Technische Universität München
Fakultät für Informatik
Informatik 5 – Lehrstuhl für Wissenschaftliches Rechnen (Prof. Bungartz)

Scalable scientific computing applications for GPU-accelerated heterogeneous systems

Christoph Karl Riesinger

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 16.05.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.07.2017 angenommen.

Abstract

In the last decade, graphics processing units (GPUs) became a major factor to increase performance in the area of high performance computing. This is reflected by numerous examples of the fastest supercomputers in the world which are accelerated by GPUs. GPUs are one representative of many-core chips, the major technology development to boost hardware performance in previous years. Many-core chips were one factor besides others such as progress in modeling, algorithmics, and data structures to allow scientific computing advance to its current state-of-the-art.

To exploit the whole computational power of GPUs, numerous challenges have to be tackled in the area of parallel programming: Latest developments in handling the core characteristics of GPUs such as two additional levels of parallelism, the memory system, and offloading shift the focus on the usage of multiple GPUs in parallel and/or combining them with the performance of CPUs (heterogeneous computing). As a consequence, hybrid parallel programming (e.g. MPI, OpenMP) concepts are required and load balancing as well as communication hiding become even more relevant to achieve good scalability.

In this work, we present approaches to benefit from GPUs for three different applications, each covering different algorithmic characteristics: First, a pipelined approach is used to determine the eigenvalues of a symmetric matrix not only enabling very high FLOPS rates but also allowing for the handling of even large systems on one single GPU. Second, the solution of random ordinary differential equations (RODEs) offers multiple levels of parallelism which is predestined for systems with multiple GPUs leading to the first implementation of an RODE solver to deal with problems of reasonable size. Finally, it is shown that a pioneering hybrid implementation of the lattice Boltzmann method making use of all available compute resources in the system where the CPU can process regions of arbitrary volume can attain good scalability.

Acknowledgements

Even if there is only one author name written on the front page of this thesis, there are numerous other persons who contributed to this document in one way or the other. So I am taking the chance to express my acknowledgements and thanks to these people.

First of all, I would like to mention my PhD supervisors Prof. Hans-Joachim Bungartz and Prof. Takayuki Aoki. They offered me the opportunity to start and successfully work on my PhD in very comfortable, pleasant, productive and competent environments, especially during my research stay abroad in Tokyo where the first actual results could be achieved.

Before achieving actual results, much groundwork has to be finished, not always done by myself. Here, I want to thank Tobias Neckel and Florian Rupp for their preliminary studies in the field of random ordinary differential equations forming the theoretical basis of part III of this document. Special acknowledgements go to Tobias who did not just contribute in a technical way as the advisor of my thesis but also became a close friend. The same gratefulness belongs to Martin Schreiber and Arash Bakhtiari for their practical effort in the area of the lattice Boltzmann method continued by me in part IV. Martin, I am not sure if you reduced or actually extended the time to finish my PhD, anyways, you definitely made this time much more valuable.

In addition, I would like to thank these people who gave me access to the computing resources essential for my research work. Robert Speck paved the way to utilize the infrastructure in Jülich, Frank Jenko enabled the access to the Max Planck resources in Garching, Maria Grazia Giuffreda arranged the usage of several clusters in Lugano, and, again, Prof. Takayuki Aoki is mentioned for his support in Tokyo. If there was any onside technical problem, Roland Wittmann was the guy you can count on.

Furthermore, I have to thank Alfredo Parra Hinojosa, again, Tobias Neckel, Philipp Neumann, and Benjamin Uekermann who significantly enhanced the quality and the language of this thesis by proofreading and reviewing. Alfredo also has to be mentioned for his pragmatic and effective approach to execute the duties of a coordinator of the Computational Science and Engineering (CSE) program and, hence, was the perfect colleague a CSE secretary can rely on. In the same way, I thank my former CSE and office colleague Marion Weinzierl.

Besides colleagues and people who supported me in a technical way (and sometimes became very good friends), there is also my family I always could count on. I want to deeply thank my girl-friend Barbara and my parents Elisabeth and Karl for doing the “cover my back” stuff and for always giving me the feeling, no the certainty that nothing can really go wrong.

Hence, every time a “we”, “our”, or “us” is mentioned on the following pages, all these just listed people are also meant in one way or the other.

Contents

Part I.

Introduction

1. Opening

This thesis is opened with an introductory part before coming to the actual applications in later parts. The first chapter of the introductory part provides the motivation, our contribution, and an outline to get started. Section 1.1 serves the motivation where current challenges of scientific computing in the context of high performance computing (HPC) are presented. It is dealing with the question “why” the work given in this document is relevant. The approaches developed to master such challenges are sketched in section 1.2. There, it is distinguished between techniques already well-established in the community and insights published for the very first time in this thesis. Finally, section 1.3 describes the structure of the rest of this document. It shows the recurring themes through this work but also lists aspects that are not within the scope of this dissertation.

1.1. Motivation

Simulation, i.e. the virtual imitation of a real-world process or system, has been established as third pillar besides theory and experiment to gain understanding and knowledge [194]. Scientific computing is the multidisciplinary field which applies simulation to specific applications, especially in science and engineering. Application engineers are looking for answers to ambitious questions in engineering and insight into phenomena of yet unfeasible quality, so simulations become more and more complex: The usage of multiple physics in one simulation (e.g. to tackle fluid-structure interaction problems where fluid and structure dynamics have to be considered), the treatment of multiple scales (e.g. when investigating membranes which requires the coupling of phenomenon on an atomic and a mesoscopic scale), the analysis of multi-dimensional problems (e.g. in finance), or the increase of resolution or numerical accuracy of the simulation are only a few examples of this growing complexity. Advances in modeling such as surrogate models or model order reduction, in numerics such as sophisticated discretization schemes and preconditioning, and in algorithmics such as procedures having a reduced computational complexity make it possible to tackle this increased complexity, as well as a higher investment of computational resources for the overall simulation. Accordingly, also other steps of the simulation pipeline such as meshing for pre-processing and visualization or qualitative statements for post-processing become more computationally expensive. Latest trends such as the consideration of stochastic processes, or—more generally—uncertainty quantification [222], and big data analysis further increase the need for computational performance.

For more than a decade, an increase of computational performance has mainly been achieved by hardware parallelization: Vector units integrated in processor cores, numerous arithmetic logic units (ALUs) per core, multiple cores per chip, more nodes per compute cluster, and so on. Hence, writing parallel software being aware of hardware parallelism plays a vital role in scientific computing and parallelization is one of the major techniques in HPC. As a result, scalability became one of the most important metrics to measure the quality of parallel software: In the same way as the number of processing elements is increased, the performance should also increase. This can either be a reduction of compute time and/or an expansion of problem size. Right now in the pre-exaFLOPS era, it seems very likely that the first machines which will be able to compute 10^{18} floating-point operations per second (FLOPS) will be equipped with accelerators. Upcoming supercomputers such as Aurora[1], Sierra[2], and Summit[3] strongly suggest this course. Accelerators such as graphics processing units (GPUs), Intel Xeon Phi[4], or more exotic examples such as the PEZY accelerator[5] or Google's Tensor Processing Unit augment existing systems using classical central processing units (CPUs) with additional computing devices. Such computing devices offer in general disproportionately high computational performance with drawbacks in programmability and applicability. Systems augmented in such a way are called *heterogeneous systems* because they host at least two different types of computing devices, i.e. CPUs and accelerators.

Heterogeneous systems make it easier to hit the sweet spot between computational performance, acquisition price, space, and energy consumption.

On the one hand, having a look on the current Top500[6] list shows that three of the top ten supercomputers are already heterogeneous systems. Thus, accelerators are no new technology but already well-established. On the other hand, efficient usage of accelerators poses different challenges for the developer: In general, accelerators introduce extra levels of hardware parallelism. Mapping existing and new algorithms and methods to heterogeneous systems and exploiting all available resources is non-trivial, especially when running complex and sophisticated simulations. Another challenge is to achieve scalability on large heterogeneous systems. This is hard due to the offloading character of accelerators. Non-uniform memory access, different cycles per byte ratios of the devices, and continuous utilization of the processing elements are further obstacles which have to be conquered. Concerning the software and programming side, new implementations may become necessary because the different levels of parallelism require to address different memory architectures (shared or distributed memory) suggesting different approaches of threading and tasking.

[1] https://www.alcf.anl.gov/articles/introducing-aurora
[2] https://asc.llnl.gov/coral-info
[3] https://www.olcf.ornl.gov/summit/
[4] Recently, with the Knights Corner generation, the Intel Xeon Phi became an independent device running its own operation system. Hence, it does not rely on any CPU anymore making it not just an accelerator but also a stand-alone device.
[5] https://www.pezy.co.jp/en/index.html
[6] https://www.top500.org/list/2016/11/

1.2. Contribution

Combining well-known aspects of accelerators from many years of experience and new, non-trivial approaches to the challenge for their efficient usage is the one of the major contributions of this thesis. We study GPUs as accelerators and how to utilize them in the best way. Therefore, we cover numerous aspects from simple single-GPU scenarios up to massively parallel scenarios using large heterogeneous systems. To this end, we work on three different applications: (1) The reduction of symmetric banded matrices to tridigonal form is an important task in numerical linear algebra when eigenvalues have to be determined, a fundamental core operation in scientific computing, (2) the solution of random ordinary differential equations (RODEs) is one example of models which incorporate stochastic processes to improve the model in terms of quality and accuracy, and (3) simulating fluid flow, and often applied real-world application, by using the lattice Boltzmann method (LBM).

When it comes to single-GPU issues, we deal with compute- and memory bandwidth-intense kernels, which, on the one hand, fit very well on GPUs. Yet on the other hand, due to the architecture of GPUs, memory latencies become a problem. Therefore, we show how to get the maximum performance from kernels that are latency-bound. Going one step further, when utilizing several GPUs in parallel, we illustrate how to use hybrid programming to tame more than two levels of parallelism and how to efficiently map algorithms to these levels. By hybrid programming, we mean the application of more than one programming paradigm (e.g. by MPI + OpenMP, MPI + CUDA, or a combination of them). Finally, to eventually fully exploit large heterogeneous systems, we demonstrate how to handle different types of computing devices (CPUs and GPUs) simultaneously. This includes tailored implementations of kernels for different kinds of computing devices as well as considering the different properties of the various communication interfaces in a large heterogeneous system. Techniques such as communication hiding are applied to continuously utilize all available compute resources. This is indispensable to achieve scalability on large heterogeneous systems.

From the application point of view, we first take the established symmetric banded matrix, reduction to tridiagonal form via Householder transformations (SBTH) algorithm and implement it on single-GPU setups. We demonstrate why the pipelined approach of the SBTH algorithm harmonizes with the architecture of modern GPUs, making them a single chip supercomputer. Since the SBTH algorithm is one possible step in the determination of eigenvalues, various scientific computing applications relying on such an operation, benefit from this improvement. Afterwards, we introduce a procedure to solve RODEs on GPU clusters. We subdivide the corresponding solution process in four building blocks: Pseudorandom number generation, Ornstein-Uhlenbeck (OU) process, averaging, and coarse timestepping. The first three building blocks are not limited to handle RODEs but are also generally applicable in other domains relying on random input, the OU process, or averaging. Finally, we implement the LBM on large-scale heterogeneous systems equipped with GPUs. The sheer computational performance and memory size of such systems stemming from two types of computing devices enable simulations of size and resolution which were not feasible before. A performance model is given to introduce a possibility to estimate performance values such as runtime, giga (10^9) lattice updates per second (GLUPS) rate, speed-up, and parallel efficiency and to detect bottlenecks on different setups of (also future) heterogeneous systems in advance.

Besides applying well-tried techniques and commonly used best practices for existing algorithms, we present several completely new approaches, methods, and tools in this thesis: Our RODE implementation and the subdivision in four building blocks is the first HPC implementation of a solver for RODEs. Solvers for RODEs are very expensive in terms of computational performance and our implementation allows simulating scenarios of reasonable size for the first time. As a quasi by-product, we offer an implementation of a pseudorandom number generator (PRNG) for normally distributed random numbers achieving—to our knowledge—best performance ever measured on GPUs. Another quasi by-product of the RODE HPC implementation is the first successful parallelization of the OU process by mapping it to prefix sums and taking parallelization schemes for this operation. The implementation of the LBM is the first implementation of this method fully exploiting all available compute resources of a heterogeneous system. Prior hybrid implementations already took the CPU into account but just for communication purposes or to treat single boundary layers of subdomains, thus wasting much of the computational performance of the CPUs. For our implementation, we present a performance model extending existing performance models by considering the CPU as a full computing device, different performance properties of intra and inter-node communication, and the fact that it does significantly matter which dimension is used for parallelization.

1.3. Outline

There are two golden threads running through this thesis. First, differential equations are the central object for modeling: Part II is dealing with the computation of eigenvalues, a totally deterministic operation. Eigenvalues are of frequent interest when it comes to the interpretation of phenomena modeled by differential equations. Part III is dealing with the solution of RODEs. RODEs are a special type of ordinary differential equations (ODEs) which are augmented with a stochastic process. Part IV is dealing with the LBM. The LBM uses an alternative discretization of the partial differential equations (PDEs) which model fluid dynamics. Second, a huge variety of GPU programming aspects is covered: Part II is dealing with features being relevant in the context of single-GPU programming and the usage of libraries. Part III is dealing with problems arising from the usage of multiple GPUs in parallel. Finally, part IV is dealing with challenges arising from reasonably handling large heterogeneous systems.

Before stepping in the three scientific computing applications, part I gives in chapter 2 a presentation of the properties of modern GPUs. It is not a tutorial how to program GPUs but a survey on this type of hardware and a discussion on how GPUs fit in the HPC landscape. This discussion covers similarities and differences between CPUs and GPUs as well as a proper specification of the GPUs and GPU-accelerated supercomputers which are used in this thesis. Part I is concluded by a selection of examples in chapter

3 where GPUs play a major role. On the one hand, this includes some cases of well-established and often-used software in scientific computing accelerated by GPUs. On the other hand, this covers some selected lighthouse projects where the usage of GPUs significantly promoted the simulation's discipline. It shows that there is actually an impact of GPU-accelerated scientific computing in the real world.

Part II starts with a presentation of the SBTH algorithm in chapter 4. A block decomposition scheme for the symmetric banded matrix is given, followed by a serial reduction procedure to tridiagonal form via Householder transformations and completed by a parallel version of the reduction procedure in a pipelined manner. Afterwards, chapter 5 shows the implementation of the SBTH algorithm for GPUs. The algorithm is subdivided in operations which can be mapped to the basic linear algebra sub-routines (BLAS). The main innovation of part II is the mapping of the pipeline character of the parallel SBTH algorithm, introducing another level of parallelsim, to concurrent kernel execution of GPUs. CUDA and OpenCL are used as programming platforms and some operations are delegated to cuBLAS [179], MAGMA [5], and clBLAS [2]. Benchmark results of our implementation of the SBTH algorithm are given in chapter 6, underlining that achieving pipelining, and thus performance, strongly depends on the used library and the capabilities of the GPU.

We suggest four building blocks to give a generally applicable approach to solve RODEs. Before going through them step by step, part III is introduced by a mathematical presentation of RODEs in chapter 7. This chapter contains a correspondence between RODEs and well-known stochastic ordinary differential equations (SODEs) and several schemes to numerically solve RODEs. Chapter 8 introduces the first building block: pseudorandom number generation. Some PRNGs are revised for their suitability on GPUs, two of them for the very first time achieving superior performance. The chapter only deals with normally distributed random numbers which are required for the second building block, the OU process. The OU process is a stochastic process whose novel first time parallelization is demonstrated in chapter 9. Depending on the actual numerical solver, different kinds of averaged values have to be calculated consuming continuous sequences of the OU process. Thus, averaging is the third building block discussed in chapter 10. Besides single and double averages, tridiagonal averages are examined. Finally, the averaged values are plugged in the actual numerical solvers to get a numerical solution of the RODE. This final step is explained in chapter 11. While at the end of chapters 8 to 10 benchmark results of the particular building blocks are given, chapter 12 provides measurements for the whole RODE solution pipeline. On the one hand, these results are the contribution of the single building blocks to the overal runtime, on the other hand, it is the scaling behavior they show when multiple paths are evaluated in parallel in a Monte Carlo-like manner.

The LBM is implemented in part IV. Eventually, we come up with a code for heterogeneous systems. Foundations of the LBM are given in chapter 13. Instead of explicitly dealing with physical values such as velocity, pressure, or density, the LBM deals with density distributions leading to a special discretization scheme. To exploit all available computing devices in a heterogeneous system, the LBM collision and propagation steps have to be implemented adequately, a decent work distribution has to be performed, and a communication hiding by computation strategy is required. Chapter 14 shows how to do this in a parallel way. To estimate the efficiency of our LBM implementation, a performance model is setup in chapter 15. It helps to identify bottlenecks and to predict performance on future heterogeneous systems. Characteristics of the particular kernels, benchmark results of single and multi subdomain scenarios, and a validation of the performance model are given in the last chapter 16 of the LBM part.

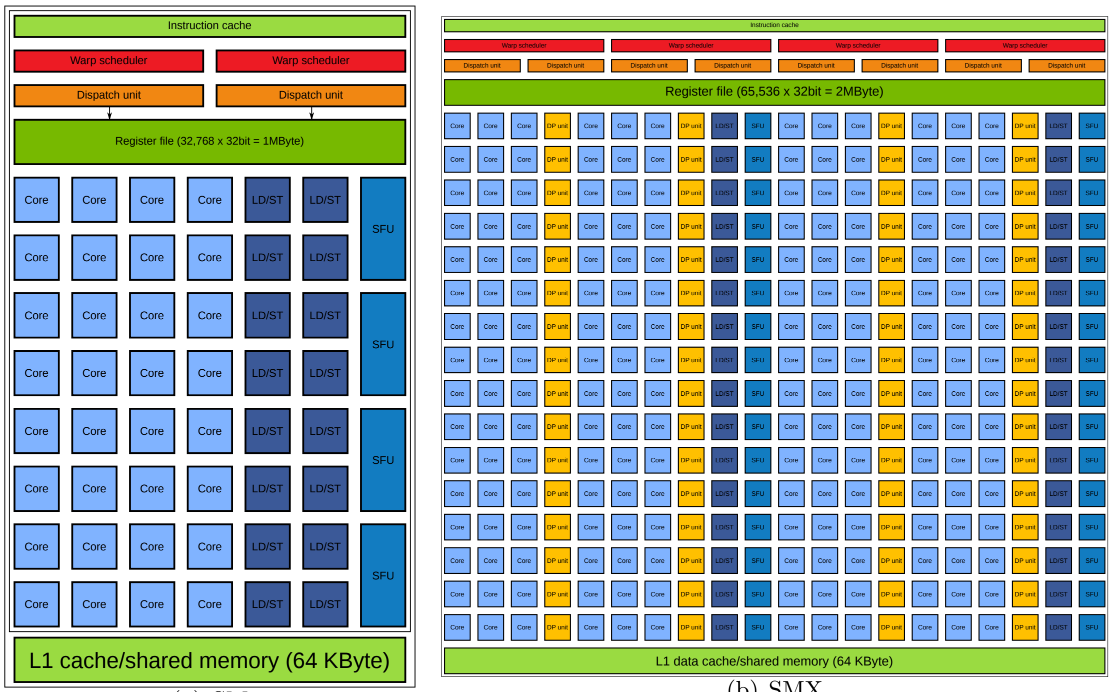
The closing part V summarizes this thesis, outlines the major contributions, and novelties and lists some remarkable numbers such as problem size and parallel efficiency in the context of our work.

At the end of every application part, a specific problem is computed to demonstrate the real-world relevance: For part II, a matrix whose eigenvalues characterize the minimal energy states of a quantum system is processed. The maximum ground motion excitation of an earthquake is determined by the methodology introduced in part III. Finally, a highly resolved lid-driven cavity scenario is executed for part IV running on the largest heterogeneous systems available achieving good scalability.

2. Architecture of GPUs

This chapter gives a survey on GPUs and provides a classification of GPUs in the HPC context. It is neither a compendium to fully cover all facets of GPUs, nor is it a comprehensive tutorial how to write (efficient) code for this kind of architectures. Instead, we first give a rough explanation of the hardware of GPUs in section 2.1 and compare them to the components of CPUs. The two-level parallelism in hardware and the memory hierarchy are the central corner stones of this section. In addition, it lists characteristic numbers of the particular GPUs utilized for this thesis in the subsequent chapters. Afterwards, section 2.2 explains the programming model of and the execution model on GPUs. On the one hand, the programming model enables the application engineer to map parallelism to the two-level parallelism of the hardware, on the other hand, it offers a way to express parallelism in a strictly scalar way. Section 2.3 sketches how the GPU handles millions of software threads and efficiently schedules them on thousands of hardware cores. Furthermore, this section discusses typical bottlenecks of GPUs and provides approaches to overcome them. Both, hardware and software aspects, are illustrated with the ecosystems of the two major GPU manufacturers: NVIDIA, using CUDA, and AMD, using OpenCL. The basic principles of the ecosystems, CUDA and OpenCL, are very similar, just the naming varies. Once the naming is introduced in this chapter for both ecosystems, we rely on NVIDIA nomenclature because NVIDIA technology was mainly, but not exclusively, used for this work. This has no impact on the general applicability of the ideas and solutions presented in this thesis: They work with the products of both companies without any limitations or major differences in performance. Finally, this chapter is completed with a discussion on supercomputers augmented by GPUs in section 2.4. Part of this discussion is the common setup of such clusters in general and the particular clusters used for this thesis in detail. The common setup exemplifies the challenges in the context of efficient communication in heterogeneous systems.

We refer to external literature to get a deeper insight in the field of GPUs besides this survey: The best document to get started with NVIDIA GPUs and their programming with CUDA is the CUDA C Programming Guide [180]. It provides a complete discussion on all aspects of NVIDIA GPUs and the CUDA programming model as well as best practices for performance optimization. To extend this view, we suggest the books of Cook [61] and Wilt [248]. NVIDIA itself recommends the book of Kirk et al. [121] for further studies. The reference for AMD GPUs and OpenCL is given in [7]. For an introduction to heterogeneous computing using OpenCL, we recommend the book by Gaster [83].


Instruction cache
Warp scheduler
Dispatch unit
Register file (32,768 x 32bit = 1MByte)
Core
LD/ST
SFU
L1 cache/shared memory (64 KByte)
Register file (65,536 x 32bit = 2MByte)
DP unit
L1 data cache/shared memory (64 KByte)

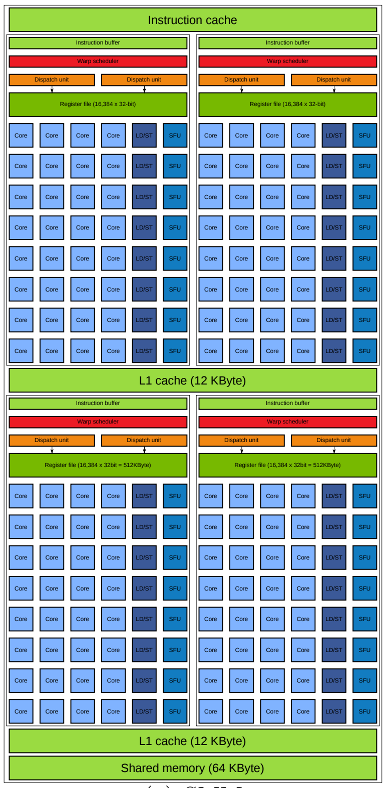
(a) SM (b) SMX

Figure 2.1.: While figure 2.1(a) shows the block diagram of a multiprocessor of NVIDIA's Fermi architecture (SM), figure 2.1(b) shows the block diagram of a multiprocessor of NVIDIA's Kepler architecture (SMX). CUDA cores are colored in blue, memory elements are colored in green, and scheduling elements are colored in red and orange. The color coding and the labeling also hold for figures 2.2 and 2.3.

