s_B is the block size (in this evaluation: 512) and n_K is the number of kernel replicas (in this evaluation: 5).

Figure 7.5 shows the measured performance in TFLOP/s for different matrix sizes. The sawtooth pattern is caused by the required padding. As expected, the performance impact of padding gets reduced for larger matrices, as shown by the shrinking height of the saw teeth.

The impact of load imbalance can be seen in the non-monotonic increase of the performance peaks. For example, at matrix size 1536 (9 blocks), only one of the five kernels runs idle before the execution finishes, while at matrix size 2048 (16 blocks), four of five kernels run idle while a single kernel computes the last block. Therefore, the overall performance drops when increasing the matrix size from 3×3 to 4×4 blocks. This effect becomes less pronounced for larger matrices where the performance for matrices that do not require padding generally comes close to the peak performance. Overall, the highest measured performance is 1.29 TFLOP/s for a matrix size of 10240×10240.

Throughput The three remaining kernels, i.e., computation of the Frobenius norm, scaling and adding of vectors and copying of vectors, are purely memory-bandwidth limited. The optimization goal for these kernels is therefore to utilize the performance of the memory channels that are available.

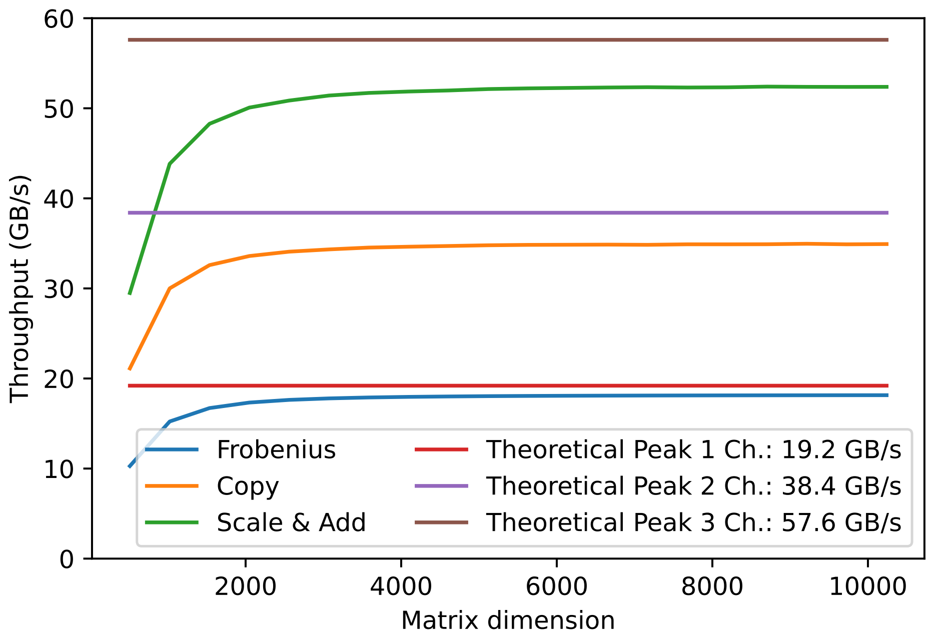

Figure 7.6: Performance in terms of throughput of the remaining kernels for different input matrix sizes. For comparison, the maximum theoretical throughput of one, two and three memory channels for the used DDR4-2400 memory are shown.

To allow the matrix multiplication kernel to perform best, it is necessary to disable automatic memory interleaving and to access different matrices over different memory channels. The throughput of the three evaluated kernels therefore depends on how many memory channels they can utilize. For the Frobenius kernel, this is just a single memory channel since only a single matrix is read. The copy kernel can utilize two memory channels, one reading in a vector and the other one writing out the copy. The scaling and adding kernel can utilize three channels as it takes two input vectors and writes out one result vector.

Figure 7.6 shows the measured throughput for all three kernels. In all cases, throughput already comes close to the measured peak values for matrix dimensions below 2000. The measurements confirm the performance expectations based on the number of memory channels. In particular, they show that the optimization of the Frobenius kernel allows it to fully utilize its input memory channel while still consuming only little FPGA resources.