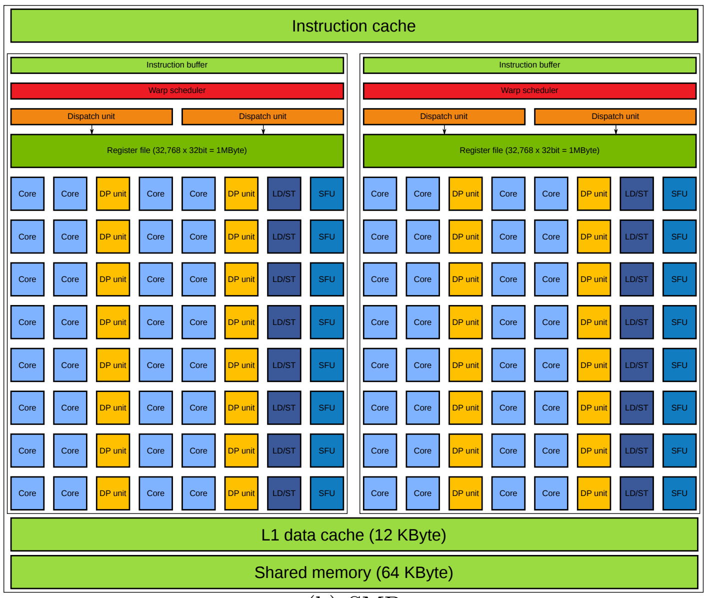
2.1. Hardware structure of GPUs

In this section, we illustrate the hardware internals of today's generally programmable GPUs and do a comparison to the components of CPUs. We focus on the general purpose parts of GPUs and neglect internals dedicated to visualization.

From the beginning, GPUs were designed to provide a high level of parallelism. That is the major difference between CPUs and GPUs. Today's GPUs can consist of thousands of hardware processing elements. The NVIDIA Tesla P40 consists of 3840, the AMD FirePro S9300 x2 of 4096 processing elements. Such devices are called *many-core chips*. In comparison to a CPU core, the processing elements are less sophisticated, but the high degree of parallelism leads to high computational performance with relatively low energy consumption. The hardware parallelism on GPUs is organized in two levels: On the lower level, we have the actual *processing elements* (NVIDIA: *CUDA core*, AMD: *stream processor*), comparable to ALUs. On the higher level, these processing elements are grouped in *multiprocessors* (NVIDIA: *streaming multiprocessor*, AMD: *compute units*).



(a) SMM



(b) SMP

Figure 2.2.: While figure 2.2(a) shows the block diagram of a multiprocessor of NVIDIA's Maxwell architecture (SMM), figure 2.2(b) shows the block diagram of a multiprocessor of NVIDIA's Pascal architecture (SMP). CUDA cores are colored in blue, memory elements are colored in green, and scheduling elements are colored in red and orange.

Depending on the actual GPU architecture, such multiprocessors consist of 32 to 192 processing elements and GPU chips contain between 1 and 64 multiprocessors. The actual processing elements are able to perform the basic arithmetic operations (addition, multiplication, fused multiply/add (FMA)) for integers and floating-point numbers as well as logic operations (and, or, xor, shift). While AMD GPUs include the functionality for all other arithmetical operations (division, transcendental functions, trigonometric functions, etc.) in the particular stream processors, NVIDIA GPUs have a special function unit (SFU) for their execution. Every multiprocessor contains, besides the actual processing units, different kinds of memory, as well as control logic such as load/store units and schedulers. GPU schedulers (NVIDIA: *warp scheduler*, AMD: *scheduler*) are able to do prefetching but not to perform out-of-order execution or branch prediction. The processing elements do not have their own dedicated program counters. Instead, every scheduler has its own program counter and executes threads in a single instruction

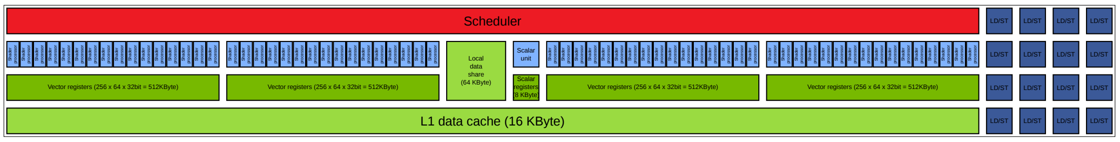

Scheduler
LD/ST
Local data share (64 KByte)
Scalar unit
Scalar registers (8 KByte)
Vector registers (256 x 64 x 32bit = 512KByte)
L1 data cache (16 KByte)

Figure 2.3.: Block diagram of an AMD's CGN compute unit. Each 16 stream processors are grouped in one SIMD-VU. Processing elements are colored in blue, memory elements are colored in green, and scheduling elements are colored in red. Four compute units share 16 KByte read-only L1 data cache and 32 KByte L1 instruction cache (not depicted in this figure).

multiple data (SIMD) (according to Flynn's taxonomy) like manner, explained in more detail in section 2.3.

Figures 2.1 and 2.2 show block diagrams of streaming multiprocessors of four consecutive generations of NVIDIA GPU architectures. Fermi (cf. figure 2.1(a)) is the second, Kepler (cf. figure 2.1(b)) the third, Maxwell (cf. figure 2.2(a)) the fourth, and Pascal (cf. figure 2.2(b)) the fifth generation of generally programmable NVIDIA GPUs. On the

GPU architecture		Fermi		Kepler		Maxwell	Pascal
model		M2050	M2090	K20x	K40m	GTX 750 Ti	P100
chip		GF110		GK110	GK110B	GM107	GP100
compute capability		2.0		3.5		5.0	6.0
#PEs	SP	14×32	16×32	14×192	15×192	5×128	56×64
	DP	-[1]		14×64	15×64	5×4	56×32
SMem (KByte)		16–48 [2]				64	
L1 cache (KByte)		16–48 [2]				12	
BCR (MHz)		1150	1300	732	745	1280	1328
PP (TFLOPS)	SP	1.0304	1.3312	3.935	4.291	1.6384	9.519
	DP	0.5152	0.6656	1.312	1.430	0.0512	4.760
PMBW (GByte/s)		148.4	177.6	249.6	288.384	96.128	719.872
$\frac{\text{FLOP}}{\text{byte}}$ ratio	SP	6.943	7.504	15.765	14.879	17.043	13.223
	DP	3.472	3.752	5.255	4.960	0.533	6.612

Table 2.1.: Properties of all NVIDIA GPUs utilized in this work. PE stands for processing element, SP for single precision (`float`), DP for double precision (`double`), SMem for shared memory, BCR for base block rate, PP for peak performance, and PMBW for peak memory bandwidth.

two older generations, resources (CUDA cores and load/store units) are not exclusively bound to warp schedulers. In theory, this approach offers more flexibility to the warp

[1]The SM microarchitecture of Fermi does not have dedicated double precision units.

[2]In the SM and SMX microarchitecture of Fermi and Kepler, there is one common memory of 64KByte for shared memory and L1 cache which has to be shared amongst them. It can devided in ratios of 2:1 (48KByte shared memory, 16KByte L1 chache), 1:1, and 1:2.

schedulers but they also become more complex. On Kepler, there is another major drawback: Not all resources can be occupied at all time by the schedulers. Hence, on the two latest architectures, every warp scheduler has its exclusive resources. In general, Maxwell and Pascal are very similar architectures: Basically, a Pascal streaming multiprocessor is a halved Maxwell streaming multiprocessor but the amount of memory per streaming multiprocessor (registers, L1 cache, shared memory) stays the same per multiprocessor. Kepler, Maxwell, and Pascal have 192, 128, and 64 single precision (`float`) and 64, 4, and 32 double precision (`double`) CUDA cores per streaming multiprocessor, respectively. Fermi has 32 single precision CUDA cores per streaming multiprocessor, but no double precision CUDA cores. Instead, two single precision CUDA cores are combined to perform double precision operations. Table 2.1 lists properties and characteristics of six different NVIDIA GPUs used throughout this thesis.

GPU architecture		GCN 1st Gen.	
model		FirePro W8000	Radeon HD 8670
chip		Tahiti PRO GL	Oland
#PEs	SP	$28 \times (4 \times 16)$	$6 \times (4 \times 16)$
	DP	-[3]	
local data share (KByte)		64	
L1 cache (KByte)		16	
base block rate (MHz)		900	1150
PP (TFLOPS)	SP	3.2256	0.768
	DP	0.8064	0.048
PMBW (GByte/s)		176	72
$\frac{\text{FLOP}}{\text{byte}}$ ratio	SP	18.327	10.667
	DP	4.582	0.667

Table 2.2.: Properties of all AMD GPUs utilized in this work. PE stands for processing element, SP for single precision (`float`), DP for double precision (`double`), PP for peak performance, and PMBW for peak memory bandwidth.

For AMD GPUs, the basic compute unit design looks slightly different. The last four generations of AMD GPUs base on the graphics core next (GCN) architecture depicted by figure 2.3. Stream units are grouped in four 16-lane wide SIMD vector units (SIMD-VUs) resulting in 64 stream units per compute unit. There is also a scalar stream unit coupled with own scalar registers, but only one such scalar unit is integrated per compute unit. Vector registers are SIMD-VU exclusive. Depending on the GCN generation, the single precision to double precision performance ratio varies from 2:1 to 16:1. There are no dedicated double precision units in AMD GPUs using the GCN technology, but double precision arithmetics is integrated in the SIMD-VUs. Table 2.2 lists properties and characteristics of two different AMD GPUs used in part II.

[3]Depending on the GPU architecture, the single precision to double precision performance ratio varies from 2:1 to 16:1.

There are various types of GPU memory. GPU memories can be divided in off-chip and on-chip memory or in physical and logical memory. While on-chip memory is located on the same die as the processing elements, off-chip memory is located external. Physical memory actually exists in hardware but logical memory only specifies a certain behavior. This results in a classical memory hierarchy: On the one hand, big but slow (in terms of bandwidth and latency) off-chip memory and on the other hand, small but fast on-chip memory. *Global memory* is physical off-chip memory. Its size is up to 32GByte and its role is comparable to classical main memory of CPUs. It offers very high memory bandwidth (up to $\sim$ 720GByte/s) if coalesced memory access is used. *Coalesced memory access* corresponds to a page load where all data of the page is immediately used. *Shared memory* (NVIDIA) or *local data share* (AMD), respectively, is physical on-chip memory and can be seen as some sort of explicit cache. The programmer can decide which data is stored at a specific position of shared memory. It is low-latency memory but its latency is one order of magnitude higher than the clock latency. Another type of GPU memory are *registers*, physically located on the GPU chip itself. In comparison to CPU registers, there is an enormous amount of registers per multiprocessor (cf. figures 2.1 to 2.3). While NVIDIA GPUs only have scalar 32bit registers, AMD GPUs also have vector registers as mentioned above. As for CPUs, there is also *cache memory* for GPUs, structured in multiple levels and physically located on-chip. L1 cache is multiprocessor-exclusive. It is functionally divided into instruction and data cache. L2 cache is shared by all multiprocessors of a GPU chip. Finally, there are three types of memory just having a logical representation. Hence, they do not refer to NVIDIA or AMD hardware but to CUDA or OpenCL. *Local memory* (CUDA) is actually cached global memory. It stores data if there is a lack of registers and it cannot be managed explicitly by the programmer. Further information on register spilling can be found in [162]. Data declared to be saved in *private memory* (OpenCL) is physically stored in registers or, depending on the register consumption, in a similar way to local memory. *Constant memory* stores constants and values passed to kernels. Physically, such values are kept in a dedicated cache. Table 2.3 lists the various types of GPU memory and their classification.

	on-chip	off-chip	hybrid
physical	shared memory registers	global memory	-
logical	constant memory	-	local memory private memory

Table 2.3.: Classification of various types of GPU memory.

The *compute capability* is a version number to identify features supported by NVIDIA GPU hardware. For the NVIDIA GPUs used throughout this thesis, the compute capability is given in table 2.1 (row “compute capability”). Properties of the GPU hardware such as number of registers and shared memory size per multiprocessor can be derived from the compute capability, too[4]. In addition, it specifies abilities of the programming model (discussed in the following section 2.2) and features such as *concurrent kernel execution* and *unified memory*. Regarding abilities of the programming model, the compute capability is similar to the OpenCL version number.

When speaking of GPUs, we have to distinguish between the actual GPU chip and the surrounding hardware which supports the GPU. The GPU chip is a piece of silicon containing the multiprocessors, additional scheduling logic, on-chip memories, and other components. Supporting hardware is e.g. global memory or memory controllers. Both, the GPU chip and the supporting hardware are packed on a dedicated device which is connected to the host system via a—in general relatively slow—bus such as the PCI-express bus or proprietary technology such as NVlink[5]. This layout is not limited to GPUs but holds for most other accelerators, too. It has a direct effect on the layout of heterogeneous systems and, thus, heterogeneous clusters (cf. section 2.4).

2.2. Programming & execution model

There are special programming models which enable the programmer to write software executable on parallel hardware discussed in the previous section 2.1. NVIDIA and AMD use the same programming model for their GPUs. CUDA, a proprietary technology, implements this programming model for NVIDIA GPUs. AMD GPUs are programmed via OpenCL, an open standard for heterogeneous computing maintained by the Khronos group[6]. OpenCL implementations are also available for NVIDIA GPUs and other parallel computing devices such as multi-core CPUs or field programmable gate arrays (FPGAs). Both, CUDA and OpenCL, extend existing programming languages such as C/C++ or Fortran by additional syntax (just CUDA), modifiers, and libraries. Hence, neither CUDA nor OpenCL are stand-alone programming languages. This section presents CUDA's and OpenCL's programming model and the common execution model for GPUs.

To execute code on the GPU, it has to be written as a *kernel*. Kernels are C functions executed in parallel on the GPU by a special invocation. In parallel means, that multiple *threads* (CUDA: *thread*, OpenCL: *work item*) are launched on the device. However, the kernels themselves are written in a scalar way. Vector data types are supported, but vector operations are not available to the programmer. Similar to the hardware, the programming model likewise provides two levels of parallelism: On the lower level, threads are grouped into *blocks* (CUDA: *thread block*, OpenCL: *work group*). On the higher level, these blocks are grouped in a *grid* in CUDA or a *ND range* in OpenCL, respectively. The number of threads per block and the number of blocks per grid have

[4]All properties derivable from the compute capability are given in appendix G of the CUDA C programming guide [180]: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capabilities`

[5]`https://www.nvidia.com/object/nvlink.html`
`https://blogs.nvidia.com/blog/2014/11/14/what-is-nvlink/`
`https://devblogs.nvidia.com/parallelforall/inside-pascal/`

[6]`https://www.khronos.org/`

to be set by the programmer and is in general much higher than the actual number of processing elements. During runtime, a thread can determine its thread and block number as well as the block and grid size via runtime variables (CUDA) or functions (OpenCL) listed in table 2.4.

		CUDA	OpenCL
number of...	...local thread number	`threadIdx`	`get_local_id()`
	...local block	`blockIdx`	`get_group_id()`
	...global thread	-[7]	`get_global_id()`
size of...	...block in threads	`blockDim`	`get_local_size()`
	...grid in blocks	`gridDim`	`get_num_groups()`
	...grid in threads	-[7]	`get_global_size()`

Table 2.4.: Runtime variables (CUDA) and functions (OpenCL) to determine thread number and parallel setup.

The execution model of GPUs is called single instruction multiple threads (SIMT): Multiple threads are simultaneously executing the same instruction. Such bunches of threads are called *warps* in CUDA and *wave fronts* in OpenCL. If processing elements and multiprocessors specify hardware entities and blocks and grid specify entities related to programming, then a warp is located in between as an execution entity. From a hardware point of view, one scheduler issues one instruction on a set of multiple processing elements. From a programming point of view, multiple warps originate from one thread block because a thread block can contain much more threads than the warp size. On NVIDIA hardware, a warp or wave front, respectively, consists of 32 threads. On AMD hardware, a wave front consists of 64 threads. SIMT is very similar to SIMD: All threads of a warp concurrently execute the same instruction, maybe on different data. However, there are differences: While SIMD operations are optional and explicitly issued, SIMT is the default way of execution on the GPU. It's not possible to alter the warp size or to deactivate the SIMT execution model. One possible idea to overcome the limitations of SIMT, making GPUs more flexible, is to interpret GPUs as devices with multiple vector units. In such an approach, the vector size is equal to the warp size and every warp, even of the same thread block, executes different code. The major drawback of this approach is its resource consumption which is the accumulated resource consumption of all the separate code executed by different warps. In addition, the kernels become very complex because they have to contain all the code for the different warps. Another idea to overcome the limitations of SIMT is concurrent kernel execution: Distinct kernels are started simultaneously to run in parallel on the GPU. If the parallel setup of every kernel is chosen small, we get a similar behavior to the first approach. Unfortunately, depending on the compute capability, only a few dozen kernels can be simultaneously executed, thus leading to a very low utilization of the GPU. However, we use concurrent

[7]CUDA does not support global thread numbers. Instead, the global thread number can be calculated by `blockIdx` · `blockDim` + `threadIdx` and the grid size in threads by `gridDim` · `blockDim`

kernel execution to realize the pipelined approach in part II. Due to the SIMT execution model, warp divergence may occur: Conditional statements can force the particular threads of a warp to execute different instructions. Nevertheless, different instructions cannot be executed in parallel within a warp because there is only one program counter per scheduler, thus, serialization occurs.

A thread block and, thus, the warps originating from it, resides on one multiprocessor and is not distributed across several of them. Vice versa, there can be various thread blocks handled by the same multiprocessor. When a kernel is invoked, threads are generated according to the specification of thread blocks and grid given by the programmer. We call this specification *parallel setup* or *grid configuration*.

The execution model and the two-level parallelism determine the visibility of the GPU memories and the lifetime of data stored in them. Global memory is accessible by all threads and it stores data for the whole runtime. Shared memory can only be accessed by the threads of the same thread block. It stores data for the execution time of a single kernel. The same holds for registers. They are only visible to the same thread, hence, one thread cannot access the data stored in a register of another thread.

The major difference between CUDA and OpenCL is the compiling policy for kernels. For CUDA kernels, compiling is done during host code compile time. In contrast, OpenCL includes kernels as plain text in the host executable. OpenCL kernels are automatically compiled right before runtime on the target system. The CUDA approach requires a dedicated compiler (called NVCC) for device code and syntax extensions during compile time while for the OpenCL approach, the host code compiler is sufficient. Thus, the CUDA compiler either has to know the target architecture where the kernels will be executed or it chooses a lowest common denominator to guarantee compatibility of the executable with the target GPU. However, for OpenCL, the GPU driver has to be capable to compile the device code during runtime.

2.3. Scheduling & GPU indicators

The major feature which makes GPUs so powerful is not just their extensive hardware parallelism and, thus, their high theoretical peak performance, but the way how warps are scheduled on multiprocessors. There are many factors which can limit performance such as insufficient memory bandwidth, long memory latencies, and low throughput operations executed by limited hardware resources or suffering from long execution latencies. If a warp has to wait for data or if there is any other reason which prevents the warp from immediate execution, it stalls. In such a scenario, a switch happens meaning the stalled warp is replaced by an active warp, being a warp which can immediately start its execution. GPUs can do such a switch with no cost because the execution context (program counters, registers, etc.) for each warp is maintained by the executing multiprocessor during the entire lifetime of the warp. This enables the GPU to hide latencies and waiting times by warps which can perform operations in a very cheap way. Thus, a high number of active warps is desired because it favors quick switches. A necessary condition to achieve this goal is the existence of a huge number of threads. Hence, a massive oversubscription of the processing elements with threads is recommended.

The ratio of actually active warps to the number of warps which can be active at maximum is called *occupancy*. It is a very important indicator in the context of GPUs. The number of actually active warps depends on the resource consumption (registers and shared memory) of warps, and thus relies on the kernel code. The maximum number of active warps depends on the hardware and is specified by the compute capability. The compute capability gives a limit of the maximum number of active warps per multiprocessor and a limit of the maximum number of active blocks (blocks having active warps) per multiprocessor. The higher the occupancy, the more warps are active. Hence, a high occupancy is advantageous for the scheduler to hide latencies. Thus, increasing the occupancy is one of the major goals to utilize GPUs efficiently, even if high occupancy does not guarantee high performance [235].

Problems are called *memory-bound* if performance is limited by the bandwidth of global memory. In rare cases, the memory bandwidth of shared memory/L1 cache or L2 cache can also limit performance. Thus, minimizing the number of global memory accesses increases performance for memory-bound problems. This can be achieved by spatial and temporal locality of data which increases cache efficiency. Furthermore, spatial locality allows coalesced memory access. Data should be loaded to shared memory if it is required multiple times. Finally, vectorized memory access can improve bandwidth utilization while decreasing the number of executed instructions [144]. Latencies to access global memory can also limit performance. Such problems are called *latency-bound*. There are two approaches to tackle latency-bound problems, described in [40]. First, high occupancy can hide latencies. It can be achieved by optimizing on-chip memory consumption and the parallel setup. Second, an increase of instruction level parallelism can hide latencies. It enables a warp to execute an independent operation while waiting for the data of the halted operation. Latency-bound problems may occur if a kernel does not perform many arithmetical instructions so the memory latency is much longer than the execution of all instructions. If performance is limited by the speed of the processing elements, problems are called *compute-bound*. According to tables 2.1 and 2.2, the FLOPS per byte ratio of GPUs is relatively high in comparison to the ratio of CPUs. Thus, compute-bound problems fit very well on GPUs. In addition, the compute performance of every new GPU generation grows faster than the memory performance.

2.4. Heterogeneous computing & GPU-equipped HPC clusters

Heterogeneous systems are systems which are equipped with at least two different kinds of computing devices. For almost all cases, heterogeneous systems contain one or more CPUs and one or more accelerators of the same type. The accelerators are used in an offloading way, meaning the program control stays with the host which triggers computations on the accelerator. Every device of a heterogeneous system has its own physical memory space which can—but has not to—be shared with other computing devices. There are basically two approaches to exploit an heterogeneous system: First, in a more "homgeneous" way, the same type of computation is carried to the host and the device.

One example for this procedure is presented in part IV where the same LBM operations are executed on the CPU and the GPU. Second, in a more "heterogeneous" way, different categories of work are processed on the computing device fitting best to the task. In section 3.1, the molecular dynamics (MD) software Gromacs is mentioned which assigns different sub-tasks of a MD simulation to the computing device which delivers best performance for the specific sub-task. Mittal et al. provide a survey of heterogeneous computing techniques incorporating GPUs in [164]. There are exceptions to this approach such as Knights Corner, the recent generation of Intel's Xeon Phi.

Heterogeneous clusters are clusters whose nodes are heterogeneous systems or a mixture of CPU-only and heterogeneous nodes. Thus, heterogeneous clusters can be seen as classical supercomputers whose nodes are equipped with (multi-core) CPUs augmented with accelerators such as GPUs. In such a case, the heterogeneous cluster is called *GPU cluster*.

system		JuDGE	Hydra	TSUBAME2.5	Piz Daint
devices	CPU	X5650	E5-2680v2	X5670	E5-2690v3
	#cores/CPU	6	10	6	12
	GPU	M2050	Tesla K20x		Tesla P100
cluster	#CPUs/node	2			1
	#GPUs/node	2		3	
	#nodes	206	338[8]	1442	5320[9]
	interconnect	QDR IB	FDR IB	QDR IB	Aries ASIC
	location	JSC	MPCDF	GSIC	CSCS
software	C++ compiler	GCC 5.0.27	GCC 4.8.0[10] ICPC 16.0[11]	GCC 4.3.4[10] ICPC 15.0.2[11]	ICPC 17.0.0
	CUDA compiler	NVCC 6.5	NVCC 6.5[10]/7.5[11]		NVCC 8
	MPI	ParaStation	IBM 1.4.0[10] Intel 5.1.3[11]	Open 1.8.2	Cray MPICH 7.5.0

Table 2.5.: Properties of all GPU clusters used to benchmark performance in parts III and IV. All CPUs are Intel Xeon CPUs.

To avoid confusion, we disambiguate some terms which are not consistently used in literature: A cluster or supercomputer consists of multiple nodes which are connected by some kind of network such as Ethernet or InfiniBand. Every node contains at least one CPU which is one physical die, also called *package*. CPUs can bundle multiple cores, i.e. processing elements with own program counter, which are programmed in a MIMD manner as shared memory system. In this thesis, no CPU hyperthreads are used.

[8]Only 338 of the total 4000 nodes of Hydra are equipped with GPUs. We limit our benchmarks to these nodes.

[9]Besides the 5320 nodes equipped with GPUs, Piz Daint has 2862 CPU-only homogeneous nodes. We limit our benchmarks to the GPU-equipped nodes.

[10]Configuration for measurements in part III.

[11]Configuration for measurements in part IV.

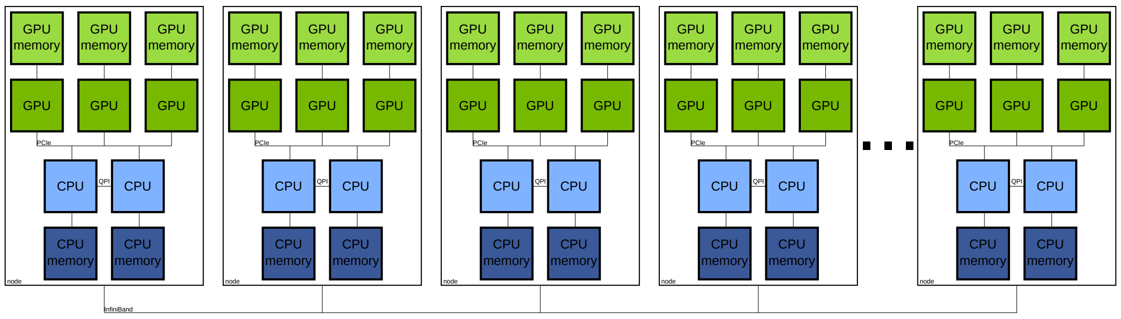


Figure 2.4.: Schematical view of the architecture of TSUBAME2.5. Components related to CPUs and GPUs are colored in blue and green, respectively. Communication between a node's CPUs and GPUs occurs via the PCIexpress bus and between two CPUs via the QuickPath interconnect (QPI). Inter-node communication is achieved via an InfiniBand network.

Heterogeneous systems require hybrid programming. For this work, inter-process communication is achieved via the message passing interface (MPI). Thus, MPI is used for the distributed memory parallelization. Multiple *MPI processes* (also called *ranks*) can reside on one single node. The shared memory parallelization is done via OpenMP. To program the GPUs, we use CUDA for the NVIDIA GPUs and OpenCL for the AMD GPUs. We do not use any technology for virtual memory space unification such as NVIDIA's unified (virtual) memory. Data transfers between different MPI processes as well as between host and device are triggered explicitly, i.e. technologies such as CUDA-aware MPI[12] are not used. The same holds for triggering executions on the particular computing devices, thus technologies such as OpenACC are not used. Due to all these options, hybrid programming is a mixture of established parallel programming models, libraries, and interfaces.

Table 2.5 lists the four GPU clusters used in parts III and IV. While Piz Daint has one CPU per node, JuDGE, Hydra, and TSUBAME2.5 have two CPUs per node each. Also the number of GPUs varies between the GPU clusters: Piz Daint has one GPU per node, JuDGE and Hydra have two GPUs per node, and TSUBAME2.5 has three GPUs per node. A schematical view of TSUBAME2.5 including CPUs, GPUs, and nodes is depicted by figure 2.4. Thus, the four GPU clusters cover a broad range from mid-size to large-scale heterogeneous clusters, equipped with CPUs and GPUs from different generations, and a varying number of CPUs and GPUs per node.

The methods and algorithms and their GPU implementations used in the application parts II to IV cover the whole range of memory-, latency-, and compute-bound problems. They run and are profiled and benchmarked on a broad variety of legacy and state-of-the art GPUs (NVIDIA GPUs are listed in table 2.1, AMD GPUs are listed in table 2.2). Our RODE and LBM solvers are hybrid implementations capable to run on GPU clusters such as the ones listed in table 2.5.

[12]`https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/`

3. Relevance of GPUs in scientific computing

For about a decade, GPUs have been playing an important role in HPC, among others visible via developments: On the one hand, many GPU-optimized libraries and frameworks for various common tasks in scientific computing have arisen during this period. In addition, many popular and state-of-the-art codes and projects of scientific computing applications have been ported partially or completely to the GPU to reduce time to solution. Section 3.1 lists some examples covering software such as building block libraries, toolboxes, and whole codes for scientific computing applications. Due to the sheer amount of successful GPU implementations and portations, section 3.1 can only provide a limited selection. On the other hand, there are lighthouse projects which revealed completely new insights in various fields of science and engineering via simulations accelerated by GPUs. Some selected lighthouse projects are presented in section 3.2. Altogether, this chapter sketches a picture of the current GPU landscape from a scientific computing application point of view, driven by recent advances.

Due to the properties of GPU hardware, some applications, algorithms, and data structures fit very well to GPUs while others do not. By fitting we mean that they can utilize a considerably amount of the GPU resources. Concerning the structures and patterns, such algorithms and data structures are regular, non-adaptive, non-dynamic, parallelizable, and have local communication in most of the cases. Examples are numerical linear algebra operations (see part II) and stencil computations, e.g. [158]. There are successful attempts to efficiently implement irregular algorithms such as graph [98, 139, 161, 28] or adaptive algorithms [236, 11] and complex data structures such as non-uniform [214, 233], hierarchical [257, 131], or sparse grids [166, 167, 80] on the GPU. Furthermore, there are dedicated frameworks dealing with such tasks such as [218, 11] and various fields besides scientific computing such as database systems try to benefit from GPUs [46]. However, they are rather the exception than the norm. The effort for such implementations is big while the potential gain is uncertain. Concerning the categorization of potential bottlenecks, compute-intensive applications fit better on the GPU than memory-intensive ones due to their bad FLOPS per byte ratio. Nonetheless, the memory bandwidth of a GPU is higher than on any host system, hence also memory-intensive applications can benefit from GPUs.

3.1. Acceleration of scientific computing software

Scientific computing highly relies on numerical linear algebra for basic operations. Thus, libraries for numerical linear algebra are one of the most frequently used software categories in scientific computing. Therefore, and because numerical linear algebra operations especially benefits from GPUs, there are various numerical linear algebra implementations for GPUs. BLAS implementations exist both as CUDA (cuBLAS [179]) and OpenCL (BLIS [234], clBLAS [2], and hcBLAS [3]) codes. Furthermore, there are also LAPACK [9] implementations for GPUs. MAGMA [5] exists in a CUDA and an OpenCL implementation. It is not limited to GPUs but some operations are also carried out heterogeneously incorporating the GPU and the CPU using a task model. Having a look on numerical solvers for GPUs, there are geometric multigrid implementations such as [125] as well as algebraic multigrid solvers such as AmgX [170]. The Vienna computing library (ViennaCL) [206] offers a broad variety of iterative solvers besides a BLAS implementation, preconditioners, and functions for singular value decomposition (SVD) and fast Fourier transformation (FFT).

Various MATLAB[1] toolboxes are accelerated by CUDA [255], e.g. leading to a speed-up of 52.92× for matrix/matrix multiplication, 26.74× for FFT, and 2.67× for quick sort [238]. In addition, it is possible to include self-written CUDA kernels in MATLAB code via one-line interface. Going one step further, DUNE and PETSc are two popular representatives of libraries explicitly solving PDEs. As part of the EXA-DUNE project, DUNE was extended by GPU-accelerated components targeting efficient finite element assembly and linear solvers [25]. More specifically, DUNE's GPU implementation of the sparse approximate inverse (SPAI) [91] solver outperforms the CPU implementation by a factor up to 9.5×. PETSc [163] uses the CUSP [65] and Thrust [30] packages from NVIDIA to accelerate its Krylov methods, nonlinear solvers, and integrators.

Computational fluid dynamics (CFD) is a scientific computing application widely used in industry and academia with usage e.g. in the automotive and aerospace sector. Two broadly applied CFD software packages are OpenFOAM [242], a free toolbox, and ANSYS Fluent. For OpenFOAM, there are various GPU-accelerated libraries and solvers such as PARALUTION [146], RapidCFD[2], or Symscape's GPU Linear Solver Library [221]. RapidCFD is an OpenFOAM fork running entirely on the GPU. In industry, the compute jobs per day throughput is one of the most important metrics because it can directly and realistically measure the development productivity. The usage of GPUs in ANSYS Fluent [215] has increased the jobs per day throughput of a truck benchmark, a steady-state pressure-based coupled solver problem, from 16 to 25. A complex Formula 1 car model benchmark, also a steady-state pressure-based coupled solver problem, has been accelerated by a factor of 2.1× when using GPUs. In addition, not only the jobs per day throughput and the time to solution has been improved but also the energy consumption which opens a second opportunity to reduce costs.

Similar to CFD, also molecular dynamics (MD) software benefits from GPUs. Gro-

[1]`https://de.mathworks.com/discovery/matlab-gpu.html`
[2]`https://sim-flow.com/rapid-cfd-gpu/`

macs is specialized to simulate biomolecular systems such as proteins and lipids. Non-bonded force calculations are carried out on the GPU while the CPU simultaneously performs bonded forces and lattice summation [186]. Hence, Gromacs follows an heterogeneous approach where more than 60% of CPU's and GPU's peak performance have been achieved. A GPU-optimization of LAMMPS [49, 48] has accelerated the execution of short-range potentials by factors from 3.7× to 11.2× on a GPU cluster. The speed-ups stem from comparisons with a parallel implementation for multi-core CPU clusters. The authors clearly indicate that the speed-ups highly depend on the cut-off radius σ and the number of simulated molecules. NAMD, a MD software used by 2002's Gordon Bell award winners [191], is specialized on biomolecular and organic systems. Using GPUs, it achieves single-chip speed-ups of 1.7× to 6.4× in comparison to a vectorized and parallelized CPU code [223] and also shows good scaling behavior on GPU clusters [97]. Additional examples for GPU-accelerated computational molecular science can be found in [101].

QUDA [59, 16] is a library for performing calculations in lattice quantum chromodynamics (QCD), the theory of strong interactions, a fundamental force describing the interactions between quarks and gluons, on GPUs written in CUDA. It serves as backend for various QCD simulation codes such as BQCD [169], Chroma [75, 249], and MILC [79]. The Helmholtz Center for Heavy Ion Research (GSI)[3] uses CL^{2}QCD [17, 18, 190] to carry out QCD simulations written in OpenCL. They run the code on the local L-CSC cluster [203], the number-one system of the Green500[4] list when it was installed in 2014. The L-CSC cluster is equipped with 640 AMD GPUs and is currently, according to the Top500 list, the fastest AMD GPU cluster.

In recent years, GPUs became extremely popular in the field of machine learning, especially for deep learning with neural networks. GPUs can massively accelerate (more than one order of magnitude) various tasks in this context such as training and classification because GPUs are efficiently performing matrix multiplications and convolutions. Almost all popular deep learning libraries such as Caffe [115], TensorFlow [4] or Torch [60] exploit GPUs. There, GPUs are not just optional but one of the driving forces for the current success of deep learning [208, 90]. This success has an impact on various fields, e.g. computer vision where GPU-accelerated approaches even outperform human recognition [57]. Thus, GPUs not just increase the level of performance, hence enlarging the quantity in terms of number of items and runtime, but also the level of quality.

3.2. Lighthouse projects

Besides the general impact of GPUs in scientific computing sketched in the previous section, there are also numerous lighthouse projects where GPUs enabled simulations and computations permitting completely new insights in the corresponding field. In the following, five selected lighthouse projects are presented.

[3]`https://www.gsi.de/`

[4]`https://www.top500.org/green500/lists/2014/11/`

Shimokawabe et al. [219] came up with a simulation of the solidification process of metal alloys. The mechanical properties of metal materials largely depend on their internal microstructures. These microstructures develop during the solidification of the metal material, thus, predicting these patterns enables the production of tailored materials. The phase-field model can be used to describe the dentritic growth which occurs during the transition from liquid to solid state. Shimokawabe et al. use the phase-field model for a simulation on the TSUBAME2.0 cluster, the direct predecessor of TSUBAME2.5 which we used for our work. They were able to handle dentritic structures of such size and complexity that they can be reasonably used in material design for the first time. To run such a large simulation, a high scalability of the code had to be assured. For the largest weak scaling scenario, a parallel efficiency of 94% and 1.02 PFLOPS has be achieved using 4000 GPUs and 16,000 CPU cores. This work was honored with a Gordon Bell award in 2011. The domain decomposition scheme which we are using in part IV is inspired by the scheme in Shimokawabe's work even if we are using a different model (LBM).

There are various examples of cosmological simulations which were significantly advanced by the usage of GPUs. An active galactic nucleus (AGN) is the shining compact region at the center of an active galaxy. AGNs can produce jets of plasma which are thousands of light years long. Sufficiently investigating such streams from earth is not feasible: Their shear size and distance make it impossible to view individual electromagnetic particles. However, the radiative signatures of the plasma streams are observable from earth. By simulating the Kelvin-Helmholtz instability (KHI), a property of turbulent plasma, it was possible to correlate the radiative signature with individual particles. To obtain such a correlation, the KHI simulation has to satisfy an adequate degree of resolution. Bussmann et al. [52] were the first to come up with such an adequate degree of resolution, revealing structures like mushrooms or whirlpools in the turbulence. Simulations were carried out on Titan[5], the largest GPU cluster currently available[6] The size of the simulation was 46× larger than any other kinetic KHI simulation previously performed. A parallel efficiency of 96% has been achieved on 18,432 GPUs (weak scaling), resulting in a sustained performance of 7.18 double precision plus 1.45 single precision PFLOPS. This work was a finalist for the Gordon Bell award in 2013. The cosmological simulation code Bonsai was used to simulate 6×10^9 years of evolution of our Milky Way galaxy [29] during which the bar structure and spiral arms were fully formed. To this end, 51×10^9 particles were used, three orders of magnitude more particles than in previous simulations. The resulting simulation data can be directly compared with observations. Simulations have been carried out on Piz Daint, the second largest GPU cluster currently available[6] and whose upgraded version is also used in our work, using 5200 GPUs and Titan using 18,600 GPUs. The largest run on Piz Daint achieved 95% of parallel efficiency and 86% on Titan, respectively (both weak scaling), resulting in a sustained performance of 24.77 PFLOPS. This work was a finalist for the Gordon Bell award in 2014.

[5]`https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/`
[6]`https://www.top500.org/list/2016/11/`

Microfluidics is a multidisciplinary field with practical applications to the design of systems in which low volumes of fluids are processed. One application is the development of lab-on-a-chip technology, e.g. to separate blood and cancer cells. Rossinelli et al. [204] were able to achieve 65.5% of the theoretical 39.4 PFLOPS of Titan by using 18,688 GPUs with their microfluidic simulation code. With this code, they simulated flow involving up to 1.43×10^9 deformable red blood cells, each consisting of 500 elements, in a volume of 132mm^3 at sub-μm resolution. Such volumes allow covering the entire functional compartments of microfluidics devices, confirming experimental findings. By optimizing for GPUs, the authors were able to process 1.8×10^9 cells per second, a three orders of magnitude improvement in comparison to the previous state-of-the-art blood flow simulation [193]. This work was a finalist for the Gordon Bell award in 2015. With such advances, it is possible to accelerate the design cycle for microfluidic systems for medical diagnosis and drug design by an order of magnitude.

Besides the classical application of scientific computing in science and engineering, GPUs recently gained major relevance in deep learning. An example which also earned much attention in popular media[7] is the software AlphaGo [220]. Go is an abstract strategy board game considered to be the most challenging of classic games for artificial intelligence. AlphaGo was first able to defeat the human european Go champion Fan Hui by 5:0 in 2015 and afterwards Lee Sedol, a 9 dan rank player who was considered the best Go player worldwide between 2007 and 2011, by 4:1 in 2016. These results count as one of the major break throughs for artificial intelligence in recent years because mastering Go is associated with intuition and experience. Two types of neural networks are used by AlphaGo: Policy networks provide guidance regarding which action to choose and value networks provide an estimate of the value of the current state of the game. All these networks, two policy networks and one value network, were computed in parallel on up to 280 GPUs during the matches to meet the time constraints. Hence, the success of AlphaGo bases on two pillars: On the one hand, the smart combination of different kinds of neural networks with other algorithms such as Monte Carlo tree search enabled the software to beat the best Go players in the world. On the other hand, the usage of GPUs during the training, reinforcement, and gaming phase made this possible in a reasonable time.

[7]`http://www.bbc.com/news/technology-35785875`
`https://www.theatlantic.com/technology/archive/2016/03/the-invisible-opponent/475611/`

Part II.

Pipelined approach to determine eigenvalues of symmetric matrices

Determining the eigenvalues of a system is one of the most applied tools of linear algebra, eminently in scientific computing. The problem is defined as finding $\lambda \in \mathbb{K}$ (*eigenvalues*) and $x \in \mathbb{K}^n$ (corresponding *eigenvectors*) for a given $A \in \mathbb{K}^{n \times n}$ such that

$$Ax = \lambda x, \qquad x \neq 0. \tag{II.1}$$

Depending on the problem, one specific (e.g. the smallest/largest), several (the $m < n$ smallest/largest, all unique, etc.), or all eigenvalues are of interest. Eigenvalue problems occur in many scientific computing applications: Computational mechanics [26], computational fluid dynamics (CFD) [104], or quantum chemistry [35] are just some examples. In this work, we limit ourselves to the symmetric real-valued eigenvalue problem, i.e. A is symmetric and $\mathbb{K} = \mathbb{R}$, determining all its eigenvalues.

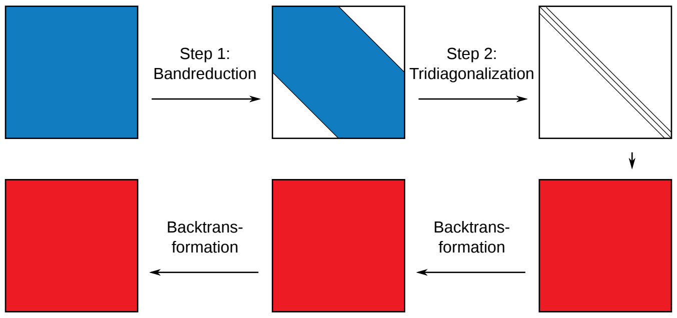


Figure II.1.: Two step approach to transform a symmetric matrix to a tridiagonal symmetric matrix. Step one transforms the matrix into a banded matrix; step two further transforms it into a tridiagonal matrix. The second row shows the according backtransformations. This figure is taken from [12].

One approach to numerically solve eigenvalue problems in parallel splits the actual task into two sub-tasks: First, the symmetric matrix is transformed into a banded symmetric matrix. Second, the banded symmetric matrix is further transformed into a tridiagonal symmetric matrix. These two sub-tasks are illustrated by figure II.1 plus the corresponding backtransformation steps which are necessary to calculate the correct associated eigenvectors. The two sub-tasks approach allows a higher degree of parallelization per sub-task than the classical one task approach. This idea was first presented in [33], improved and extended in [32] and readressed by [13] for the eigenvalue solvers for petaflop applications (ELPA) project [149]. Additional details and an extended timeline is provided in [22]. In this part, we focus on the second transformation step, corresponding to the `SBTRD` routine from LAPACK [9]. LAPACK's reference implementation uses for `SBTRD` an algorithm by Kaufman [119, 120], but we deal with the symmetric banded matrix, reduction to tridiagonal form via Householder transformations (SBTH) algorithm introduced by Lang [128]. It offers interesting properties such an extra level of parallelism well fitting on GPUs. Since we are only dealing with symmetric matrices, we omit this attribute when mentioning matrices. Hence, a symmetric banded matrix is just called banded matrix. The symmetry of the matrices allows a reduction of operations and memory consumption because only the lower (or upper, respectively) triangle has to be processed and stored. We only investigate the transformation step from a banded matrix to a tridiagonal matrix and neglect the first transformation step from a square matrix to a banded matrix as well as the determination of the eigenvalues from the tridiagonal matrix and the backtransformation. Several algorithms for the determination of eigenvalues of a tridiagonal symmetric matrix can be found in [64, 189].

From a modeling point of view, eigenvalues are of interest when it comes to the interpretation of phenomena described by differential equation. From a GPU point of view, this part deals with single-GPU aspects such as concurrent kernel execution and memory-bound kernels as well as the usage of numerical linear algebra libraries. Both, CUDA and OpenCL libraries are employed. The choice of the linear algebra library has a significant impact on the performance, not just because of the performance of particular operations but also because of individual overhead introduced additionally to the actual kernel execution. All operations of the SBTH algorithm can either be mapped to BLAS level 1 or level 2 routines. Usually, this makes GPUs not the first choice for the SBTH algorithm because level 3 routines fit better on GPUs. However, GPUs perform well in this case due to the extra level of parallelism of the SBTH algorithm. We do not discuss how to execute the second transformation step on multiple GPUs or on heterogeneous systems because one single GPU already provides good performance for this task. Thus, no communication between host and device and several GPUs, respectively, is required. Such projects are presented in [96, 254], but they deal with the entire eigenvalue problem.

Lang published the SBTH algorithm in 1993, almost two and a half decades before the writing of this thesis. It was first implemented for the Intel iPSC/860 supercomputer[8]. Although GPUs have quite a different architecture compared to the iPSC/860 and the age of the SBTH algorithm, it fits very well on GPUs. That is a phenomenon observable quite often: Decades-old algorithms, maybe originally designed for vector architectures, experience a revival in the context of GPUs, especially with the background of its SIMT execution model. Hence, not just GPUs benefit from such long-introduced parallel algorithms but SIMD architectures in general.