Entire Sign Iteration In addition to the performance of the single kernels, we also evaluate the performance of running the entire sign iteration on the FPGA. We do so using the same scenario as used in Sections 7.1 and 7.2, i.e., for a submatrix of dimension 3972. We measure a practical performance of the entire sign iteration of 1.01 TFLOP/s. This is close to the performance of the matrix multiplication kernel, showing the low overhead caused by the additionally required operations. During execution, the FPGA consumes up to 110 W, leading to an energy efficiency of 9.18 GFLOP/J. Including data transfers from and to the host, we measure an overall performance of 0.96 TFLOP/s (8.73 GFLOP/J). Since data is only transferred once before and after execution of the sign iteration, the corresponding overhead varies with the number of iterations performed on the FPGA. The presented numbers have been measured by performing six iterations which required 1.62 s for the computations on the FPGA and 79 ms for data transfers.

While the raw performance numbers of this accelerator design are behind what was achieved in Section 7.2 by offloading only matrix multiplications, its practical performance is at around 78% of the peak performance of the used matrix multiplication kernel and the overhead of data transfers becomes negligible. In addition, it relieves the host CPU from all compute and memory intensive tasks and therefore allows the CPU to be used for other tasks.

7.4 Summary of Findings

In this chapter, three approaches to offload the computations on submatrices to hardware accelerators have been presented. For the iterative methods discussed in this work, i.e., computation of the matrix sign function and matrix p-th roots, matrix multiplications are the only compute limited operation. As shown in Section 7.1, modern GPUs such as NVIDIA GPUs with tensor cores provide high performance and energy efficiency, in particular when using low-precision arithmetic. The Submatrix Method now allows to utilize this performance for matrix operations on large sparse matrices and with that for the LSDFT method discussed in this work.

For offloading computations to FPGAs, we have shown two approaches. Offloading only the matrix multiplications allows to use the most performant matrix multiplication kernel available for a given FPGA. However, as shown in Section 7.2, this approach comes with high overheads caused by regular data transfers between host and FPGA. We therefore also showed how a dedicated iteration accelerator design can be constructed from a set of kernels. While providing less performance, this approach comes with less data transfer overhead and allows to easily implement further iterative schemes based on the same FPGA design.

The performance of the FPGA accelerator is limited mainly by the floating-point performance of the used matrix multiplication kernel. Recently, a new matrix multiplication design for Intel FPGAs has been proposed which achieves 3 TFLOP/s on the used Stratix 10 FPGA [92]. Incorporating this new kernel into the accelerator design proposed in this chapter has the potential to bring performance and energy efficiency of the FPGA accelerator closer to what has been achieved with GPUs.

Chapter 8

Conclusion

In this chapter, we conclude this thesis by providing a brief summary of its contents and results as well as an outlook on possible future work.

8.1 Summary

In this thesis, we described and evaluated two approximation approaches in the context of Linear Scaling DFT and used them to accelerate ab-initio molecular dynamics simulations. Specifically, we looked at density matrix based DFT as it is implemented in the quantum chemistry code CP2K.

First, we examined iterative algorithms for two central kernels of the considered method, namely the computation of inverse p-th roots and the matrix sign function of matrices. We evaluated them with respect to their resilience against errors caused by low precision in the used floating-point arithmetic or for the storage of intermediate results. For both kernels, the considered algorithms still converge when using half-precision arithmetic and even when using much lower precision for the storage of intermediate results. Interestingly, the initial rate of convergence is barely influenced by the reduced precision. Additionally, for the computation of inverse p-th roots, a refinement of the computed solution using higher precision arithmetic is possible.

As a second approximation technique, we introduced the Submatrix Method, which transforms the application of a unary matrix operation on a large sparse input matrix into a set of independent applications of this matrix function on much smaller dense matrices. Although results computed using this method are only approximations of the solutions, we showed that they are sufficiently accurate for application in Linear Scaling DFT as well as entirely different areas, such as preconditioning. The fact that the operations on different submatrices are entirely independent of each other allows massive parallelization of these computations. At the same time, the much smaller dense submatrices are very well suited to be processed by hardware accelerators such as GPUs and FPGAs.

We demonstrated these properties and the practicality of the described approximation techniques by providing an open source implementation within CP2K. Simulations of benchmark systems containing liquid water showed absolutely linear scaling with the system size and overall good strong and weak scaling properties. Compared to the conventional method implemented in CP2K, the submatrix based approach showed favorable weak scaling properties and overall better performance if accuracy requirements on computed energies do not exceed a certain limit. Finally, we demonstrated how the submatrix operations can be offloaded to GPUs and FPGAs and how low-precision arithmetic units of modern GPUs can be exploited in order to increase the achievable floating-point performance and energy efficiency. On a single NVIDIA RTX 2080 Ti we achieved a practical performance of 35 TFLOP/s and an energy efficiency of 140 GFLOP/J using half-precision arithmetic, which corresponds to a third of its theoretical peak performance and energy efficiency.

8.2 Outlook and Future Work

The work presented in this thesis provides manifold starting points for future work, both in research and in practical development of software and hardware libraries.

- In this work, only the main computational hotspot of AIMD is considered as a target for the Submatrix Method. Consequently, submatrices regularly need to be constructed from distributed sparse matrices and results have to be regularly collected back into the original, distributed sparse format. A worthwhile goal is to move more of the involved computations into the realm of submatrix computations, such that data transfers can be reduced. In fact, follow-up research has already started where submatrix elements are computed directly instead of being extracted from a large distributed matrix, allowing to move the computations involved in generating the input matrices to accelerators as well. Using a cluster of 1536 NVIDIA A100 GPUs, up to 100 million atoms could be simulated using this approach achieving 324 PFLOP/s using mixed precision [B2].
- We have demonstrated that the Submatrix Method can be used for different matrix functions and in different application areas. It is reasonable to assume that there are more applications than electronic structure methods that can profit from utilizing this method, warranting further research. In this context it can be worthwhile to implement the method as a generic, parameterizable library to ease the adoption by other researchers.
- For the use of GPUs in HPC, there are different comprehensive libraries available and many scientific codes already incorporate support for GPUs. So far, the selection of corresponding libraries for FPGAs is rather limited. Due to the high entry barrier of FPGA development, the presence of such libraries seems vital for a broad adoption of this accelerator type in HPC. The approach for an FPGA accelerator library for iterative schemes that is discussed in this work has great potential as a starting point for a more comprehensive linear algebra library.

Overall, the methods presented in this thesis and their integration into Linear Scaling DFT prepare ab-initio molecular dynamics simulations for the ever-growing degree of parallelism that needs to be exploited in today's HPC clusters and the growing importance of hardware accelerators in these systems.

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

AC Approximate Computing. 2, 5, 11, 12, 31, 58

AIMD Ab-Initio Molecular Dynamics. 1, 2, 19, 20, 22, 28, 29, 30, 92

ALM Adaptive Logic Module. 86, 87

ASIC Application-Specific Integrated Circuit. 8, 11, 13

BLAS Basic Linear Algebra Subprograms. 27, 65

BRAM Block RAM. 9

BSP Board Support Package. 86, 95

CG Conjugate Gradient. 48, 49, 95

CMOS Complementary Metal-Oxide-Semiconductor. 13

COO COOrdinate format. 60, 61

CPU Central Processing Unit. 5, 6, 7, 8, 52, 53, 54, 55, 56, 58, 65, 69, 70, 78, 79, 83, 90, 94

CSC Compressed Sparse Column. 43, 53

CSR Compressed Sparse Row. 26, 44

DBCSR Distributed Block Compressed Sparse Row. 26, 27, 60, 62, 63, 64, 66, 69, 75, 93

DDR Double Data Rate. 6, 78, 89, 101

DFT Density Functional Theory. 1, 2, 19, 20, 22, 23, 26, 28, 29, 30, 32, 36, 41, 44, 46, 52, 59, 69, 73, 91, 92, 102