Part II is structured as follows: In chapter 4, the SBTH algorithm is given. This includes a block decomposition scheme for the banded matrix, the serial reduction to tridiagonal form, and the parallelization of this reduction leading to a two-level parallelization well-fitting on GPUs. While chapter 4 focuses on the mathematical and algorithmical aspects of the SBTH algorithm, chapter 5 deals with the actual implementation. To do so, basic operations are identified which can be mapped to BLAS routines to benefit from corresponding GPU implementations, so the actual computational work is delegated to CUDA and OpenCL libraries and the amount of self-written code is low. In addition, the pipelined structure of the SBTH algorithm can be exploited to increase the degree of parallelism, which is the major contribution of this part, including the challenges arising from practical limitations of GPUs. Finally, our SBTH GPU implementation is benchmarked on various GPUs and a comparison with the performance of the ELPA implementation is drawn. The results are presented in chapter 6.

[8] `https://www.phy.ornl.gov/csep/CSEP/IP/IP.html`

4. The SBTH algorithm

In this chapter, we present the SBTH algorithm introduced by Lang [128]. The SBTH algorithm transforms a banded matrix into a tridiagonal matrix. It is based on the work of Murata et al. [168] and can perform this operation in parallel. The successive application of Householder transformations makes it numerically stable. Alternative approaches using Givens rotations can be found in [212, 213]. The SBTH algorithm avoids fill-in and therefore keeps the banded structure of the matrix. Originally, it was designed for distributed memory systems. In this work, we adapt it for the shared memory architecture of GPUs, thus neglecting all aspects of communication between processing elements. This works well because banded matrices are sparse and, thus, consume little memory. Hence, even large systems can be handled by one single GPU. Our adaptation is not limited to GPUs but their shared memory architecture makes GPUs a single chip supercomputer for this application. Furthermore, some additional aspects of [128] such as the parallelization of the basic operations are neglected in this work. The naming scheme for the identifiers is unaltered taken from [128].

Before the actual SBTH algorithm can be illustrated, a block decomposition scheme for the system matrix A is introduced in section 4.1. It enables the independent execution of different operations on different parts of A. Using the blocking scheme, the iterative SBTH algorithm is explained in section 4.2. The application of Householder transformations successively eliminates single columns of A. Finally, the iterative SBTH algorithm is parallelized in section 4.3 exploiting SBTH's property that different operations can be independently applied on different parts of A. This leads to a pipelined parallel algorithm well-fitting on GPUs due to its two-level parallelism.

4.1. Block decomposition of a banded matrix

Let $A = (a_{i,j})$ $(1 \leq i, j \leq n)$ be a banded matrix with bandwidth d $(2 \leq d \leq n-1)$. Hence, d specifies the number of subdiagonals, or expressed in a more formal way, $a_{i,j} = 0$ for $|i-j| > d$. The SBTH algorithm successively alters A in $n-2$ orthonormal transformations to tridiagonal form. During the ν-th iteration, the ν-th column (or row, respectively) of A is eliminated, i.e. the elements $a_{\nu+2,\nu}, \ldots, a_{\nu+d,\nu}$ are made zero. Thus, after the ν-th iteration, the first ν columns (or rows, respectively) of A have tridiagonal form, resulting in a leading $\nu \times \nu$ sub-matrix $T^{(\nu)}$.

The current state of A at the beginning of the ν-th iteration is denoted by $A^{(\nu)}$ (with $A^{(1)} = A$) and a partitioning illustrated in figure 4.1 is applied. There are $b-1$ blocks $A^{(\nu)}_{\beta,\beta} \in \mathbb{R}^{d \times d}$ $(1 \leq \beta < b)$ along the (non-tridiagonal) diagonal of $A^{(\nu)}$ and one final block $A^{(\nu)}_{b,b} \in \mathbb{R}^{r \times r}$. The relationship between n, d, ν, b, and r is given by $n - \nu = (b-1) \cdot d + r$

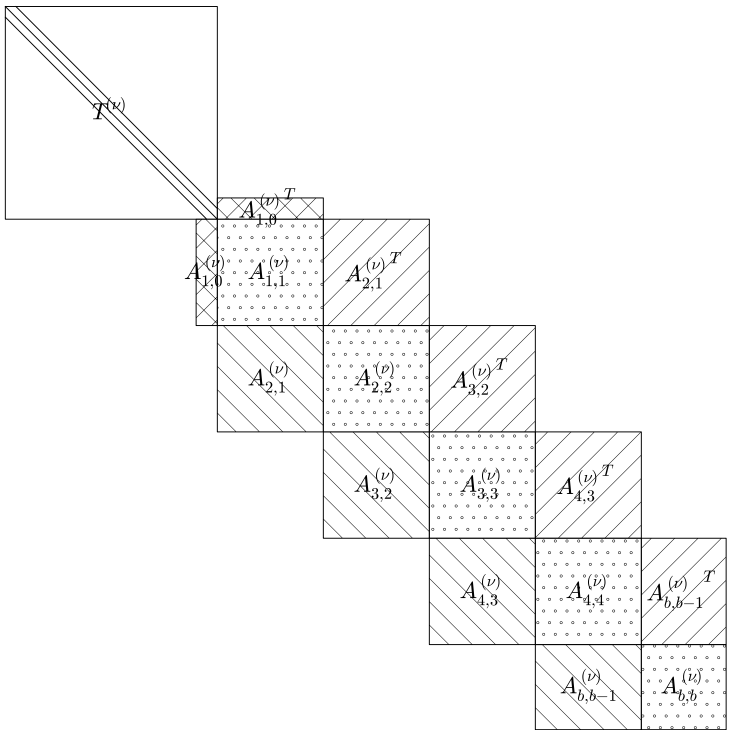


Figure 4.1.: Partitioning of $A^{(\nu)}$ with $b = 5$. The leading tridiagonal $\nu \times \nu$ sub-matrix is denoted by $T^{(\nu)}$. Diagonal blocks are denoted by $A_{\beta,\beta}^{(\nu)}$ and hatched with dots, subdiagonal blocks by $A_{\beta+1,\beta}^{(\nu)}$ and diagonally hatched. The column (or row, respectively) to be eliminated in the ν-th step is denoted by $A_{1,0}^{(\nu)}$ and hatched with crosses.

($1 \le r \le d$), where r is simply the size of the final diagonal block. Below (or right, respectively) the diagonal blocks, there are $b-2$ subdiagonal blocks $A_{\beta+1,\beta}^{(\nu)} \in \mathbb{R}^{d\times d}$ ($1 \le \beta < b-1$) and one final block $A_{b,b-1}^{(\nu)} \in \mathbb{R}^{r\times d}$. $A_{1,0}^{(\nu)} \in \mathbb{R}^d$ denotes the column (or row, respectively) to be eliminated in the ν-th iteration.

4.2. Serial reduction

We go through the SBTH algorithm step by step. Blocks marked with $\sim$ refer to transformed blocks altered in the current iteration ν. Q denotes an orthonormal transformation in matrix form, i.e. $Q^TQ = I$, more specifically a Householder transformation.

The $n-2$ orthonormal transformations mentioned in section 4.1 can be expressed via

$$A^{(\nu+1)} = {Q^{(\nu)}}^T \cdot A^{(\nu)} \cdot Q^{(\nu)}, \qquad 1 \le \nu \le n-2 \tag{4.1}$$

with $Q^{(\nu)} \in \mathbb{R}^{n\times n}$ orthonormal. Considering the block decomposition scheme from section 4.1, equation (4.1) translates to

$$\begin{aligned}
\tilde{A}^{(\nu)}_{1,0} &= Q^{(\nu)^T}_{1} \cdot A^{(\nu)}_{1,0} \\
\tilde{A}^{(\nu)}_{\beta,\beta} &= Q^{(\nu)^T}_{\beta} \cdot A^{(\nu)}_{\beta,\beta} \cdot Q^{(\nu)}_{\beta}, \qquad 1 \le \beta \le b \\
\tilde{A}^{(\nu)}_{\beta+1,\beta} &= {Q^{(\nu)}_{\beta+1}}^{T} \cdot A^{(\nu)}_{\beta+1,\beta} \cdot Q^{(\nu)}_{\beta}, \qquad 1 \le \beta < b.
\end{aligned}$$

At the beginning of the algorithm ($\nu = 1$), the subdiagonal blocks $A^{(\nu)}_{\beta+1,\beta}$ for $1 \le \beta < b-1$ are upper triangular and the final subdiagonal block $A^{(\nu)}_{\beta,\beta-1}$ is upper trapezoidal, i.e. $a_{i,j} = 0$ for $i - j > 0$. Later, for $\nu > 1$, all subdiagonal blocks are full without any special pattern or structure. The diagonal blocks $A^{(\nu)}_{\beta,\beta}$ with $1 \le \beta \le b$ are symmetric for all $1 \le \nu \le n-2$.

Algorithm 1 The serial SBTH algorithm

```
1:  procedure SBTH_SERIAL(in/out: $A$, in: $n$, in: $d$)
2:      for $\nu \in \{1,\dots,n-2\}$ do
3:          Adopt the block decomposition for iteration $\nu$
            according to figure 4.1
4:          $b \leftarrow \lceil \frac{n-\nu}{d} \rceil$
5:          $r \leftarrow n - \nu - (b-1)\cdot d$
6:          Determine Householder transformation $Q^{(\nu)}_1$ such
            that $\tilde{A}^{(\nu)}_{1,0} = Q^{(\nu)^T}_1 \cdot A^{(\nu)}_{1,0}$ has the form $(*,0,\dots,0)^T$
7:          $A^{(\nu)}_{1,0} \leftarrow \tilde{A}^{(\nu)}_{1,0}$                 ▷ Eliminate $\nu$-th column
8:          for $\beta \in \{1,\dots,b\}$ do
9:              $\tilde{A}^{(\nu)}_{\beta,\beta} \leftarrow Q^{(\nu)^T}_{\beta} \cdot A^{(\nu)}_{\beta,\beta} \cdot Q^{(\nu)}_{\beta}$
10:             $A^{(\nu)}_{\beta,\beta} \leftarrow \tilde{A}^{(\nu)}_{\beta,\beta}$          ▷ Update $\beta$-th diagonal block
11:             if $\beta < b$ then   ▷ There are $b$ diagonal but only $b-1$ subdiagonal blocks
12:                 Determine Householder transformation $Q^{(\nu)}_{\beta+1}$
                    such that the first column of
                    $\tilde{A}^{(\nu)}_{\beta+1,\beta} = {Q^{(\nu)}_{\beta+1}}^{T} \cdot \left(A^{(\nu)}_{\beta+1,\beta} \cdot Q^{(\nu)}_{\beta}\right)$
                    has the form $(*,0,\dots,0)^T$
13:                 $A^{(\nu)}_{\beta+1,\beta} \leftarrow \tilde{A}^{(\nu)}_{\beta+1,\beta}$     ▷ Update $(\beta+1)$-th subdiagonal block
```

Algorithm 1 depicts the serial SBTH algorithm as pseudocode. After adopting the block decomposition for $A^{(\nu)}$ (lines 3–5), the ν-th column is eliminated. A Householder transformation, expressed by $Q^{(\nu)}_1 \in \mathbb{R}^{d\times d}$, is determined to set all but the first components to zero when applied to $A^{(\nu)}_{1,0}$, i.e. $\tilde{A}^{(\nu)}_{1,0} = Q^{(\nu)^T}_1 \cdot A^{(\nu)}_{1,0}$ has the form $(*,0,\dots,0)^T$ (lines 6–7). To keep $A^{(\nu)}$ consistent, $Q^{(\nu)}_1$ has also to be applied to the remainder of $A^{(\nu)}$, not just $A^{(\nu)}_{1,0}$. Hence, $Q^{(\nu)}_1$ is successively applied to all diagonal blocks $A^{(\nu)}_{\beta,\beta}$ and subdiagonal blocks $A^{(\nu)}_{\beta+1,\beta}$ (line 8) in an interleaved way: For a given β, first, the diagonal block is updated (line 10) before the subdiagonal block is updated (line 13). Considering the block decomposition for $A^{(\nu+1)}$ for the next iteration, the update of the subdiagonal block has to be modified: Since the block decomposition is shifted by one row to the bottom and one column to the right for every new iteration, components at positions $(2,1),\ldots,(d,1)$ of each subdiagonal block $A^{(\nu)}_{\beta+1,\beta}$ would get lost because they are not considered by any block of the $(\nu+1)$-th iteration. Thus, one has to make sure that the first column of $A^{(\nu)}_{\beta+1,\beta}$ is also eliminated after applying $Q^{(\nu)}_{\beta}$, i.e. has form $(*,0,\ldots,0)^T$ avoiding fill-in and keeping the banded structure. To perform this elimination, the Householder transformation $Q^{(\nu)}_{\beta}$ has to be modified, resulting in a new update $Q^{(\nu)}_{\beta+1}$ (line 12) which is used for the next pair of diagonal and subdiagonal block. The whole procedure is repeated for all columns of A (outer loop in line 2). Determining the correct Householder transformations is presented in sections 5.1 and 5.2.

4.3. Parallel reduction

In the following, a block pair of diagonal block $A^{(\nu)}_{\beta,\beta}$ and the associated subdiagonal block $A^{(\nu)}_{\beta+1,\beta}$ is denoted by $B^{(\nu)}_{\beta} = \begin{pmatrix} A^{(\nu)}_{\beta,\beta} \\ A^{(\nu)}_{\beta+1,\beta} \end{pmatrix}$ $(\beta \geq 0)$.

For now, the SBTH algorithm first determines a transformation to eliminate the ν-th column and then successively applies this transformation to all block pairs of $A^{(\nu)}$ before the $(\nu+1)$-th elimination is issued. The transformation has to be successively adapted to avoid fill-in. To parallelize the SBTH algorithm, the determination and application of the next transformations $Q^{(\nu+1)}_{\beta}$ is already started while the previous transformations $Q^{(\nu)}_{\beta+1}$ are still determined and applied. By following this idea, the more transformations are applied simultaneously, the higher the degree of parallelism becomes. This parallelization approach only works correctly if the sequence of transformations is kept, so $Q^{(\nu+1)}_{\beta}$ cannot be handled before $Q^{(\nu)}_{\beta}$ and $Q^{(\nu)}_{\beta+1}$ before $Q^{(\nu)}_{\beta}$, respectively.

The modified, parallel version of the SBTH algorithm is given by algorithm 2. By splitting each iteration into two distinct steps (line 2), it is possible to maintain the parallel SBTH algorithm synchronized. The iteration number ν can be derived from the step number via $\nu = \lfloor \frac{\text{step}}{2} \rfloor + 1$ (line 3) but ν will be altered during the current step. During each step, basically the same operations as for the serial SBTH algorithm are applied, but there is a slight modification distinguishing even and odd steps: For even steps (line 4), the initial transformation $Q^{(\nu)}_{1}$ is determined (line 5) and the adapted transformation $Q^{(\nu)}_{\beta}$ is applied to all $B^{(\nu)}_{\beta}$ with even β (line 8). For odd steps (line 15), only the adapted transformation $Q^{(\nu)}_{\beta}$ is applied to all diagonal and subdiagonal blocks with odd index β (line 16). The iteration index ν has to be adapted after every transformation of a block pair $B^{(\nu)}_{\beta}$ (lines 7, 14, and 21) because the transformation of the $(\beta+2)$-th block pair stems from the previous iteration. Lines 9–13 and 17–20 are copied

Algorithm 2 The parallel SBTH algorithm

```
 1: procedure SBTH_PARALLEL(in/out: $A$, in: $n$, in: $d$)
 2:     for step $\in \{0, \dots, 2 \cdot (n-2) - 1\}$ do
 3:         $\nu \leftarrow \lfloor \frac{\text{step}}{2} \rfloor + 1$, $b \leftarrow \lceil \frac{n-\nu}{d} \rceil$, $r \leftarrow n - \nu - (b-1) \cdot d$
 4:         if step % 2 == 0 then                                        ▷ For even steps
 5:             Determine Householder transformation $Q_1^{(\nu)}$ such
                that $\tilde{A}_{1,0}^{(\nu)} = Q_1^{(\nu)^T} \cdot A_{1,0}^{(\nu)}$ has the form $(*, 0, \dots, 0)^T$
 6:             $A_{1,0}^{(\nu)} \leftarrow \tilde{A}_{1,0}^{(\nu)}$    ▷ Eliminate $\nu$-th column, only necessary for even steps
 7:             $\nu \leftarrow \nu - 1$                ▷ Last block in pipeline stems from previous iteration
 8:             for $\beta \in \{2, 4, 6, \dots, \min(\text{step}, b)\}$ do      ▷ Only even blocks are updated
 9:                 $\tilde{A}_{\beta,\beta}^{(\nu)} \leftarrow Q_\beta^{(\nu)^T} \cdot A_{\beta,\beta}^{(\nu)} \cdot Q_\beta^{(\nu)}$
10:                 $A_{\beta,\beta}^{(\nu)} \leftarrow \tilde{A}_{\beta,\beta}^{(\nu)}$                   ▷ Update $\beta$-th diagonal block
11:                 if $\beta < b$ then
12:                     Determine Householder transformation $Q_{\beta+1}^{(\nu)}$
                        such that the first column of
                        $\tilde{A}_{\beta+1,\beta}^{(\nu)} = Q_{\beta+1}^{(\nu)^T} \cdot \left( A_{\beta+1,\beta}^{(\nu)} \cdot Q_\beta^{(\nu)} \right)$
                        has the form $(*, 0, \dots, 0)^T$
13:                     $A_{\beta+1,\beta}^{(\nu)} \leftarrow \tilde{A}_{\beta+1,\beta}^{(\nu)}$        ▷ Update $(\beta + 1)$-th subdiagonal block
14:                 $\nu \leftarrow \nu - 1$        ▷ Next block in pipeline stems from previous iteration
15:         else                                                         ▷ For odd steps
16:             for $\beta \in \{1, 3, 5, \dots, \min(\text{step}, b)\}$ do      ▷ Only odd blocks are updated
17:                 $\tilde{A}_{\beta,\beta}^{(\nu)} \leftarrow Q_\beta^{(\nu)^T} \cdot A_{\beta,\beta}^{(\nu)} \cdot Q_\beta^{(\nu)}$
18:                 $A_{\beta,\beta}^{(\nu)} \leftarrow \tilde{A}_{\beta,\beta}^{(\nu)}$                   ▷ Update $\beta$-th diagonal block
19:                 if $\beta < b$ then
20:                     see lines 13 and 14
21:                 $\nu \leftarrow \nu - 1$        ▷ Next block in pipeline stems from previous iteration
```

from lines 9–13 of algorithm 1. They perform the actual transformation and adaptation of $Q_\beta^{(\nu)}$. Since for every step only every second $B_\beta^{(\nu)}$ is processed, these processes do not interfere with each other. Hence, the loops in lines 8 and 16 are parallelizable.

Figures 4.2 to 4.4 illustrate the block decomposition and the block pairs processed during steps 0 to 5, 25, and finally 41 to 43 before the parallel SBTH algorithm terminates. Beginning from subfigure 4.3(a), it becomes visible that the block decomposition is not as consistent as in figure 4.1 for the serial SBTH algorithm because during one step it is possible that multiple transformations from consecutive iterations are applied. The SBTH algorithm requires $2 \cdot b - 1$ steps before it reaches its maximum degree of parallelism because it takes $2 \cdot b - 1$ steps until $Q_\beta^{(1)}$ migrates through the block pairs $B_\beta^{(\nu)}$ (cf. subfigure 4.3(d)). Furthermore, the degree of parallelism becomes smaller and smaller towards the end of the parallel SBTH algorithm because $\nu - 1$ columns are al-

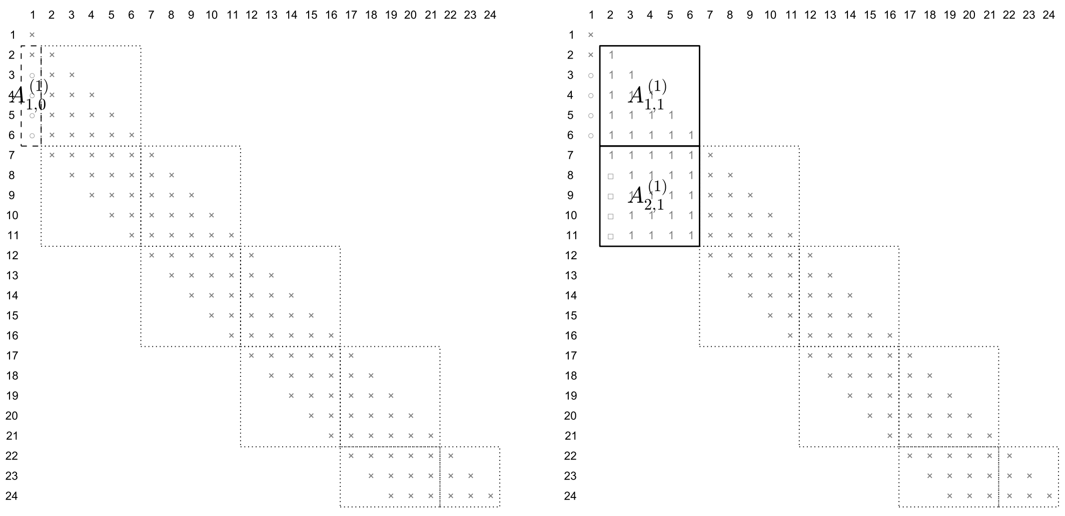
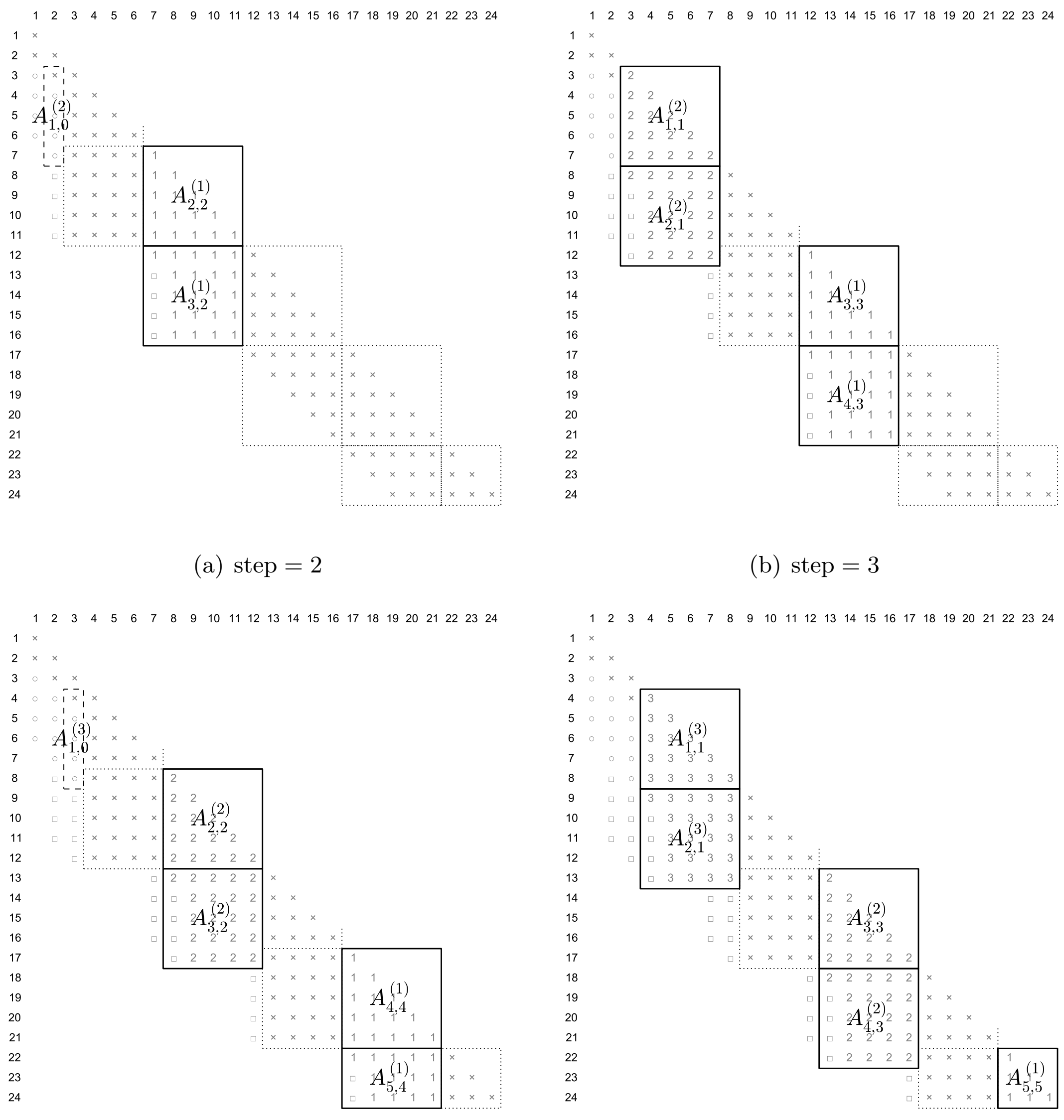

(a) step = 0 (b) step = 1

Figure 4.2.: Block decomposition and block pairs processed by the SBTH algorithm for $n = 24$, $d = 5$, and steps 0 and 1. $\times$ marks values $\neq 0$, $\bigcirc$ marks elements regularly eliminated during the ν-th iteration, and $\square$ marks elements eliminated during the processing of a subdiagonal block to avoid fill-in. Solid framed blocks are processed in the current step, dashed framed blocks are eliminated during the ν-th iteration, and dotted framed blocks are not touched in this step to avoid race conditions. The digits in currently processed blocks indicate the iteration ν to which to local block decomposition belongs. Due to symmetry, only the lower triangle is considered.

ready eliminated and these columns are not processed anymore (cf. subfigures 4.4(a) to 4.4(d)), i.e. the size of the banded remainder shrinks. Thus, only $n - \nu$ columns are still processed leading to a decreasing number of block pairs b. This pattern of first generating more and more work, adding new tasks while old tasks are processed, and finally running out of work makes it a pipelined algorithm. All elements currently residing in the pipeline can be processed in parallel. The workload per item in the pipeline, i.e. the workload to process a block pair, is equal for every item so no load balancing mechanism is required. Pipeline items can easily be evenly distributed among processing elements. Furthermore, the actual workload can be predicted accurately for every n, d, and ν.