DRAM Dynamic RAM. 9, 80, 81

DSP Digital Signal Processing. 9, 10, 13, 78, 79, 80, 86, 87

DZVP Double-Zeta Valence Polarized. 28, 29, 30, 71, 72, 73, 95

FF Flip Flop. 8, 86, 87

FLOP FLoating-point OPerations. 5, 6, 7, 8, 64, 77, 78, 79, 80, 86, 88, 89, 90, 91, 92

FPGA Field Programmable Gate Array. 1, 2, 5, 6, 8, 9, 10, 11, 13, 40, 75, 78, 79, 80, 83, 85, 86, 89, 90, 91, 92, 93, 95

GDDR Graphics DDR. 6

GPU Graphics Processing Unit. 1, 2, 5, 6, 7, 8, 13, 27, 40, 75, 76, 78, 79, 90, 91, 92

GTH Goedecker-Teter-Hutter. 28, 66, 71, 73, 76

HLS High-Level Synthesis. 10, 83

HPC High-Performance Computing. 1, 2, 5, 6, 8, 9, 10, 11, 12, 19, 75, 78, 79, 92

II Initiation Interval. 80, 83

KS-DFT Kohn-Sham DFT. 23, 24

LAPACK Linear Algebra PACKage. 53

LSDFT Linear Scaling DFT. 1, 2, 26, 28, 29, 31, 32, 36, 38, 41, 44, 47, 49, 50, 52, 56, 58, 59, 69, 73, 75, 90, 91, 92

LUT Lookup Table. 8

MAC Multiply-ACcumulate. 7, 9

MD Molecular Dynamics. 19, 20, 22, 26, 29, 30

MPI Message Passing Interface. 6, 27, 52, 53, 54, 57, 59, 60, 66

PCIe PCI Express. 6, 9

PLL Phase-Locked Loop. 9

RAM Random Access Memory. 86, 87, 101, 102

RDMA Remote Direct Memory Access. 6

SCF Self Consistent Field. 24, 29, 30, 59

SDK Software Development Kit. 78, 86

SIMD Single Instruction Multiple Data. 6, 7, 8

SR Short Range. 28, 66, 71, 73, 76

SRAM Static RAM. 9

SZV Single-Zeta Valence. 28, 29, 62, 66, 71, 72, 73, 76, 95

TZVP Triple-Zeta Valence Polarized. 28

Author's Peer-Reviewed Publications

[A1] Thomas Kühne et al. "CP2K: An electronic structure and molecular dynamics software package - Quickstep: Efficient and accurate electronic structure calculations". In: *The Journal of Chemical Physics* 152.19 (2020). DOI: 10.1063/5.0007045.

[A2] Michael Lass et al. "A Submatrix-Based Method for Approximate Matrix Function Evaluation in the Quantum Chemistry Code CP2K". In: *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2020. DOI: 10.1109/SC41405.2020.00084.

[A3] Varadarajan Rengaraj et al. "Accurate Sampling with Noisy Forces from Approximate Computing". In: *Computation* 8.2 (2020). DOI: 10.3390/computation8020039.

[A4] Dorothee Richters et al. "A General Algorithm to Calculate the Inverse Principal p-th Root of Symmetric Positive Definite Matrices". In: *Communications in Computational Physics* 25.2 (2019). DOI: 10.4208/cicp.OA-2018-0053.

[A5] Michael Lass, Thomas Kühne, and Christian Plessl. "Using Approximate Computing for the Calculation of Inverse Matrix p-th Roots". In: *Embedded Systems Letters* 10.2 (2018). DOI: 10.1109/LES.2017.2760923.

[A6] Michael Lass et al. "A Massively Parallel Algorithm for the Approximate Calculation of Inverse p-th Roots of Large Sparse Matrices". In: *Proc. Platform for Advanced Scientific Computing Conference (PASC)*. ACM, 2018. DOI: 10.1145/3218176.3218231.

[A7] Heinrich Riebler et al. "Efficient Branch and Bound on FPGAs Using Work Stealing and Instance-Specific Designs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10.3 (2017). DOI: 10.1145/3053687.