Hence, there are two levels of parallelism in the parallel SBTH algorithm: On the one hand side, there is the pipeline whose elements can be executed in parallel. On the other hand side, there are the diagonal and subdiagonal block transformations themselves which can also be parallelized. The next chapter deals with the implementation details of the pipeline as well as the parallelization of the transformations.



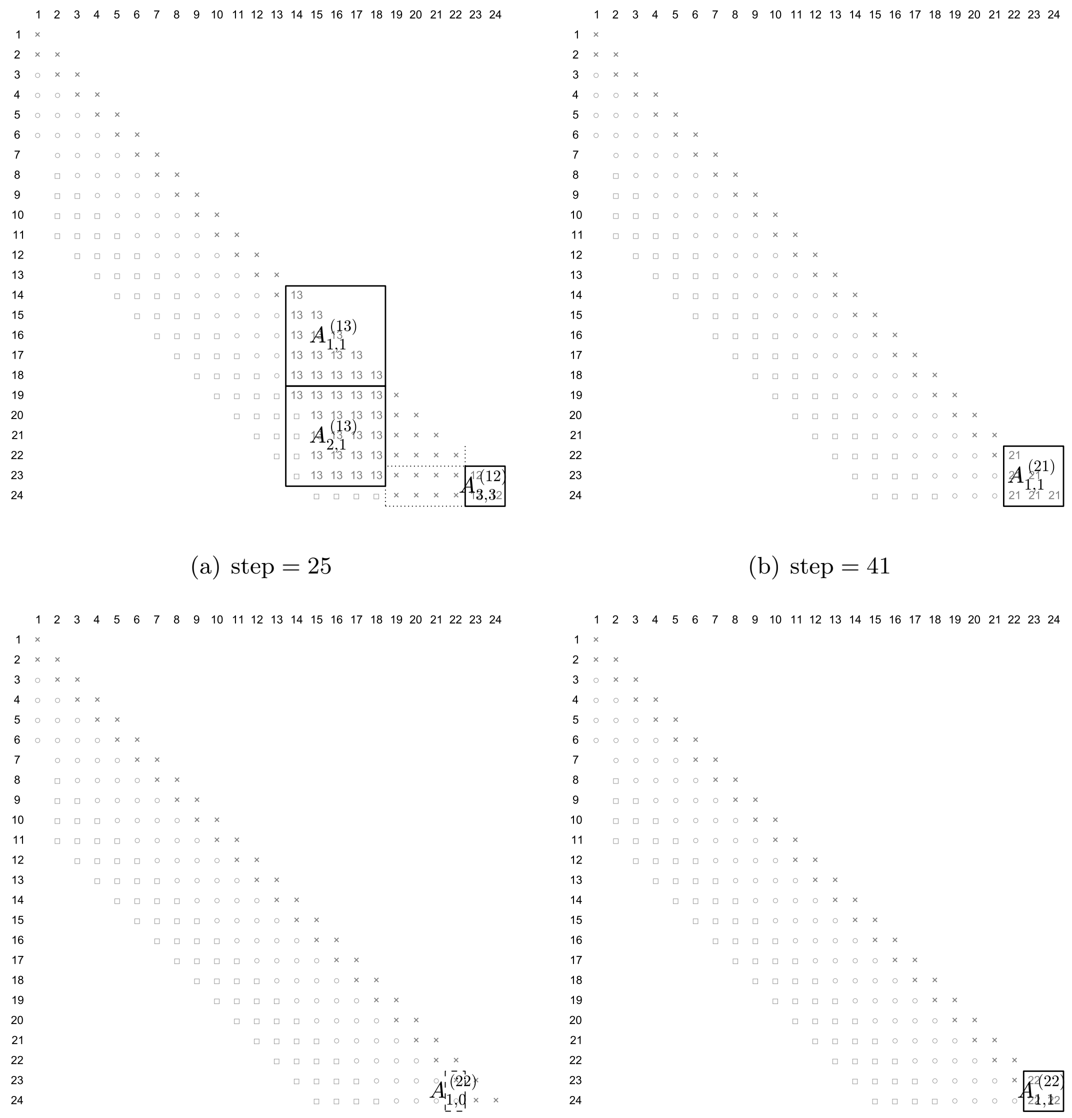
(a) step = 2

(b) step = 3

(c) step = 4

(d) step = 5

Figure 4.3.: Block decomposition and $B^{(\nu)}_{\beta}$ processed by the parallel SBTH algorithm for $n = 24$, $d = 5$, and steps 2 to 5. Marks ($\times$, $\bigcirc$, and $\square$) and frames convention correspond to marks and frames convention of figure 4.2. Solid framed blocks currently reside in the pipeline and, thus, are processed in parallel.


(a) step = 25

(b) step = 41

(c) step = 42

(d) step = 43

Figure 4.4.: Block decomposition and $B_\beta^{(\nu)}$ processed by the parallel SBTH algorithm for $n = 24$, $d = 5$, and steps 25 and 41 to 43. Marks ($\times$, $\bigcirc$, and $\square$) and frames convention correspond to marks and frames convention of figure 4.2. Solid framed blocks currently reside in the pipeline and, thus, are processed in parallel.

5. Implementation of the SBTH algorithm

For now, we discussed the algorithmical aspects of the SBTH algorithm. In this chapter, we investigate further implementation aspects of the SBTH algorithm, mainly the determination and application of Householder transformations on parts of the matrix and the mapping of linear algebra operations to BLAS routines.

The implementation presented in this chapter triggers operations from the host side which are then executed on the device. Hence, the program control stays on the CPU while the linear algebra operators are performed on the GPU to benefit from its computational and memory performance. From a technical point of view, it would be possible to completely shift the program control from the host side to device, resulting in a reduced overhead of function calls and kernel invocations. The major drawback of this idea would be a significant loss in flexibility concerning program control and extensibility. Thus, we do not further pursue this approach. Since all operations are performed on the GPU, there is no necessity to copy data, e.g. particular block pairs, between host and device during runtime. This reduces overhead and complexity.

The actual linear algebra operations required by the SBTH algorithm are not implemented by ourselves but left to libraries implementing the BLAS routines. On the one hand, this enables us to simply switch between different BLAS implementations for the GPU and compare their performance and suitability for the pipelined approach. On the other hand, we benefit from the quality and experience invested in these BLAS implementations. A drawback of this policy is the absence of some optimization techniques such as *kernel fusion*, i.e. merging multiple kernels into one single kernel. Moreover, successful caching is made more difficult. The SBTH algorithm only relies on BLAS level 1 and level 2 routines. Both are memory-bound, the computational load is low. We abbreviate the BLAS routines according to the BLAS quick reference guide [1] and correspondingly use the parameter signatures in the code listings 5.1 and 5.2 in this chapter. In particular, five different BLAS level 1 routines are required by the SBTH algorithm: (1) vector sum (`AXPY`), (2) vector copy (`COPY`), (3) dot product (`DOT`), (4) euclidean norm (`NRM2`), and (5) vector scaling (`SCAL`). From the set of BLAS level 2 routines, four are actually applied: (1) general matrix/vector product (`GEMV`), (2) general rank-1 update (`GER`), (3) symmetric rank-2 update (`SYR2`), and (4) symmetric matrix/vector product (`SYMV`). An explanation of rank-k updates is given at the end of section 5.2.

Householder transformations are required twice in the SBTH algorithm: First, they are needed for the elimination of the ν-th column of A. We explain in section 5.1 how to find and apply such a transformation. Second, the transformations from section 5.1 are used to update the block pairs leading to a new transformation to avoid fill-in in the subdiagonal blocks. Section 5.2 shows how the block pairs are updated and the new transformation is determined. Since pipelining is the most relevant feature of the SBTH algorithm for us, we discuss the usage of CUDA streams and OpenCL command queues in section 5.3 to achieve parallelism within the pipeline. Finally, we present in section 5.4 a simple but very effective storage scheme for matrix A exploiting its banded structure.

5.1. Determination of Householder transformations

In section 4.2, we are using Householder transformations to eliminate the ν-th column but we do not yet explain how to calculate a correct Householder transformation. Their determination is now demonstrated in this section.

Algorithm 3 Determination of a Householder vector v whose corresponding orthonormal transformation eliminates all but the first components of a vector x. $||\cdot||_2$ denotes the euclidean norm and $\text{sgn}(x)$ the sign function.

```
1: procedure FIND_HOUSEHOLDER(in/out: $x \in \mathbb{R}^d$, out: $v \in \mathbb{R}^d$, out: $\tau \in \mathbb{R}$)
2:     $\rho \leftarrow \text{sgn}(x_1) \cdot ||x||_2$        ▷ $x_1$ is the first component of $x$
3:     $v \leftarrow \frac{1}{x_1+\rho} \cdot x$
4:     $v_1 = 1$        ▷ The first component of the Householder vector is always 1
5:     $\tau \leftarrow \frac{x_1+\rho}{\rho}$        ▷ Tangent of the rotation angle
6:     $x \leftarrow (-\rho, 0, \ldots, 0)^T$        ▷ Elimination of $x$
```

Instead of storing the whole Householder matrix Q describing the Householder transformation, we use Householder vectors v equivalent to Q. Using Householder vectors instead of Householder matrices offers two advantages: First, storing the Householder transformations is much more efficient, since only $\mathcal{O}(d)$ memory is needed instead of $\mathcal{O}(d^2)$. Second, the application of the transformations can be done with $\mathcal{O}(d^2)$ rather than $\mathcal{O}(d^3)$ instructions per $d \times d$ block. v is the vector orthonormal to the reflection hyperplane of the Householder transformation. The relationship between Householder matrix and Householder vector is given by $Q = I - \tau vv^T (\tau \in \mathbb{R})$ with τ expressing the tangent of the rotation angle of Q. We show how to determine a Householder vector v whose corresponding orthonormal transformation eliminates all but the first components of a vector $x = (x_1, \ldots, x_n)^T$, i.e. $Q^T x = (*, 0, \ldots, 0)^T$. This algorithm is taken from [129] and depicted by algorithm 3. It eliminates x and returns v and τ. Furthermore, we provide a simplified code snippet implementing the algorithm in listing 5.1. The code directly works on A and eliminates its ν-th column, corresponding to lines 6 and 7 of algorithms 1 and 2. The listing illustrates the mapping of the linear algebra operations of algorithm 3 to BLAS routines, more specifically, only BLAS level 1 routines have to be carried out. The most expensive operation of them on the GPU is by far `NRM2`.

```
void apply_householder(A, d, nu, v)
{
    // 1 / (A(nu+2,nu+1)+sgn(A(nu+2,nu+1))*||A(nu+2:nu+d+1,nu)||)
    norm = NRM2(d, A[nu+1][nu], 1);
    sigma = sgn(A[nu+1][nu]) * norm;
    norm = 1. / (A[nu+1][nu] + sigma);
```

```
 7      // First component of Householder vector is always 1
 8      v[0] = 1.;
 9      // Copy remaining components of Householder vector from A
10      COPY(d - 1, A[nu+2][nu], 1, v[1], 1);
11      // Scale all but the first components of Householder vector
12      SCAL(d - 1, norm, v[1], 1);
13      // Scale Householder vector to length sqrt(2)
14      norm = NRM2(d, v, 1);
15      norm = M_SQRT2 / norm;
16      SCAL(d, norm, v, 1);
17      // Eliminate nu-th column by setting it to (-sigma, 0, ..., 0)
18      A[nu+1][nu] = -sigma;
19      COPY(d - 1, ZERO, 0, A[nu+2][nu], 1);
20  }
```

Listing 5.1: Simplified code to determine the Householder vector v and eliminate the ν-th column of A by using BLAS level 1 operations. Variable declaration and memory allocation are neglected. This code works directly on matrix A, stored linearly in memory. The `[][]` notation is used to make the code more readable. Indexing is 1-based for comments and 0-based in code. `ZERO` and `ONE` are the scalar constants 0 and 1, `M_SQRT2` corresponds to $\sqrt{2}$.

Although it is possible, not all operations are delegated to BLAS routines. Lines 5, 6, 15, and 18 only perform scalar instructions. In regards to performance, it does not make sense to call an external library, but on the GPU, such a direct manipulation of data is not possible from the host. The BLAS routine `COPY` can be used to emulate a broadcast operation: By setting the increment argument of the source vector to 0, the first component of the source vector is broadcasted to all components of the target vector. Such an operation has to be performed when setting multiple components of a vector to a specific value as in line 19. Not all BLAS implementations allow this misuse of the `COPY` routine but explicitly exclude an increment argument 0.

5.2. Transformation of block pairs

The transformation of a block pair as described in algorithm 1, lines 8–13, and algorithm 2, lines 8–13 or 16–21, respectively, consists in applying of the Householder transformation to the diagonal and subdiagonal block and the update of this transformation, so that it additionally eliminates the first column of the subdiagonal block. Similarly to the previous section, Householder transformations are performed using vectors instead of matrices.

Algorithm 4 lists the necessary steps for a block pair update. It updates the β-th of b block pairs of A during the ν-th iteration using v_m and returns an adapted transformation v_s for the $(b+1)$-th block pair during the same iteration. Variables with subscript m refer to changes of the diagonal block, while variables with subscript s refer to alternations of the subdiagonal block. Lines 2–4 update the diagonal block $A^{(\nu)}_{\beta,\beta}$. Since the desired transformation v_m is already passed to algorithm 4, it does not have

Algorithm 4 Transformation of a block pair and update of the transformation to avoid fill-in. v_m is the Householder vector to transform the diagonal block and is determined by algorithm 3. v_s is the Householder vector determined during the transformation of the subdiagonal block and will be used in the next block pair transformation $\beta + 1$. Variables with subscript m refer to changes of the diagonal block, while variables with subscript s refer to alternations of the subdiagonal block.

```
1: procedure PROCESS_BLOCK(in/out: $A$, in: $\nu$, in: $\beta$, in: $b$, in: $v_m$, out: $v_s$)
2:     $x_m \leftarrow A^{(\nu)}_{\beta,\beta} \cdot v_m$
3:     $w_m \leftarrow x_m - \frac{1}{2}(v_m^T x_m) \cdot v_m$
4:     $A^{(\nu)}_{\beta,\beta} \leftarrow A^{(\nu)}_{\beta,\beta} - v_m w_m^T - w_m v_m^T$          ▷ Symmetric rank-2 update
5:     if $\beta < b$ then          ▷ Transformation of subdiagonal block
6:         $x_s \leftarrow A^{(\nu)}_{\beta+1,\beta} \cdot v_m$
7:         $h_s \leftarrow A^{(\nu)}_{\beta+1,\beta}(1:d,1) - v_{m1} \cdot x_s$          ▷ First column of $A^{(\nu)}_{\beta+1,\beta} \cdot Q_\beta$
8:         Determine Householder vector $v_s$ with length $||v_s||_2 = \sqrt{2}$ such
           that $(I - v_s v_s^T) \cdot h_s = (*, 0, \ldots, 0)^T$
9:         $z_s^T \leftarrow v_s^T \cdot A^{(\nu)}_{\beta+1,\beta}$
10:        $w_s \leftarrow z_s - (v_s^T x_s) \cdot v_m$
11:        $A^{(\nu)}_{\beta+1,\beta} \leftarrow A^{(\nu)}_{\beta+1,\beta} - x_s v_m^T - v_s w_s^T$          ▷ General rank-2 update
```

to be identified separately. If there is a subdiagonal block (line 5), it is updated by the original transformation v_m (line 6). The first column of the updated subdiagonal block has to be eliminated to avoid fill-in during the next iterations which is assured by lines 7 and 8. Finally, the subdiagonal block is updated, eventually eliminating its first column and considering the original transformation (line 11). This update v_s is performed via a rank-2 update and the necessary temporary vectors z_s and w_s are computed in lines 9 and 10.

```
void process_block(A, n, d, b, nu, beta, v_m, v_s)
{
    // Column/row index of upper left element of diagonal block
    idx = nu + beta * d + 1;

    // x_m = A_beta,beta * v_m
    SYMV(LO, d, ONE, A[idx][idx], n, v_m, 1, ZERO, x_m, 1);
    // w_m = x_m - 0.5 * (v^T * x_m) * v_m
    dot = DOT(d, v_m, 1, x_m, 1);
    dot *= -0.5;
    COPY(d, x_m, 1, w_m, 1);
    AXPY(d, dot, v_m, 1, w_m, 1);
    // A_beta,beta = A_beta,beta - v_m * w_m^T - w_m * v_m^T
    SYR2(LO, d, MINUS, v_m, 1, w_m, 1, A[idx][idx], n);

    if (beta < b - 1)
17      {
18          // x_s = A_beta+1,beta * v_m
19          GEMV(R, d, d, ONE, A[idx+d][idx], n, v_m, 1, ZERO, x_s, 1)
20          // h_s = A_beta+1,beta(1:d,1) - v_m(0) * x_s
21          COPY(d, A[idx+d][idx], 1, h_s, 1);
22          norm = -v_m[0];
23          AXPY(d, norm, x_s, 1, h_s, 1)
24          // 1 / (h_s(0) + sgn(h_s(0)) * || h_s ||)
25          norm = NRM2(d, h_s, 1);
26          norm = 1.0 / (h_s[0] + sgn(h_s[0]) * norm);
27          // First component of Householder vector is always 1
28          v_s[0] = 1.;
29          // Copy remaining components v_s(2:d) from h_s(2:d)
30          COPY(d - 1, h_s[1], 1, v_s[1], 1);
31          // Scale components v_s(2:m) of Householder vector
32          SCAL(d - 1, norm, v_s[1], 1);
33          // Scale v_s to length sqrt(2)
34          norm = NRM2(d, v_s, 1);
35          norm = M_SQRT2 / norm;
36          SCAL(d, norm, v_s, 1);
37          // z_s^T = v_s^T * A_beta+1,beta
38          GEMV(T, d, d, ONE, A[idx+d][idx], n, v_s, 1, ZERO, z_s, 1)
39          // w_s = z_s - (v_s^T * x_s) * v_m
40          dot = -DOT(d, v_s, 1, x_s, 1);
41          COPY(d, z_s, 1, w_s, 1);
42          AXPY(d, dot, v_m, 1, w_s, 1);
43          // A_beta+1,beta = A_beta+1,beta - x_s*v_m^T - v_s*w_s^T
44          GER(d, d, MINUS, x_s, 1, v_m, 1, A[idx+d][idx], n);
45          GER(d, d, MINUS, v_s, 1, w_s, 1, A[idx+d][idx], n);
46      }
47  }
```

Listing 5.2: Simplified code to determine the Householder vector v and eliminate ν-th column by using BLAS level 1 operations. Variable declaration and memory allocation are neglected. This code works directly on matrix A, stored linearly in memory. The `[][]` notation is used to make the code more readable. Indexing is 1-based for comments and 0-based in code. `MINUS`, `ZERO`, and `ONE` are the scalar constants -1, 0, and 1, respectively, `M_SQRT2` corresponds to $\sqrt{2}$. When calling `GEMV`, `R` and `T` indicates that the matrix should be interpreted in a regular or transposed way, respectively.

The mapping of the linear algebra operations of algorithm 4 to BLAS routines is shown by listing 5.2. Again, scalar instructions (lines 10, 22, 26, 28, and 35) are not mapped to BLAS routines because they are executed faster if directly issued instead of calling a library function. On the GPU, one cannot avoid to even express the scalar instructions with calls to the BLAS implementation or to use self-implemented kernels. Now, also BLAS level 2 routines are utilized besides BLAS level 1 routines. A rank-k update is a matrix sum adding a matrix of rank k to an arbitrary matrix. The rank-k matrix is specified by k outer vector/vector products. Rank-1 and rank-2 updates are BLAS level 2 routines and exist for general and symmetric matrices. In line 4 of algorithm 4, there is a symmetric rank-2 update, defined as

$$A := \alpha \cdot xy^T + \alpha \cdot yx^T + A, \qquad A \in \mathbb{R}^{n\times n}, x, y \in \mathbb{R}^n, \alpha \in \mathbb{R},$$

and implemented by the BLAS routine `SYR2` applied in line 14 of listing 5.2. Furthermore, there is a general rank-2 update in line 11 of algorithm 4. Since not all BLAS implementations used by us offer an implementation of a general rank-2 update, two successive general rank-1 updates defined as

$$A := \alpha \cdot xy^T + A, \qquad A \in \mathbb{R}^{n\times n}, x, y \in \mathbb{R}^n, \alpha \in \mathbb{R},$$

are applied instead in lines 44 and 45 of listing 5.2.

5.3. Pipelining

We trigger dedicated library calls for every block pair update. Thus, to execute the items in the pipeline of a specific step in parallel, multiple kernels have to run concurrently to achieve this level of parallelism. Such a feature is supported by modern GPUs and is called *concurrent kernel execution*. Depending on the usage of CUDA or OpenCL (the same holds for the usage of libraries written in CUDA or OpenCL), either *streams* or *command queues* have to be applied to perform concurrent kernel execution. Both are controlled by the host.

A stream is a sequence of commands that are executed in issue-order. Such commands can be kernel executions or memory operations such as host to device or vice versa copy operations. Whereas the commands within a stream are executed one after another, multiple streams can be executed in parallel, so multiple kernels can be executed in parallel. The most common usage of streams is the hiding of communication by computation: While data is exchanged between host and device, computations can be performed on the GPU by running two streams, one for the communication and one for the computation. For our purposes, we use streams to run multiple kernels offered by BLAS implementations for the GPU in parallel.

Command queues are the equivalent of CUDA steams in OpenCL, but offer some additional features. For example, the commands within a command queue can also be executed out-of-order which is not advantageous for our purposes here. Furthermore, a command can also be assigned to multiple command queues in OpenCL, which is not possible with streams in CUDA. We assign every item of the pipeline (multiple calls to BLAS routines) to a separate stream or command queue, respectively to achieve concurrent kernel execution.

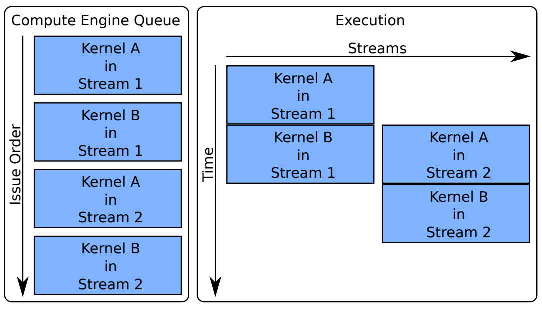
Streams and command queues can be synchronized via events. The maximum number of commands concurrently executed on the GPU is specified by the compute capability. Values vary between 4 and 128 concurrently running streams where 16 and 32 are the most frequent values. We use the number of concurrently executed kernels as one of the most important benchmark criterion in the proceeding chapter 6 because it reflects the capability of the pipelined approach to be parallelized.

Several properties of the GPU hardware have to be considered to achieve concurrent kernel execution in practice. First, the execution of a kernel resides on at least one multiprocessor. It is not possible that two distinct kernel dispatches are concurrently executed on the same multiprocessor but one executed kernel can reside on multiple multiprocessors which are then "blocked" for other kernel executions. Thus, the maximum speed-up is limited, besides other factors explained in chapter 6, by the number of multiprocessors of the GPU. The parallel setup being used when a kernel is issued can be used to guarantee that every kernel execution only occupies one multiprocessor by limiting the number of thread blocks to one. Hence, the parallelism within one kernel is only steered by the number of threads per block. In most cases, the programmer does not have the possibility to control the number of launched blocks per kernel when using libraries for the GPU because this feature is hidden behind the call to the library function.

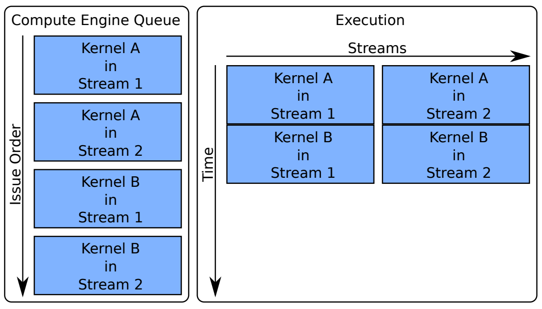
Second, the order of issuing commands is crucial for NVIDIA GPUs. Basically, this can be done in two different ways: *Issuing depth first* means that first all commands assigned to one stream are issued before the commands assigned to another stream. This would be the straightforward strategy resulting from algorithm 4, the transformation of $B_{\beta}^{(\nu)}$ and update of the transformation to avoid fill-in: All BLAS routines required to process a block pair are assigned to one stream before the BLAS routines required by an independent block pair are assigned to the next stream. *Issuing breadth first* is the orthogonal strategy: Commands are assigned to streams in an alternating way. Once a command is assigned to a specific stream, the next issued command is assigned to another stream. On NVIDIA GPUs there are dedicated engines controlling either memory operations or computations. These engines can be seen one level above the warp schedulers. We focus on the compute engines. Typically, there are only one or two compute engines per GPU, depending on the compute capability. Each engine manages a queue to dispatch CUDA commands. This leads to four possible states of a command:

- **Issued**:
  The command is called from the host by the programmer.
- **Dispatched**:
  The command is assigned to GPU resources by the engine from the engine queue.
- **Executed**:
  The command is run on the GPU managed by the warp scheduler(s).
- **Completed**:
  The execution of the command is finished.

According to [195], a CUDA command is dispatched from a specific engine queue if the following three conditions are satisfied:



(a) Issue depth first



(b) Issue breadth first

Figure 5.1.: Different strategies in the order of issuing four commands and assigning them to two streams. The left boxes show the issue and thus dispatch order (condition (c)) of the compute engine queue. The right boxes show the (parallel) execution of kernels using two streams. Subfigure 5.1(a) illustrates issuing depth first. The execution of “kernel B in stream 1” has to wait until “kernel A in stream 1” is completed due to condition (b). Thus, “kernel A in stream 2” also has to wait until “kernel A in stream 1” is completed due to condition (c). This does not hold in subfigure 5.1(b). “Kernel A in stream 2” can be executed immediately because there is no other command assigned to stream 2 (condition (b)) yet and “Kernel A in stream 1” is already dispatched (condition (c)). The layout of this figure is taken from [195].

(a) Resources are available:
An operation originating from a kernel launch requires at least one empty multi-processor to be executed.

(b) Preceding calls in the same stream have been completed:
Only one operation per stream is executed per time.

(c) Preceding calls in the same queue have been dispatched:
The order of issued operations is kept.

The combination of conditions (b) and (c) indicates that issuing breadth first is superior to issuing depth first. Using issuing depth first has to wait until the previous command issued to a certain stream has completed before the next command assigned to the certain stream can be dispatched and, thus, executed due to condition (b). Commands issued to other streams are blocked as depicted by subfigure 5.1(a). Using issuing breadth first always satisfies conditions (b) and (c) as long as hardware resources are available and a stream is not executing a command dispatched long ago in the past. This leads to the maximum possible degree of parallelism for concurrent kernel execution as shown by subfigure 5.1(b).

5.4. Matrix storage format

The system matrix A can be stored very efficiently if its banded and symmetric properties are exploited properly. In this section we present an optimal storage format for A in terms of memory consumption. Furthermore, the proposed storage format enables coalesced memory access on GPUs. During runtime, matrix A is stored continuously as a two-dimensional array on the device for the whole runtime. Updates of A are directly applied to it, so no copy of A is ever made. Since it is distinguished between even and odd steps, no race conditions occur even if there is only one instance of A. The following expression illustrates the transformation steps of A in (5.1) and (5.2) and the final storage layout in memory by (5.3).

$$\begin{pmatrix} a_{1,1} & a_{2,1} & a_{3,1} & \cdots \\ a_{2,1} & a_{2,2} & a_{3,2} & \cdots \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots \\ a_{4,1} & a_{4,2} & a_{4,3} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \in \mathbb{R}^{n \times n} \rightsquigarrow \begin{pmatrix} a_{1,1} & 0 & 0 & \cdots \\ a_{2,1} & a_{2,2} & 0 & \cdots \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots \\ a_{4,1} & a_{4,2} & a_{4,3} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \in \mathbb{R}^{n \times n} \tag{5.1}$$

$$\rightsquigarrow \begin{pmatrix} a_{1,1} & a_{2,2} & a_{3,3} & \cdots \\ a_{2,1} & a_{3,2} & a_{4,3} & \cdots \\ a_{3,1} & a_{4,2} & a_{5,3} & \cdots \\ a_{4,1} & a_{5,2} & a_{6,3} & \cdots \\ \vdots & \vdots & \vdots & \ddots \\ a_{2\cdot d,1} & a_{2\cdot d+1,2} & a_{2\cdot d+2,3} & \cdots \end{pmatrix} \in \mathbb{R}^{2\cdot d \times n} \tag{5.2}$$

$$\Downarrow$$

$$[a_{1,1}, a_{2,1}, \ldots, a_{2\cdot d,1}, a_{2,2}, a_{3,2}, \ldots, a_{2\cdot d+1,2}, a_{3,3}, a_{4,3}, \ldots, a_{2\cdot d+2,3}, \ldots] \tag{5.3}$$

Only the lower triangular part of A is stored as depicted in equation (5.1) due to the symmetry of A. A is a banded matrix, thus, for every column, the right remainder of the matrix could be shifted "one element to the top" as sketched by equation (5.2) leading to a $\mathbb{R}^{d+1\times n}$ matrix. The diagonal elements of A are stored at the first index in each column. This measurement reduces the memory consumption from $\mathcal{O}(n^2)$ to $\mathcal{O}(d \cdot n)$, which saves a factor of n of memory enabling the GPU to store even huge matrices. The updates of the subdiagonal blocks converts them to full blocks instead of upper triangular blocks. Hence, some additional memory has to be spent with using a $\mathbb{R}^{2\cdot d\times n}$ matrix instead of a $\mathbb{R}^{d+1\times n}$ matrix. A depiction of this compression scheme is shown in figure 4.7 in [12] omitted by us for the sake of compactness. BLAS routines still work with the compressed version of A by adapting the argument that specifies the size of the leading dimension (generally denoted by `lda`) by -1 corresponding to the offset saved per column. As depicted by (5.3), A is stored in column-major order because most BLAS implementations expect a column-major order storage scheme. To enable coalesced memory access, we work on the lower triangle of A instead of the upper, i.e. eliminating columns instead of rows as continuously described in the previous sections. Analogously, row-major order could be applied considering right instead of lower subdiagonal blocks. This would even work with BLAS implementations expecting column-major order by adapting the element offsets and the size of the leading dimension of A, but with significantly worse performance due to a lack of coalesced memory access.

During the tridiagonalization process, the Householder vectors ${v_m}_\beta^{(\nu)}$ and ${v_s}_\beta^{(\nu)} (1 \leq \nu \leq n-2, 1 \leq \beta < b-1)$ have to be computed. Since we do not perform any backtransformation steps, we do not store all these Householder vectors but just those required for the current and next elimination iteration.

6. Results

After introducing the SBTH algorithm and its opportunities for parallelization in chapter 4 and discussing its implementation details in chapter 5, we conclude this part with a detailed measurement of our implementation and a corresponding discussion. We use two Toeplitz matrices of different size to carry out our experiments in sections 6.1 and 6.2. Toeplitz matrices are diagonal-constant matrices, thus, all entries of a particular diagonal are the same. According to our experience, the values of the elements of the banded matrix have no influence on runtime. Hence, for a given n and d, it does not make any difference in runtime if a random, Toeplitz, or any other matrix is processed. There are special methods to treat Toeplitz matrices efficiently [24, 54], but we neglect such treatment in this work and interpret the matrices as arbitrary symmetric banded matrices. The smaller matrices use $n = 1000$, thus $A \in \mathbb{R}^{1000\times1000}$, the large ones are 5000×5000 matrices. The bandwidth d is varied from 16 to 256 by factors of 2. Three

manufacturer		NVIDIA			AMD	
GPU		K20x	P100	750 Ti	W8000	HD 8670
host compiler	GCC	-	-	6.2.1	-	6.2.0
	ICC	15.0.2	16.0.3	-	16.0.4	-
device compiler	NVCC	7.5	8.0		-	
BLAS	cuBLAS	7.5	8.0			
	MAGMA	2.2.0				
	clBLAS	-		2.10.0		

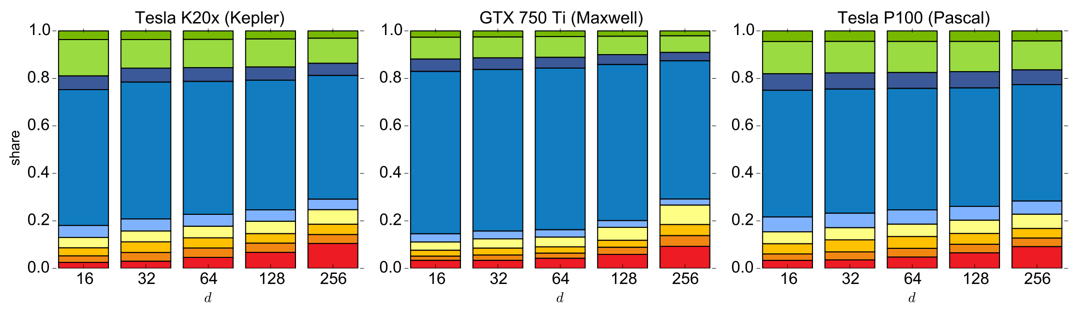
Table 6.1.: Software setup being used to compile and run our SBTH implementation for profiling and benchmarking in this chapter. Numbers in table cells are version numbers of the utilized software. clBLAS does not require a device compiler during compile time and it is not benchmarked on the GPUs carrying out the tests for cuBLAS and MAGMA.

different BLAS implementations for GPUs are used: (a) cuBLAS [179] is a CUDA implementation provided by NVIDIA for their GPUs and is shipped with the CUDA toolkit. (b) MAGMA [5] is available in a CUDA and a OpenCL implementation. It is developed at the University of Tennessee. More background on MAGMA was already given in section 3.1. In this work, we only utilize the CUDA version of MAGMA. MAGMA also supports (partial) execution of some routines on the CPU but we do not exploit this feature in the following. (c) clBLAS [2] is the BLAS part of clMath, an OpenCL library originally developed by AMD and since 2013, provided as open source. Our version of the parallel SBTH algorithm using clBLAS also runs on CPUs and other computing devices supporting OpenCL but we do not consider this capability for benchmarking. Benchmarks are carried out on five different GPUs representing four distinct GPU architectures: The CUDA versions are benchmarked on the Tesla K20x, the GTX 750 Ti, and the Tesla P100, the clBLAS version is run on the FirePro W8000, the Radeon HD 8670, and the GTX 750 Ti. Hardware properties for NVIDIA GPUs are given in table 2.1, table 2.2 lists the characteristics of the corresponding AMD GPUs. All benchmarks use double precision for values. We only provide experimental runtime results. For results reflecting the numerical properties of the SBTH algorithm, we refer to [14]. The software setup (used compilers, version numbers of the BLAS libraries, etc.) is summarized in table 6.1. MAGMA and clBLAS are available as open source. They are compiled with the same compilers used for our SBTH implementation. For all runs delegating work to clBLAS, OpenCL 1.2 is used.

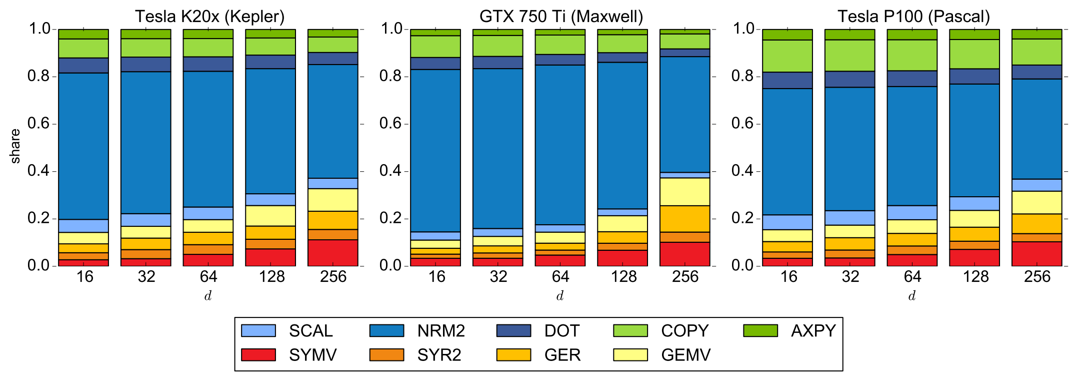
The remainder of this chapter is structured as follows: First, we measure the contribution of each particular BLAS routine to total runtime in section 6.1. Afterwards, we measure the total runtimes of our SBTH versions in section 6.2. Five different GPUs are utilized and matrix size n, bandwidth d, and especially the pipeline length are varied as described above. These total runtimes are used to determine the achieved speed-up when using the pipelined approach. Finally, we do a runtime comparison of our GPU implementations with the distributed memory parallelized ELPA library in section 6.3. Results for the ELPA library are taken from [12].

6.1. Profiling

In this section, we analyze the shares each particular BLAS routine has in total runtime. We neglect the fact that some routines are called more ofter than others (e.g., `GER` and `GEMV` are called approximately twice as often as `SYR2` and `SYMV`) but only consider accumulated runtimes of the particular BLAS routines. Table 6.2 lists the exact numbers how often a particular kernel assigned to a BLAS routine is called by our implementation of the parallel SBTH algorithm. The values in table 6.2 are determined by the NVIDIA command line profiler `nvprof` [182] using the cuBLAS version on the Tesla K20x. These numbers do not have to be the same on other GPUs using other BLAS implementations because depending on library and optimization for a particular GPU, one call to a BLAS routine can trigger multiple kernel calls. Only kernel execution times are considered in this section. Overhead such as kernel invocations and management on the host (loops over ν and β, carrying out breadth first issuing, calculating indices of block pairs, etc.) are neglected. Furthermore, times for pre- and post-processing such as read-in of the matrix or copy operations between host and device are not taken into account. Instead of providing absolute runtimes, normalized values are given, which means the sum of kernel execution times is normalized to 1. This makes it possible to compare different matrix sizes easily, even when the actual runtimes differ much as shown in section 6.2. All experiments in this section use a pipeline length of 1. This does not limit the general validity of the results because the absolute runtimes of the BLAS routines are not affected



(a) $n = 1000$



(b) $n = 5000$

Figure 6.1.: Shares in runtime of BLAS routines in the SBTH algorithm depending on bandwidth d using cuBLAS on three different NVIDIA GPUs for matrix sizes $n = 1000$ (cf. subfigure 6.1(a)) and $n = 5000$ (cf. subfigure 6.1(b)). The overall runtime is normalized to one, thus the height of the bars gives the percentage of the BLAS routines in runtime. BLAS level 1 routines are colored in green and blue, level 2 routines are colored in yellow and red.

by the pipeline length. We focus on the cuBLAS implementation running on the Tesla K20x, the GTX 750 Ti, and the Tesla P100. Shares are similar when using MAGMA and clBLAS, so we do not consider them separately in this section. Figure 6.1 shows the shares of all nine BLAS routines required by the SBTH algorithm. Shares are stacked on top of each other with BLAS level 2 routines on the bottom colored in yellow and red and BLAS level 1 routines on the top colored in green and blue.

The most striking result from all benchmarks is the dominance in runtime of the BLAS level 1 routine `NRM2`. Between 41.6% ($n = 5000$, $d = 128$ on the Tesla K20x) and 68.6% ($n = 5000$, $d = 16$ on the GTX 750 Ti) of total runtime is spent in this routine. This dominance still holds if the runtime of one single `NRM2` execution is compared to the runtime of other BLAS routines executed a single time. Actually, a dominance of the BLAS level 2 routines is expected because at least one of the operands is a matrix, and

n	1000					5000				
d	16	32	64	128	256	16	32	64	128	256
`AXPY`	100,974	69,268	34,500	17,044	8892	253,234	202,496	202,143	151,096	101,852
`COPY`	236,735	163,278	82,158	41,430	21,024	592,114	473,616	473,466	355,083	240,974
`DOT`	67,524	46,506	23,330	11,694	5864	168,956	135,225	135,148	101,210	68,557
`NRM2`	136,468	95,040	48,672	25,392	13,728	339,164	271,524	272,314	205,489	141,108
`SCAL`	169,168	116,756	58,820	29,732	15,158	422,758	338,258	338,387	253,838	172,404
`GEMV`	66,900	45,524	22,340	10,700	4868	168,656	134,741	134,164	99,767	66,602
`GER`	66,900	45,524	22,340	10,700	4868	168,726	134,692	134,118	99,738	66,586
`SYR2`	34,075	23,744	12,160	6344	3430	84,534	67,806	68,025	51,341	35,261
`SYMV`	34,075	23,744	12,160	6344	3430	84,680	67,806	68,052	51,341	35,256

Table 6.2.: Number of calls to particular cuBLAS kernels assigned to BLAS routines of our corresponding implementation of the parallel SBTH algorithm on the Tesla K20x for different matrix sizes n and bandwidths d.

so $\mathcal{O}(d^2)$ data is processed per routine call instead of $\mathcal{O}(d)$ for level 1 routines. However, for such small bandwidths, the times of the operations to compute the Euclidean norm, namely reduction and square root, dominate the time to treat all d^2 elements of a diagonal or subdiagonal block, respectively. This is a phenomenon only observable on the GPU but not on the CPU. As expected, the larger the bandwidth gets, the larger the share of the BLAS level 2 routines becomes. For example, on the GTX 750 Ti using the large $n = 5000$ matrix, the share grows by a factor of $3.37\times$ from 11.1% ($d = 16$) to 37.3% ($d = 256$).

In most cases, the second largest share of the BLAS routines has the `COPY` routine. There are some exceptions such as for $n = 5000$ and $d = 256$ where the share of `GEMV` is the second largest on the GTX 750 Ti. The share of the `COPY` routine varies between 6.3% ($n = 5000$, $d = 256$ on the GTX 750 Ti) and 15.3% ($n = 1000$, $d = 16$ on the Tesla K20x) and includes the runtimes of the `cudaMemset()` function being used by cuBLAS to implement copy operations with increment argument 0. In contrast to “compute” operations such as all other utilized BLAS routines, copy operations such as the `COPY` routine cannot be overlapped with itself by concurrent kernel execution. Here, by copy operations, we do not mean copy operations between host and device but copy operations on the device performed by kernels. This just slightly limits the maximum theoretical speed-up of the pipelined approach because copy operations can still be overlapped with other compute operations.

`GEMV` is the most dominant BLAS level 2 routine. Its share of the total runtime is between 3.5% ($n = 1000$ and $n = 5000$, $d = 16$ on the GTX 750 Ti) and 11.7% ($n = 5000$, $d = 16$ on the GTX 750 Ti) and its share of the accumulated runtime of all BLAS level 2 routines is between 24.8% ($n = 1000$, $d = 256$ on the Tesla K20x) and 33.8% ($n = 5000$, $d = 128$ on the Tesla K20x).

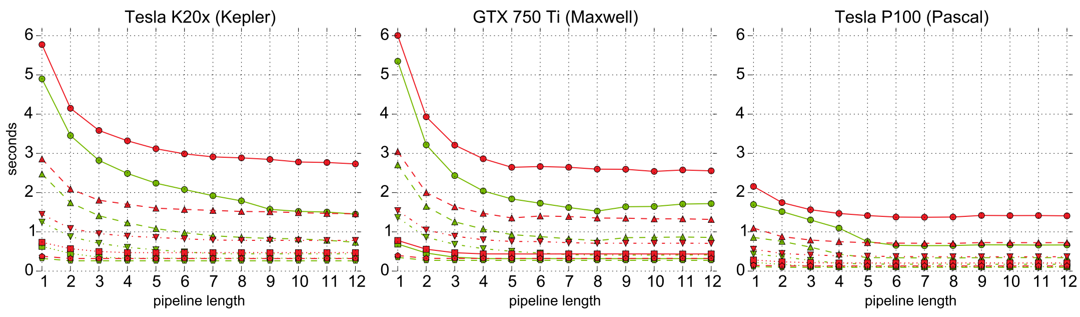
Finally, it can be observed that all ratios of the particular BLAS routines are quite similar for same d but different n and GPU. This leads to the conclusion that all operations, independently of whether it is a BLAS level 1, level 2, or copy routine, perform similarly on the three different generations of NVIDIA GPUs applied in this section and no GPU architecture is significantly better suited for one or the other BLAS routine.