[A8] Michael Lass, Dominik Leibenger, and Christoph Sorge. "Confidentiality and Authenticity for Distributed Version Control Systems - A Mercurial Extension". In: *Proc. 41st Conference on Local Computer Networks (LCN)*. Received Best Paper Award. IEEE, 2016. DOI: 10.1109/lcn.2016.11.

Author's Preprints, Presentations, Software and Artifacts

[B1] Michael Lass. *FPGA Accelerator Design for the Matrix Sign Function*. 2021. DOI: 10.5281/ZENODO.5731593.

[B2] Robert Schade et al. "Enabling Electronic Structure-Based Ab-Initio Molecular Dynamics Simulations with Hundreds of Millions of Atoms". In: *Preprint* (2021). arXiv: 2104.08245 [physics.comp-ph].

[B3] Michael Lass et al. *Evaluation of the submatrix method within CP2K*. 2020. DOI: 10.5281/ZENODO.3878760.

[B4] Michael Lass et al. *Modifications of the CP2K code for evaluation of the submatrix method*. 2020. DOI: 10.5281/ZENODO.3878669.

[B5] Michael Lass. *Prototypic Implementation of the Submatrix Method*. 2018. DOI: 10.5281/ZENODO.5731531.

[B6] Michael Lass, Thomas Kühne, and Christian Plessl. "Using Approximate Computing in Scientific Codes". In: *Workshop on Approximate Computing (AC)*. 2016.

Bibliography

[1] Erich Strohmaier et al. *TOP500 List - November 2021*. 2021. URL: `https://www.top500.org/lists/top500/2021/11/` (visited on 11/22/2021).

[2] Erich Strohmaier et al. *GREEN500 List - November 2021*. 2021. URL: `https://www.top500.org/lists/green500/2021/11/` (visited on 11/22/2021).

[3] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. University of Tennessee, 1994.

[4] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998).

[5] Paderborn Center for Parallel Computing. *OCuLUS (Owl CLUSter)*. 2021. URL: `https://pc2.uni-paderborn.de/hpc-services/available-systems/oculus/` (visited on 11/13/2021).

[6] NVIDIA Corporation. *NVIDIA Turing GPU Architecture Whitepaper*. 2018. URL: `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf` (visited on 11/28/2021).

[7] Paderborn Center for Parallel Computing. *Noctua 1*. 2021. URL: `https://pc2.uni-paderborn.de/hpc-services/available-systems/noctua1` (visited on 11/13/2021).

[8] BittWare. *520N PCIe Card with Intel Stratix 10 GX FPGA*. 2021. URL: `https://www.bittware.com/fpga/520n/` (visited on 11/13/2021).

[9] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st. Morgan Kaufmann Publishers Inc., 2012.

[10] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science & Engineering* 12.3 (2010).

[11] David A. Patterson John Hennessy. *Computer Architecture*. Elsevier LTD, 2019.

[12] Ronny Krashinsky et al. *NVIDIA Ampere Architecture In-Depth*. 2020. URL: `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/` (visited on 09/28/2021).

[13] OpenACC-Standard.org. *The OpenACC Application Programming Interface*. Version 3.1. 2020.

[14] NVIDIA Corporation. *cuBLAS: Dense Linear Algebra on GPUs*. URL: `https://developer.nvidia.com/cublas` (visited on 11/28/2021).

[15] Intel Corporation. *Intel Stratix 10 Variable Precision DSP Blocks User Guide (UG-S10-DSP)*. 2021.

[16] Xilinx, Inc. *UltraScale Architecture DSP Slice User Guide (UG579)*. v1.11. 2021.

[17] Tobias Kenter, Jens Forstner, and Christian Plessl. "Flexible FPGA design for FDTD using OpenCL". In: *Proc. 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017.

[18] Tobias Kenter et al. "OpenCL-Based FPGA Design to Accelerate the Nodal Discontinuous Galerkin Method for Unstructured Meshes". In: *Proc IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018.

[19] Tobias Kenter et al. "Algorithm-hardware co-design of a discontinuous Galerkin shallow-water model for a dataflow architecture on FPGA". In: *Proc. Patform for Advanced Scientific Computing Conference*. ACM, 2021.

[20] Maxeler Technologies Ltd. *Programming MPC Systems*. 2013. URL: `https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf` (visited on 11/28/2021).

[21] P. Klavík et al. "Changing Computing Paradigms Towards Power Efficiency". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical & Engineering Sciences* 372.2018 (2014).

[22] Sparsh Mittal. "A Survey of Techniques for Approximate Computing". In: *ACM Computing Surveys* 48.4 (2016).

[23] V. K. Chippa et al. "Analysis and Characterization of Inherent Application Resilience for Approximate Computing". In: *Proc. 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. ACM, 2013.

[24] Zhixi Yang et al. "Approximate XOR/XNOR-based adders for inexact computing". In: *Proc. 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*. IEEE, 2013.

[25] P Kulkarni, P Gupta, and M Ercegovac. "Trading Accuracy for Power with an Underdesigned Multiplier Architecture". In: *Proc. 24th Annual Conference on VLSI Design*. IEEE, 2011.

[26] Kan Shi, David Boland, and George A. Constantinides. "Accuracy-Performance Tradeoffs on an FPGA through Overclocking". In: *Proc. 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2013.

[27] David May and Walter Stechele. "Voltage over-scaling in sequential circuits for approximate computing". In: *Proc. International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, 2016.

[28] Bernd Ulmann. *Analog Computing*. de Gruyter Oldenbourg, 2013.

[29] Weikang Qian et al. "An Architecture for Fault-Tolerant Computation with Stochastic Logic". In: *IEEE Transactions on Computers* 60.1 (2011).

[30] Hadi Esmaeilzadeh et al. "Neural Acceleration for General-Purpose Approximate Programs". In: *Proc. 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012.

[31] T. K. Callaway and E. E. Swartzlander. "Power-delay characteristics of CMOS multipliers". In: *Proc. 13th IEEE Sympsoium on Computer Arithmetic*. 1997.

[32] Nicholas J. Higham. *Functions of Matrices*. CAMBRIDGE, 2008.

[33] Nicholas J Higham and Lijing Lin. “A Schur–Padé algorithm for fractional powers of a matrix”. In: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011).

[34] Nicholas J. Higham and Lijing Lin. “An Improved Schur–Padé Algorithm for Fractional Powers of a Matrix and Their Fréchet Derivatives”. In: *SIAM Journal on Matrix Analysis and Applications* 34.3 (2013).

[35] D. A. Bini, N. J. Higham, and B. Meini. “Algorithms for the matrix pth root”. In: *Numerical Algorithms* 39.4 (2005).

[36] Günther Schulz. “Iterative Berechung der reziproken Matrix”. In: *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 13.1 (1933).

[37] Charles Kenney and Alan J. Laub. “Rational Iterative Methods for the Matrix Sign Function”. In: *SIAM Journal on Matrix Analysis and Applications* 12.2 (1991).

[38] Stefan Heinen et al. “Machine learning the computational cost of quantum chemistry”. In: *Machine Learning: Science and Technology* 1.2 (2020).

[39] Ananth Y. Grama et al. “N-Body Computational Methods”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, 2011.

[40] Daan Frenkel. *Understanding molecular simulation : from algorithms to applications*. Academic Press, 2002.

[41] Wolfgang Eckhardt et al. “591 TFLOPS Multi-trillion Particles Simulation on SuperMUC”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[42] Nikola Tchipev et al. “TweTriS: Twenty trillion-atom simulation”. In: *The International Journal of High Performance Computing Applications* 33.5 (2019).

[43] Haoxuan Ding et al. “Perturbational Imaging of Molecules with the Scanning Tunneling Microscope”. In: *The Journal of Physical Chemistry C* 124.47 (2020).

[44] William C. Swope et al. “A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters”. In: *The Journal of Chemical Physics* 76.1 (1982).