6.2. Scalability of the pipelined approach

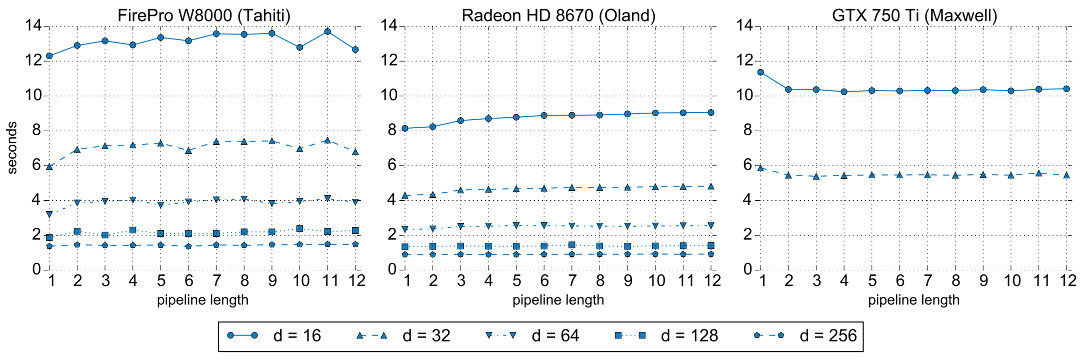
We measure the absolute runtimes of our implementations of the parallel SBTH algorithm to determine the speed-ups achieved by the pipelined approach in this section. These total runtimes include the actual execution times of the kernels plus all overheads such as kernel invocation times and all management on the host but no times for pre-processing such as loading/initializing the matrices in GPU memory and post-processing such as validating the results. The utilized GPUs, BLAS implementations for the GPU, and applied parameters such as matrix size n, bandwidth d, and pipeline length are given in the introduction of this chapter.

Since we are interested in the influence of the pipeline length on the runtime of the parallel SBTH algorithm, we put the pipeline length on the abscissas and the runtime on the ordinates of figures 6.2 to 6.3. Correspondingly, speed-up is assigned to the ordinates of figures 6.4 to 6.5 when investigating the speed-up behavior. The maximum pipeline length considered is 12. According to our experience, longer pipelines only have a minor or even degrading effect on runtime as can already be observed in figures 6.4 to 6.5 for long pipeline lengths. Results stemming from CUDA and OpenCL libraries are plotted in dedicated subfigures throughout this section to provide visual clarity. CUDA results are presented in the top subfigures, OpenCL results in the bottom subfigures. There is a dedicated subplot per GPU in every line. cuBLAS results are colored in green, MAGMA results in red, and clBLAS results in blue. Different bandwidths are drawn as separate lines distinguished by distinct markers. Due to a bug in the OpenCL implementation of NVIDIA for their GPUs, there are only results for $d = 16$ and 32 of the clBLAS version on the GTX 750 Ti. For $d = 64$, 128, and 256, this version simply crashes in multiple BLAS routines with a general error. While figures 6.2 and 6.3 show the absolute runtimes for $n = 1000$ and 5000, figures 6.4 and 6.5 depict the corresponding speed-ups for the two different-sized matrices. Matrix size and bandwidth are kept constant for an individual line, thus, the benchmarks are strong scaling tests.

Before discussing the speed-up behavior, we emphasize important observations regarding the actual runtimes. The performance of the cuBLAS version is superior to the performance of the MAGMA version for all variations of parameters. For BLAS level 1 and level 2 routines, MAGMA simply delegates work to cuBLAS by calling the corresponding cuBLAS functions. MAGMA offers a dedicated implementation only for BLAS level 3 routines. Hence, it is impossible for the MAGMA version to perform better than the cuBLAS version. Instead, additional overhead is introduced, e.g. by delegating function calls or mapping the way MAGMA deals with streams to the way cuBLAS deals with them. This mapping leads to a worse scaling behavior of our version using MAGMA in comparison to the one using cuBLAS. Both versions using the CUDA libraries clearly



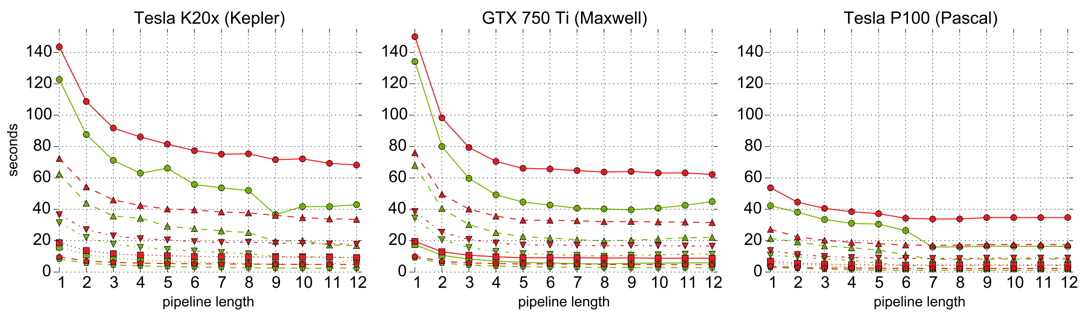
(a) CUDA libraries



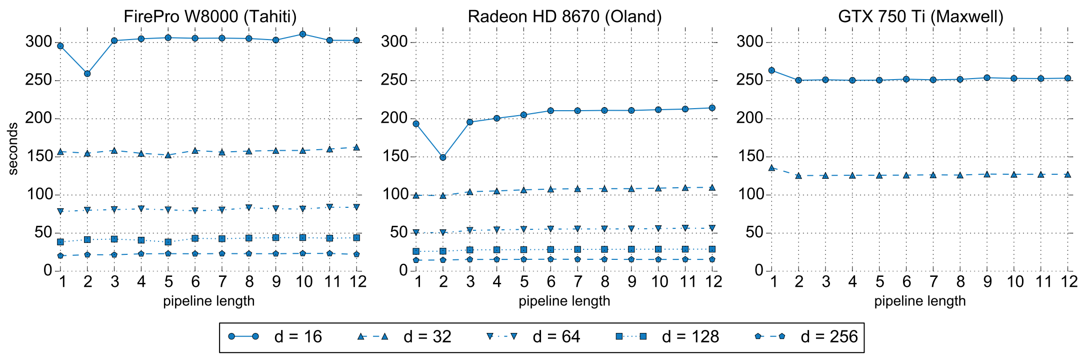
(b) OpenCL library

Figure 6.2.: Runtimes in seconds of the parallel SBTH algorithm on five different GPUs in dependence of pipeline length. Matrix size is $n = 1000$ and matrix bandwidth d varies from 16 to 256, indicated by different lines. Figure 6.2(a) shows the runtimes of cuBLAS (green) and MAGMA (red) on three different NVIDIA GPUs. Figure 6.2(b) shows the runtimes of clBLAS (blue) on two different AMD GPUs and one NVIDIA GPU.

outperform the version relying on clBLAS. This difference does not originate from an inequality in performance of the various GPUs as can be seen when drawing a direct performance comparison on the GTX 750 Ti. A performance penalty between 1.96× ($n = 5000$, $d = 16$) and 2.17× ($n = 1000$, $d = 32$) can be measured when comparing the cuBLAS and clBLAS versions for a pipeline length of 1 on the Maxwell GPU. There are two major reason for this performance gap: First, in contrast to cuBLAS, the clBLAS version is much less or not at all optimized for particular GPU architectures, especially for BLAS level 1 and level 2 routines. Second, a considerable amount of runtime is spent with kernel invocations and management on the host. The number of kernel calls is quite large, as given in table 6.2 for different n and d. In addition, such a kernel invocation is relatively expensive in comparison to the actual work the BLAS kernel is performing. For our SBTH algorithm implementation, we observed that the overhead to invoke an



(a) CUDA libraries



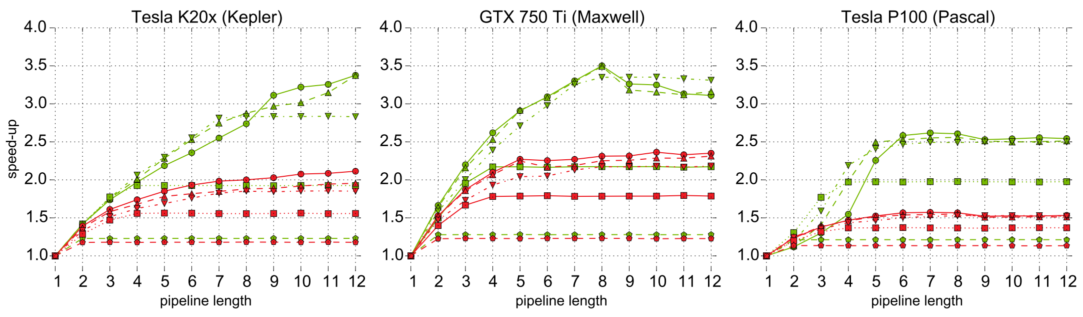
(b) OpenCL library

Figure 6.3.: Runtimes in seconds of the parallel SBTH algorithm on five different GPUs in dependence of pipeline length. Matrix size is $n = 5000$ and matrix bandwidth d varies from 16 to 256, indicated by different lines. Figure 6.3(a) shows the runtimes of cuBLAS (green) and MAGMA (red) on three different NVIDIA GPUs. Figure 6.3(b) shows the runtimes of clBLAS (blue) on two different AMD GPUs and one NVIDIA GPU.

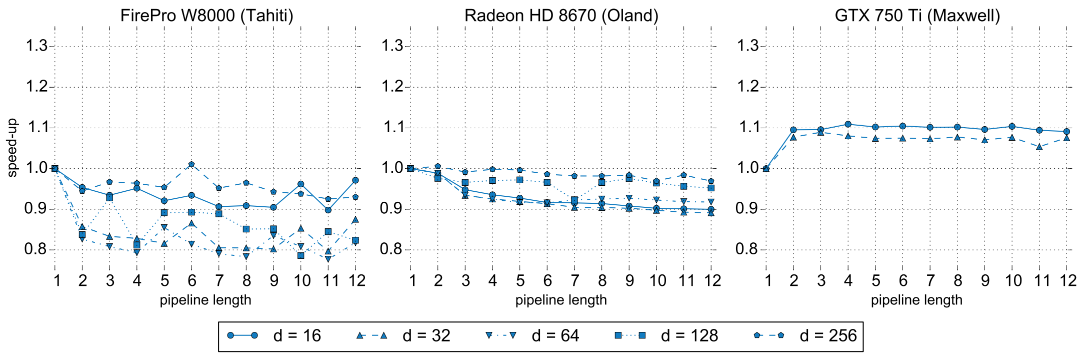
OpelCL kernel from clBLAS is much bigger than the overhead to invoke a CUDA kernel from cuBLAS. Thus, the difference in performance between the clBLAS and cuBLAS version is significant.

According to [128] and [12], the SBTH algorithm requires $\mathcal{O}(n^2 d)$ operations in total. Thus, the expected runtimes tend to grow quadratically in n. This assumption is testified empirically when comparing the runtimes for $n = 1000$ and $n = 5000$.

A surprising discovery is the influence of the matrix bandwidth d on the runtime. The wider the band gets, the shorter the runtime. Since broader bands should lead to higher computational costs to process a block pair, we would expect the opposite behavior. However, a larger d leads to a reduction in the number of block pairs to process per iteration and, thus, of total block pairs to process. Moreover, since only BLAS level 1 and 2 routines are applied and the bandwidth stays relatively small, the influence of d



(a) CUDA libraries

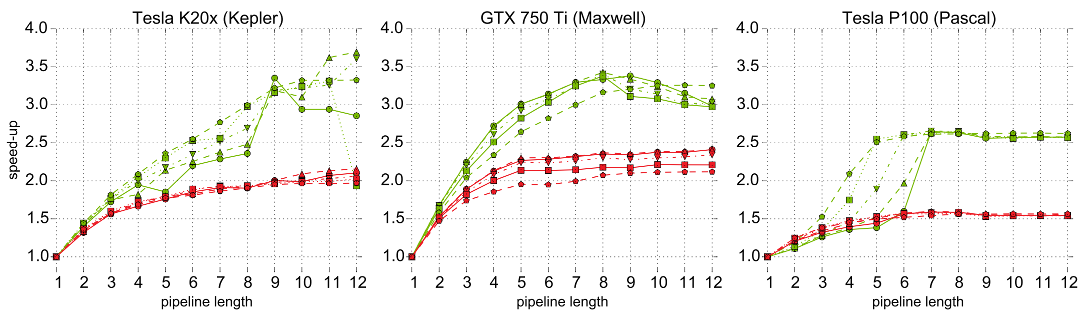


(b) OpenCL library

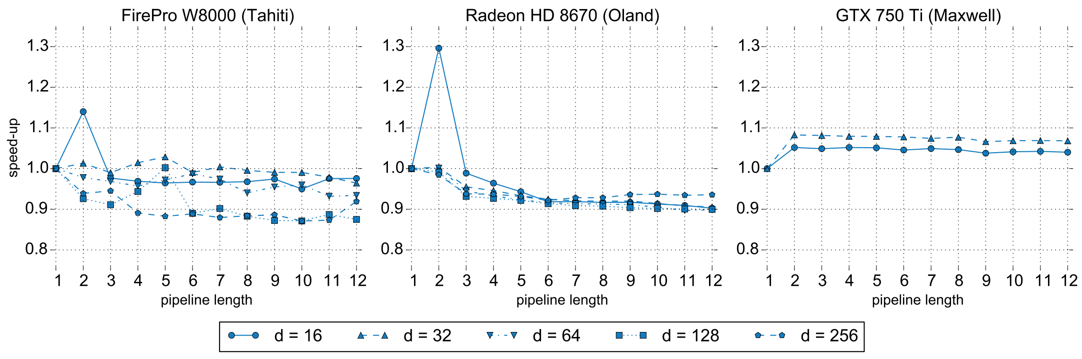
Figure 6.4.: Speed-ups of the parallel SBTH algorithm on five different GPUs in dependence of pipeline length for $n = 1000$. Color coding, line captions, and markers are equal to figure 6.2.

on the computational costs per BLAS routine is small.

There is not much difference in runtimes when comparing different GPUs using the same parameters n, d, and pipeline length with each other. They vary between 1.5% ($n = 5000$, $d = 256$, pipeline length 4) and 48.5% ($n = 5000$, $d = 16$, pipeline length 5) for cuBLAS, between 0.2% ($n = 1000$, $d = 256$, pipeline length 4) and 19.1% ($n = 5000$, $d = 16$, pipeline length 5) for MAGMA, and between 1.3% ($n = 1000$, $d = 32$, pipeline length 1) and 73.6% ($n = 5000$, $d = 16$, pipeline length 2) for clBLAS. The only exception is the Tesla P100 showing significantly better performance than the Tesla K20x and the GTX 750 Ti. This does not reflect the ratio of theoretical performance parameters of the GPUs given in tables 2.1 and 2.2. On the one hand, overhead times other than the actual runtimes of the kernels have a considerable share of the total runtime. On the other hand, examining the runtime of a single kernel execution, the number of required calculations and amount of transferred data is quite low. Therefore, high-performance GPUs such as the Tesla K20x or the FirePro W8000 do not show clearly improved runtimes than low performance GPUs such as the GTX 750 Ti and the Radeon HD



(a) CUDA libraries



(b) OpenCL library

Figure 6.5.: Speed-ups of the parallel SBTH algorithm on five different GPUs in dependence of pipeline length for $n = 5000$. Color coding, line captions, and markers are equal to figure 6.3.

8670.

The speed-up behavior is best for cuBLAS. While there is still a considerable speed-up when using MAGMA, no significant speed-up can be observed when using clBLAS. This holds for all three GPUs being used to benchmark the clBLAS version. Hence, the usage of OpenCL command queues leads to no or only negligible concurrent kernel execution. Furthermore, the usage of multiple command queues can even decrease performance due to additional overhead, as illustrated by both AMD GPUs for $n = 1000$. In general, the speed-up for the cuBLAS and MAGMA version is higher the smaller the matrix bandwidth with maximum speed-up of 3.70× for cuBLAS (Tesla K20x, $n = 1000$, $d = 32$, pipeline length 12) and 2.41× for MAGMA (GTX 750 Ti, $n = 5000$, $d = 16$, pipeline length 12). The smaller the bandwidth, the more block pairs are potentially available for parallel execution. Eventually, the pipelined approach leads to a clear acceleration of the SBTH algorithm on the GPU but this improvement is smaller than the number of GPU multiprocessors or block pairs b, being indicators for the maximum theoretical speed-up. There are multiple reasons for this disparity: First, the number of block pairs that could be processed in parallel becomes smaller the larger ν becomes, so the degree of potential parallelism becomes smaller near the end of the transformation, as discussed in section 4.3. Second, not all BLAS kernels reside on one single multiprocessor, which is required for a high degree of concurrent kernel execution as claimed in section 5.3. Third, there is a considerably large portion of work that cannot be parallelized such as the kernel invocations and the management on the host. According to Amdahl's law [8], this limits the maximum speed-up. Finally, there are BLAS routines impossible to be simultaneously executed by concurrent kernel execution such as the `COPY` routine as indicated in section 6.1.

6.3. Comparison with ELPA

Until now, we have compared our SBTH implementation using different BLAS implementations for GPUs and varied parameters such as n, d, and pipeline length. In this section, we compare our implementation with the banded to tridiagonal matrix reduction method of ELPA [149] also using the SBTH algorithm. ELPA is a distributed memory parallelized library for the CPU supporting the whole two sub-tasks tridiagonalization and backtransformation depicted in figure II.1. Recently [150], kernels for GPUs were introduced in ELPA but they concentrate on compute-intensive parts such as transformation from full matrix to banded matrix and backtransformation.

Table 6.3 lists the runtimes of our parallel SBTH implementation in the upper part and the runtimes of ELPA taken from figure 4.13 in [12] in the lower part. Analogously to the results in the section 6.2, runtimes of our implementation contain all overhead times but no times for pre- and post-processing. A common pipeline length of 8 is used for all runs of our implementation. The ELPA experiments were conducted on the Power6 cluster of the Max Planck Computing & Data Facility[1] (MPCDF) (formerly Rechenzentrum Garching (RZG)), the same supercomputing center where the Hydra cluster (cf. table 2.5) is located. Each node of the cluster is equipped with 16 IBM Power6 CPUs, each CPU consisting of two cores. One MPI process is started per core. In the end of 2012, the cluster was shut down but the results are still suited for a comparison. The test matrix has size $n = 79{,}872$ and bandwidth $d = 576$. ELPA can do the transformation in a 1-step or a 2-step tridiagonalization (not to be mistaken with the two sub-tasks "to banded matrix" and "to tridiagonal matrix"): The 1-step tridiagonalization directly transforms the input matrix from bandwidth 576 to bandwidth 1. Our GPU implementation follows this approach. The 2-step tridiagonalization first transforms the matrix to an intermediate bandwidth before eventually bringing it to tridiagonal shape. ELPA results use an intermediate bandwidth of 64. Due to a higher degree of parallelism, the 2-step tridiagonalization performs much better on the GPU, especially for high core numbers, but not on the GPU. Hence, we compare the tridiagonalization on the GPU with the 2-step tridiagonalization on the CPU.

The clBLAS version on the FirePro W8000 shows comparable runtimes to ELPA on 32 cores, thus, a whole node of the Power6 cluster. Since our clBLAS version does not

[1] `https://www.mpcdf.mpg.de/`

library	GPU	runtime in seconds
cuBLAS	Tesla K20x	96.5
	GTX 750 Ti	104.3
	Tesla P100	41.1
MAGMA	Tesla K20x	166.8
	GTX 750 Ti	162.3
	Tesla P100	77.1
clBLAS	Radeon HD 8670	400.0
	FirePro W8000	262.5
number of steps	**number of CPU cores**	**runtime in seconds**
1	32	1431.0
	64	683.8
	128	382.1
	256	345.7
2	32	261.5
	64	139.4
	128	86.1
	256	72.1

Table 6.3.: Runtimes for a n = 79,872, d = 576 problem. The top part of this table gives the runtimes of our GPU implementations using cuBLAS, MAGMA, and clBLAS. Five different GPUs are utilized. The pipeline length is 8. The bottom part gives the runtimes of the corresponding module of ELPA running on a Power6 cluster using 32, 64, 128, and 256 cores.

achieve any significant speed-up when applying command queues, the runtimes would still be close if a pipeline length of 1 would be used. Increasing the number of utilized CPU cores, the runtime of the MAGMA version on the Tesla K20x and the GTX 750 Ti does not differ very much to ELPA on 64 cores. Hence, this combination of BLAS implementation and GPU is competitive against two nodes of the Power6 cluster. As shown in the previous section 6.2, the cuBLAS version performs best enabling the Tesla K20x to have just 12.1% worse performance than ELPA on 128 cores. Finally, the Tesla P100 running the cuBLAS version can even outperform ELPA on 256 cores taking only 57% of its runtime.

The results show that our SBTH implementation is coequal to the corresponding module of a state-of-the-art eigenvalue solver in terms of runtime. Both utilize the SBTH algorithm but are implemented on different computing devices. It is hard to draw a comparison between the two codes because ours runs on GPUs while ELPA is optimized for distributed memory clusters and the results for comparison were taken more than four years ago. However, the Tesla K20x was already available at that time and shows only slightly worse performance than a setup of four nodes of the Power6 cluster. The recently released Tesla P100 is even competitive against an eight nodes setup. Two final aspects should be considered in this comparison: First, the scalability of ELPA breaks down for larger core numbers, beginning from 256 cores. Assuming scalability even at this level of parallelism, ELPA would outperform the Tesla P100. Second, a distributed memory parallelized implementation can exploit much more memory than is available on a single GPU. The matrix processed in this section already requires about 700MBytes of memory, even with the storage scheme of section 5.4. Auckenthaler can deal with much larger matrices in [12] such as rp1088 (n = 1,088,092, d = 2964), rt1379 (n = 1,379,917, d = 4157), and rc1965 (n = 1,965,206, d = 5427), matrices from the field of network analysis, all being too large for one single GPU.

Concluding part II

We present the SBTH algorithm in chapter 4 and how to implement it on single GPUs in chapter 5. By delegating operations to BLAS routines, a significant speed-up can be achieved using concurrent kernel execution. As demonstrated in chapter 6, this leads to a competitive GPU implementation of the `SBTRD` routine, an important step in the determination of eigenvalues.