[45] Michael Gastegger and Philipp Marquetand. “Molecular Dynamics with Neural Network Potentials”. In: *Machine Learning Meets Quantum Physics*. Springer International Publishing, 2020.

[46] Matti Hellström and Jörg Behler. “High-Dimensional Neural Network Potentials for Atomistic Simulations”. In: *Machine Learning Meets Quantum Physics*. Springer International Publishing, 2020.

[47] Huziel E. Sauceda et al. “Construction of Machine Learned Force Fields with Quantum Chemical Accuracy: Applications and Chemical Insights”. In: *Machine Learning Meets Quantum Physics*. Springer International Publishing, 2020.

[48] Mark Tuckerman et al. “Car-Parrinello Method”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, 2011.

[49] W. Kohn and L. J. Sham. “Self-Consistent Equations Including Exchange and Correlation Effects”. In: *Physical Review* 140 (4A 1965).

[50] Elliott H. Lieb. "Density functionals for coulomb systems". In: *International Journal of Quantum Chemistry* 24.3 (1983).

[51] Mel Levy and John P. Perdew. "The Constrained Search Formulation of Density Functional Theory". In: *Density Functional Methods In Physics*. Springer US, 1985.

[52] R. O. Jones. "Density functional theory: Its origins, rise to prominence, and future". In: *Reviews of Modern Physics* 87 (3 2015).

[53] Janice A. Steckel David Sholl. *Density Functional Theory*. Wiley-Interscience, 2009.

[54] NobelPrize.org. *The Nobel Prize in Chemistry 1998*. 1998. URL: `https://www.nobelprize.org/prizes/chemistry/1998/summary/` (visited on 09/18/2021).

[55] W. Kohn. "Nobel Lecture: Electronic structure of matter – wave functions and density functionals". In: *Reviews of Modern Physics* 71 (5 1999).

[56] M. Born and R. Oppenheimer. "Zur Quantentheorie der Molekeln". In: *Annalen der Physik* 389.20 (1927).

[57] J. C. Slater. "Atomic Shielding Constants". In: *Physical Review* 36.1 (1930).

[58] S. F. Boys. "Electronic wave functions - I. A general method of calculation for the stationary states of any molecular system". In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 200.1063 (1950).

[59] S. Mohr et al. "Daubechies wavelets for linear scaling density functional theory". In: *Journal of Chemical Physics* 140.20 (2014).

[60] Per-Olov Löwdin. "On the Non-Orthogonality Problem Connected with the Use of Atomic Wave Functions in the Theory of Molecules and Crystals". In: *Journal of Chemical Physics* 18.3 (1950).

[61] Joost VandeVondele, Urban Borštnik, and Jürg Hutter. "Linear Scaling Self-Consistent Field Calculations with Millions of Atoms in the Condensed Phase". In: *Journal of Chemical Theory and Computation* 8.10 (2012).

[62] E. Prodan and W. Kohn. "Nearsightedness of electronic matter". In: *Proc. of the National academy of Sciences of the United States of America* 102.33 (2005).

[63] W. Kohn. "Density Functional and Density Matrix Method Scaling Linearly with the Number of Atoms". In: *Physical Review Letters* 76 (17 1996).

[64] Stefan Goedecker. "Linear scaling electronic structure methods". In: *Reviews of Modern Physics* 71 (4 1999).

[65] Jörg Kussmann, Matthias Beer, and Christian Ochsenfeld. "Linear-scaling self-consistent field methods for large molecules". In: *WIREs Computational Molecular Science* 3.6 (2013).

[66] Jürg Hutter et al. "CP2K: Atomistic simulations of condensed matter systems". In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 4.1 (2013).

[67] Joost VandeVondele et al. "Quickstep: Fast and accurate density functional calculations using a mixed Gaussian and plane waves approach". In: *Computer Physics Communications* 167.2 (2005).

[68] Urban Borštnik et al. "Sparse matrix multiplication: The distributed block-compressed sparse row library". In: *Parallel Computing* 40.5-6 (2014).

[69] Lynn Elliot Cannon. "A Cellular Computer to Implement the Kalman Filter Algorithm". PhD thesis. Montana State University, 1969.

[70] Alexander Heinecke et al. "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation". In: *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2016.

[71] Ole Schütt et al. "GPU-Accelerated Sparse Matrix-Matrix Multiplication for Linear Scaling Density Functional Theory". In: *Electronic Structure Calculations on Graphics Processing Units*. Wiley, 2016.

[72] Wikimedia Commons. *The first five atomic orbitals of neon*. 2020. URL: `https://commons.wikimedia.org/wiki/File:Neon_orbitals.png` (visited on 11/27/2021).

[73] W. Pauli. "Über den Zusammenhang des Abschlusses der Elektronengruppen im Atom mit der Komplexstruktur der Spektren". In: *Zeitschrift für Physik* 31.1 (1925).

[74] Jan C. A. Boeyens. *Chemistry from First Principles*. Springer Netherlands, 2008.

[75] Joost VandeVondele and Jürg Hutter. "Gaussian basis sets for accurate calculations on molecular systems in gas and condensed phases". In: *The Journal of Chemical Physics* 127.11 (2007).

[76] R Kubo. "The fluctuation-dissipation theorem". In: *Reports on Progress in Physics* 29.1 (1966).

[77] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001. URL: `http://www.scipy.org/` (visited on 11/28/2021).

[78] S. Behnel et al. "Cython: The Best of Both Worlds". In: *Computing in Science & Engineering* 13.2 (2011).

[79] A. Schöll, C. Braun, and H. J. Wunderlich. "Applying efficient fault tolerance to enable the preconditioned conjugate gradient solver on approximate computing hardware". In: *Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. 2016.

[80] Stephan Mohr. "Fast and accurate electronic structure methods : large systems and applications to boron-carbon heterofullerenes". PhD thesis. University of Basel, 2013.

[81] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011).

[82] T.H. Cormen et al. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009.

[83] François Le Gall. "Powers of Tensors and Fast Matrix Multiplication". In: *Proc. 39th Int. Symp. on Symbolic and Algebraic Computation*. ISSAC '14. ACM, 2014.

[84] Intel Corporation. *MPI Library*. 2021. URL: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html` (visited on 11/28/2021).

[85] E. Angerson et al. "LAPACK: A portable linear algebra library for high-performance computers". In: *Proc. ACM/IEEE Conference on Supercomputing*. 1990.

[86] Intel Corporation. *Math Kernel Library*. 2021. URL: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html` (visited on 11/28/2021).

[87] Laura Susan Blackford et al. "ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance". In: *Proc. 1996 ACM/IEEE Conf. on Supercomputing*. IEEE Computer Society, 1996.

[88] Mariana Kolberg, Gerd Bohlender, and Dalcidio Claudio. "Improving the Performance of a Verified Linear System Solver Using Optimized Libraries and Parallel Computation". In: *Proc. 8th Int. Conf. High Performance Computing for Computational Science (VECPAR)*. Springer Berlin Heidelberg, 2008.

[89] Intel Corporation. *Intel FPGA SDK for OpenCL*. 2021. URL: `https://www.intel.de/content/www/de/de/software/programmable/sdk-for-opencl/overview.html` (visited on 11/28/2021).

[90] Paolo Gorlani, Tobias Kenter, and Christian Plessl. "OpenCL Implementation of Cannon's Matrix Multiplication Algorithm on Intel Stratix 10 FPGAs". In: *Proc. Int. Conf. on Field-Programmable Technology (ICFPT)*. IEEE, 2019.

[91] Paolo Gorlani. *Github repository: cannon-fpga*. 2019. URL: `https://github.com/pc2/cannon-fpga` (visited on 10/10/2021).

[92] Paolo Gorlani and Christian Plessl. "High Level Synthesis Implementation of a Three-dimensional Systolic Array Architecture for Matrix Multiplications on Intel Stratix 10 FPGAs". In: *Preprint* (2021). arXiv: `2110.11521 [cs.AR]`.