Although the transformation process of a banded to a tridiagonal matrix using the SBTH algorithm is memory-bound (it only applies BLAS level 1 and level 2 routines), the use of GPUs is beneficial. The parallel SBTH algorithm (cf. section 4.3) introduces an additional level of parallelism, namely a pipelined approach where the tasks in the pipeline can be executed in parallel. This structure can be mapped to the concurrent kernel execution feature (cf. section 5.3) of GPUs leading to an acceleration of up to 3.70×. An even higher speed-up is limited by the choice of the BLAS implementation for the GPU, the programming platform, and overheads originating from the host-controlled approach we select. Such overheads have a serious share of the total runtime (cf. section 6.1). Delegating work to BLAS routines prohibits improvements such as kernel fusion and makes enhancements such as caching more complicated. Furthermore, including already existing BLAS implementations for the GPU may be convenient but hides many further opportunities for optimization, such as setting the parallel setup manually. Moving the program control from the host to the device could significantly accelerate the execution for the cost of flexibility and simplicity. The same holds for writing custom, hardware-aware kernels. However, our shared memory implementation for GPUs offers similar performance to the corresponding module of the ELPA library (cf. section 6.3) running on distributed memory systems and also relying on the SBTH algorithm. This performance is not solely driven by the pipelined approach of the parallel SBTH algorithm but also by the parallel execution of the particular BLAS routines and the high memory bandwidth of GPUs. To our knowledge, there is no alternative GPU implementation of the SBTH algorithm in special or of the `SBTRD` routine in general, thus, we are limited to comparisons with CPU implementations.

Depending on the utilized GPU and library, single GPUs can show equivalent performance or even outperform multi-node distributed memory parallel systems. They are powerful shared memory computing devices, which make sophisticated parallel implementations using message passing obsolete, even if the problem is not compute-bound.

So far, we deal with only one GPU. The SBTH algorithm can be extended to multiple GPUs by realizing the distributed memory parallelization approach used in the original paper [128]. On the one hand, this would enable the processing of larger matrices such as rp1088, rt1379, and rc1965 because more memory is available. On the other hand, already the usage of single GPU provides a high degree of computational power. In the following parts III and IV, multiple GPUs are exploited simultaneously.

Part III.

Multiple levels of parallelism to solve random ordinary differential equations

Mathematical models considering random input are of increasing interest. Fields such as uncertainty quantification (UQ) [222], the science of quantitative characterization and reduction of uncertainties, show that such models presently gain much attention in current research. One further class of such models are random ordinary differential equations (RODEs), an alternative modeling approach to well-known stochastic ordinary differential equation (SODE) systems, especially in the context of time-dependent problems.

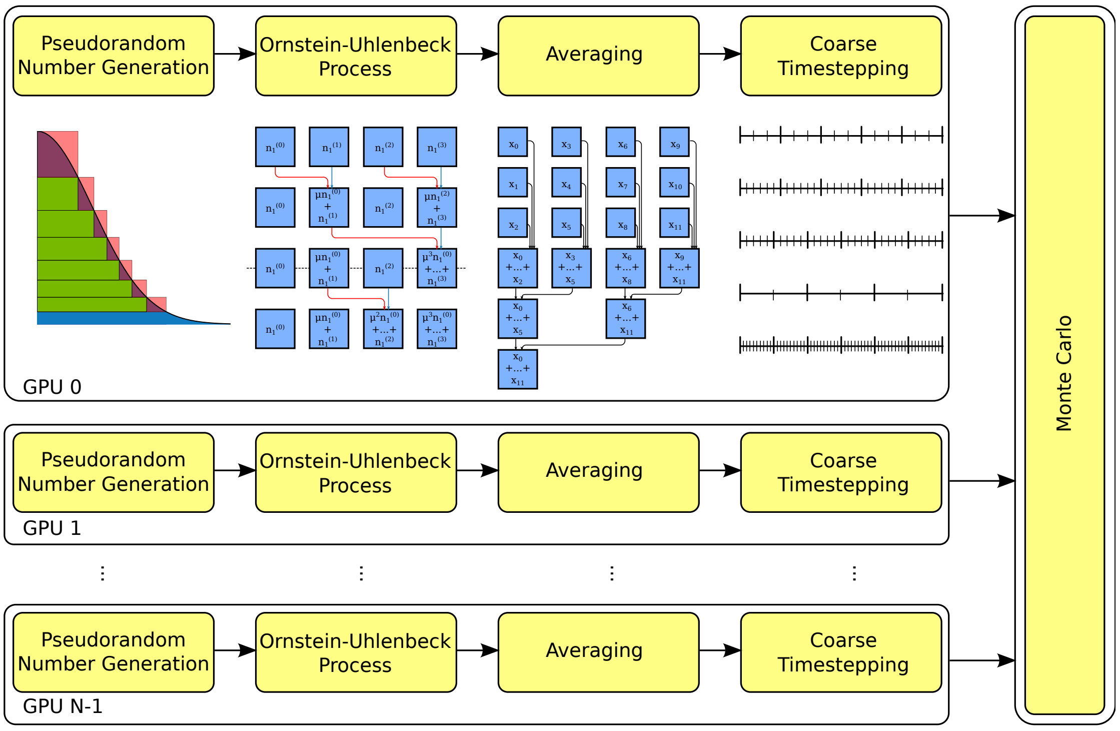


Figure III.1.: This figure illustrates how the four building blocks — generation of random numbers satisfying normal distribution, realization of the OU process, averaging of the elements of the OU process, and the coarse timestepping — interact with each other. On every GPU, one realization of the OU process is computed using different sequences of normal random numbers. In the end, all solutions from the different GPUs based on different realizations of the OU process are merged in a Monte Carlo-like manner. This figure and caption is taken from our contribution [199].

The numerical treatment of RODEs is computationally challenging: We follow a Monte Carlo approach to tackle RODEs by alternatively solving a massive amount of deterministic ordinary differential equations (ODEs). In addition, the numerical solvers applied to each deterministic ODE require a very fine sub-sampling to achieve a reasonable order of convergence. Every sub-sampling step is associated with the realization of a normally distributed random variable making the whole solution process computationally extremely demanding. However, this kind of solution offers multiple ways for parallelization making it very interesting for GPU-equipped clusters. We map the various possibilities for parallelization on the multiple levels of parallelism of GPU clusters, namely two levels of parallelism per GPU and an extra level of parallelism by applying numerous GPUs in parallel.

This endeavor leads to a workflow consisting of four consecutive building blocks to solve arbitrary RODEs. The idea of four building blocks has been first presented in our contribution [199] being the first attempt to generalize the solution process of RODEs. Each building block represents a major algorithmic and computational task. Figure III.1 depicts the interaction between the four building blocks to solve an RODE for one particular realization of the underlying stochastic process and the Monte Carlo approach for a variety of realizations. While the first building block deals with the generation of normally distributed random numbers, building block two consumes these numbers to realize different paths of the Ornstein-Uhlenbeck (OU) process, the elemental stochastic process of RODEs. Most of the accomplishments in the context of random number generation on GPUs has been published by us in [197, 198] and we follow the structure of these two publications. One of the most significant achievements is the parallelization of the OU process. The third building block averages large sub-sequences of an OU process path eventually plugged in the coarse timestepping solver representing the fourth building block. Building blocks one to three are not specific for a particular RODE, not even limited to RODEs in general, but are also relevant for other fields. Only the final building block has to be adapted to a specific application. So our implementation of the first three building blocks can also be utilized in a library-like way. Thus, this work is relevant not only in the context of RODEs but also for other fields or approaches that rely on the efficient generation of normally distributed random numbers or the efficient solution of the popular OU process. All four building blocks are parallelized on the GPU using up to two levels of parallelism being mapped to the two levels of parallelism of a GPU. Every GPU performs the four building blocks for a distinct realization of the OU process. To obtain the overall solution, i.e. the solution of the RODE, the expected value of the path-wise solutions from the particular GPUs has to be determined in a Monte Carlo-like manner. This is the only point where global communication is required. Until this point, the GPUs can run totally independent from each other. Basing on this property, excellent values for parallel speed-up and, thus, parallel efficiency are expected and achieved because the workload per GPU is constant and equal during the entire runtime. Hence, the actual challenge lies in the parallelization of the building blocks.

Extra levels of parallelism could be introduced, e.g. by utilizing parallelization in time [21, 81]. However, we restrict ourselves to the above-mentioned three levels of parallelization, thus, parallelization in time is not part of this thesis. There are also efforts to solve stochastic differential equations (SDEs) on GPUs [112, 66]. In this thesis, we limit ourselves to the treatment of RODEs.

RODEs are the central mathematical object of this subsequent part. They are a special type of general differential equations running through this thesis as a golden thread. Additionally, this part covers further aspects of GPU programming. Due to the intrinsic multi-level parallelism, solving RODEs simultaneously on multiple GPUs makes sense. For all four building blocks, custom GPU kernels are developed and analyzed. They are either latency- or memory-bound. However, GPUs fit very well to the computational tasks as shown in the building block's specific chapters 7 to 11. Accordingly, we not just present an HPC implementation to solve RODEs but introduce the first attempt to solve RODEs in a reasonable scale. This allows to derive insight for real-world problems as shown in section 12.4. Computations are only carried out on the GPUs. The computational capacities of the CPUs in the corresponding GPU cluster are not considered and communication between GPUs is realized via MPI.

The notation from previous part II does not apply to the following part III. Some of the identifiers are reused but with a different meaning. All kernels are written in CUDA, thus, all analysis and benchmarking is carried out on NVIDIA GPUs. However, none of the algorithmical aspects is limited to CUDA or NVIDIA hardware but can also be adopted to other GPUs in particular and parallel computing devices in general. For benchmarking of the random number generation, the OU process, and the averaging, the Tesla M2090, the Tesla K40m, and the GTX 750 Ti are used. Every GPU represents another GPU architecture (Fermi, Kepler, and Maxwell) and the potential performance ranges from mid-range consumer hardware (GTX 750 Ti) to high-end professional products (M2090 and Tesla K40m). Incorporating the Monte Carlo approach, our RODE solver is tested in its entirety on the three GPU clusters JuDGE, Hydra, and TSUBAME2.5. Technical details for the three GPUs are listed in table 2.1 and for the three clusters in table 2.5. The benchmarks of the OU process, the averaging, and lastly the entire RODE solver are carried out in single and double precision. Just the benchmarks of the random number generation are restricted to single precision.

The remainder of this part is structured as follows: Before discussing the four building blocks in detail, we provide an introduction to RODEs in chapter 7. This includes a conceptional discussion of RODEs, their mathematical relation to SODEs, and a listing of numerical schemes to reasonably solve RODEs. The four building blocks derive from the numerics for RODEs. Throughout this work, we apply the Kanai-Tajimi (KT) earthquake model as an example RODE also introduced in chapter 7. Afterwards, each of the four building blocks is assigned to a dedicated chapter beginning with the generation of normally distributed pseudorandom numbers in chapter 8. The parallelization of the OU process is demonstrated in chapter 9, and chapter 10 deals with various methods to average sequences of values essential for the RODE solvers. Since the first three building blocks are not only relevant in the context of RODEs but can be used as separate modules for other applications, they are individually benchmarked in the corresponding chapter. Thus, chapters 8 to 10 are concluded each by a separate results section evaluating the performance on single GPUs. This does not hold for the final building block chapter 11 dealing with the application of the actual numerical solver because it is application-specific. Finally, chapter 12 concludes this part by evaluating the interplay of all four building blocks, the Monte Carlo approach, its scalability on three GPU clusters, and its statistical properties.

7. Random ordinary differential equations

In this chapter, we first review the concept of RODEs, their relation to UQ, and their connection to the more prominent SODEs in section 7.1. Afterwards, we introduce the KT earthquake model for ground motion under earthquake excitations in section 7.2. The KT earthquake model is the RODE application being used in this thesis for demonstration. Finally, we present several low- and high-order numerical schemes essential to deal with RODEs in section 7.3.

This chapter does not contain any new concepts or ideas in the context of RODEs. Instead, we recapture work from various contributors, being cited at the corresponding positions, to build a solid foundation for the subsequent chapters of part III. For the sake of a compact presentation, we only focus on aspects relevant for our multi-GPU implementation to solve RODEs and the relation of the mathematical foundations to the four building blocks. Further material, for example a detailed mathematical discussion, is provided in [51, 173]. We use the notation from [173, 172].

7.1. Random & stochastic ordinary differential equations

In UQ, a number of input parameters are represented by a random distribution. A particular value combination of the input parameters leads to a corresponding smooth, distinct solution, similar to the deterministic case [222]. In contrast to such UQ problems, a variety of non-smooth problems with stochastic input in the form of stochastic processes exist. Such problems are typically formulated as SODEs or stochastic partial differential equations (SPDEs). The solutions of such SODEs and SPDEs are again non-smooth stochastic processes.

RODEs are less prominent than the well-known, white noise driven, SODEs [114] but they offer some nice and conveniant features: They are immediately applicable to additive white noise excited systems, naturally model capacity of real processes, and, most important, provide a conceptually easier formalism. In addition, there are new numerical methods with high convergence rates for RODEs discussed in more detail in section 7.3. Hence, RODEs represent an interesting alternative for modeling random behavior in real-world problems.

An SODE can be disassembled in a stochastic, white-noise driven part, called *diffusion*, and a deterministic part, called *drift*:

$$\mathrm{d}X_t = \underbrace{a(X_t, t)}_{\text{drift (deterministic)}} \mathrm{d}t + \underbrace{b(X_t, t)}_{\text{diffusion (stochastic)}} \mathrm{d}W_t$$

with W_t the standard Wiener process and a and b sufficiently well-defined, progressively measurable functions with respect to the filtration generated by W_t. The solution X_t represents a stochastic Markovian diffusion process that is as well adapted to the filtrations of the Wiener process. See [211] for more details. Solving SODEs is challenging since it requires considerable mathematical effort such as Itô's calculus [118, 201]. Itô's calculus extends the methods of calculus to stochastic processes and typically results in low-order numerical schemes.

An alternative approach are RODEs. Let $(\Omega, \mathcal{A}, \mathbb{P})$ be a probability space, $I = [t_0, T] \subseteq \mathbb{R}_0^+$ an interval, and $X : I \times \Omega \to \mathbb{R}^d$ an $\mathbb{R}^d$-valued stochastic process with continuous sample paths. Then, an RODE is defined as

$$\frac{\mathrm{d}X_t}{\mathrm{d}t} = f(X_t(\cdot), t, \omega), \qquad X_t(\cdot) \in \mathbb{R}^d \tag{7.1}$$

with a continuous function $f : \mathbb{R}^d \times I \times \Omega \to \mathbb{R}^d$. X_t is a stochastic process on the interval I and is called *path-wise solution* of RODE (7.1). For (almost) every fixed realization $\omega \in \Omega$ of the driving stochastic process X_t, the RODE (7.1) turns into a deterministic, non-autonomous ODE

$$\dot{x} = \frac{\mathrm{d}x}{\mathrm{d}t} = F_\omega(x, t), \qquad x := X_t(\omega) \in \mathbb{R}^d. \tag{7.2}$$

To cover the statistics of the solution of RODE (7.1), a lot of solutions of the deterministic ODE (7.2) in the sense of, e.g., Monte-Carlo with different $\omega \in \Omega$ are necessary. This set of different realizations introduces one level of parallelism in our RODE solver implementation because for every particular ω, the corresponding ODE can be treated totally independently.

An SODE allows a path-wise solution if, for example, the drift a and diffusion b are globally Lipschitz-continuous functions. If an SODE is finite-dimensional and allows a path-wise solution, the SODE can be transformed into an RODE by using the Doss-Sussmann/Imkeller-Schmalfuss (DSIS) correspondence explained in [225, 110, 114]. This results in a change of the driving stochastic process from the white noise of the SODE into, typically, a stationary OU process for the RODE, cf. [110, 109]. The DSIS correspondence also allows a transformation in the opposite direction, namely from an RODE to an SODE. Simple examples for both directions are given in [114]. To demonstrate the DSIS correspondence, we use the KT earthquake model in the next section.

7.2. The Kanai-Tajimi earthquake model

Throughout this part of the thesis, we use a model proposed by Kanai [116, 117] and Tajimi [227] which is a simple but still decently sophisticated approach to obtain ground excitations induced by earthquakes. It describes the earth's crust as one layer represented by a damped mass-spring-like system. The mass-spring is excited by an white-noise driven event propagating from the bottom to the top of the system. This is a serious simplification of the geology but due to Snell's law of refraction, the direction of propagation of seismic waves is almost vertically upward if the source of an earthquake is reasonable deep [140]. The KT model does not model a single event in time starting the earthquake but the white-noise driven source continuously excites the system. For high frequencies, the KT model displays the characteristic properties of earthquakes quite well, whereas for low frequencies inaccuracies occur [187]. The solution of the KT model is an acceleration $\ddot{u}_g(t)$ affecting, for example, ground-bound structures such as buildings. Thus, $\dot{u}_g(t)$ gives the velocity of the ground for a certain point in time t and $u_g(t)$ the corresponding position of the point inducing the excitation. In this section, we introduce the KT model in its SODE form and show how to transform it to an RODE via the DSIS. Along the way, we define the OU process and offer a timestepping method to numerically solve it.

The SODE formulation of the KT model with ground acceleration $\ddot{u}_g(t)$ has the following form:

$$\ddot{u}_g = \ddot{x}_g + \xi_t = -2\zeta_g\omega_g\dot{x}_g - \omega_g^2 x_g,$$

where ζ_g and ω_g are parameters representing the geological conditions of the ground. Housner [106] suggests $\zeta_g = 0.64$ and $\omega_g = 15.56\frac{rad}{s}$ representing stable and solid ground conditions. We apply these values in our implementation. ξ_t is a stochastic process describing zero mean Gaussian white noise. Langtangen discusses a certain stochastic oscillator model in [130]. x_g represents the solution of such a stochastic oscillator driven by ξ_t. Solving the KT SODE for ξ_t gives

$$-\xi_t = \ddot{x}_g + 2\zeta_g\omega_g\dot{x}_g + \omega_g^2 x_g, \qquad x_g(0) = \dot{x}_g(0) = 0. \tag{7.3}$$

According to previous section 7.1, W_t denotes a Wiener process. Using W_t, the OU process O_t can be expressed via the SODE

$$\mathrm{d}O_t = -\frac{1}{\tau}O_t\,\mathrm{d}t + \sqrt{c}\,\mathrm{d}W_t \tag{7.4}$$

with parameters τ and c representing the diffusion and relaxation times of the OU process. We set $\tau = c = 1$ in this work but our implementation supports arbitrary τ and c, as well as arbitrary ζ_g and ω_g. Expected value and variance for the OU process are given in [62]. According to [87], the OU process can be numerically exactly integrated. With a *(coarse) timestep size h*, $\mu_X = e^{-\frac{h}{\tau}}$, and $\sigma_X = \sqrt{\frac{c\tau}{2}(1-\mu_X^2)}$, the exact solution of (7.4) is

$$O_{t+h} = O_t \cdot \mu_X + \sigma_X \cdot n_1. \tag{7.5}$$

n_1 is a sample from normal distribution $\mathcal{N}(0,1)$ with zero mean and variance 1. The realization of such a sample on a computer consumes a considerable amount of runtime, hence, chapter 8 deals with implementations of this operation on GPUs forming the first building block. Moreover, the realization of the OU process (7.5) is strictly sequential. Its parallelization leads to the second building block explained in chapter 9.

With the OU process (7.4), the SODE (7.3) can be reformulated as a first-order RODE system via the DSIS correspondence:

$$\begin{pmatrix} \dot{z}_1 \\ \dot{z}_2 \end{pmatrix} = \begin{pmatrix} -(z_2 + O_t) \\ -2\zeta_g\omega_g(z_2 + O_t) + \omega_g^2 z_1 + O_t \end{pmatrix} \tag{7.6}$$

with $Z = (z_1, z_2)^T \in \mathbb{R}^2$. The transformation is demonstrated in more detail in [173].

In the now following section, we provide low- and high-order schemes to numerically solve (7.6).

7.3. Numerical schemes for RODEs

Classical solvers for ODEs such as Euler, Heun, or Runge-Kutta schemes in general can also be applied to deal with path-wise RODEs (7.2) in general and thus, also with the KT RODE (7.6) in special. However, such solvers suffer a considerable decrease in their order of convergence if applied to path-wise RODEs, more precisely, the convergence lies in order of $\mathcal{O}(h^{\frac{1}{2}})$. Due to the stochastic process ξ_t, the right-hand side of the KT RODE (7.6) is only continuous or at most Hölder continuous but not differentiable in t. Hence, the right-hand side is not smooth leading to the decrease in the order of convergence. Therefore, various solvers for path-wise RODEs have been developed [122, 92]. In this section, we present two classes of path-wise RODE solvers. First, averaged schemes are introduced in subsection 7.3.1 followed by the higher-order K-RODE-Taylor schemes in subsection 7.3.2. We conclude this section with a short discussion on the correct choice of a particular solver in subsection 7.3.3.

7.3.1. Averaged schemes

In principle, averaged schemes are simply modified classical solvers such as Euler and Heun using averaged values of the right-hand side of (7.6). Consequently, this leads to the averaged Euler and averaged Heun schemes.

Let us consider the family of RODEs with separable vector field

$$\frac{\mathrm{d}x}{\mathrm{d}t} = F_\omega(t, x) := G(t) + g(t) \cdot H(x). \tag{7.7}$$

$H : \mathbb{R}^c \to \mathbb{R}^c$ is an at least once continuously differentiable function, $g : [0, T] \to \mathbb{R}$, and $G : [0, T] \to \mathbb{R}^c$. Separating the right-hand side of (7.6) according to (7.7),

$$F_\omega(t, Z) = \begin{pmatrix} -(z_2 + O_t) \\ -2\zeta_g\omega_g(z_2 + O_t) + \omega_g^2 z_1 + O_t \end{pmatrix}$$

leads for example to

$$G(t) := -O_t \begin{pmatrix} 1 \\ 2\zeta_g\omega_g - 1 \end{pmatrix} \tag{7.8a}$$

$$g(t) := 1 \tag{7.8b}$$

$$H(Z) := -\begin{pmatrix} z_2 \\ 2\zeta_g\omega_g z_2 - \omega_g^2 z_1 \end{pmatrix} \tag{7.8c}$$

with $c = 2$ and shows that g and H are only deterministic parts while G incorporates the OU process.

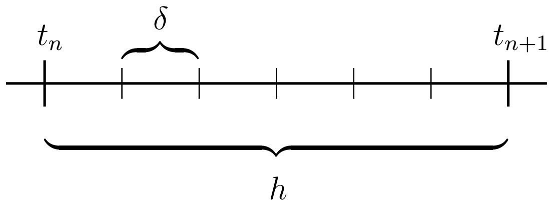


Figure 7.1.: Subdivision of the interval $[t_n, t_{n+1}]$ with length h in $M = 6$ sub-intervals of length δ. This figure is taken from [172].

The underlying idea of the averaged schemes is to apply a classical explicit step with timestep size $h = t_{n+1} - t_n$ to (7.7) and substituting G and g with averaged values $\bar{G}$ and $\bar{g}$. n denotes the n-th interval $[t_n, t_{n+1}]$ of discretized time $[0, T]$. $\bar{G}$ and $\bar{g}$ are obtained by using a considerably finer discretization in time to compensate for the reduced smoothness of the stochastic processes. Hence, the time interval $[t, t + h]$ is subdivided in M sub-intervals of length $\delta = \frac{h}{M}$ as depicted by figure 7.1.

The application of the explicit Euler scheme to (7.7) results in

$$y_{n+1} = y_n + h \cdot F_\omega(t_n, y_n).$$

The single-averaged values $\bar{g}^{(1)}_{h,\delta}(t)$ and $\bar{G}^{(1)}_{h,\delta}(t)$ defined as

$$\bar{g}^{(1)}_{h,\delta}(t) = \frac{1}{M} \sum_{j=0}^{M-1} g(t + j\delta) \tag{7.9a}$$

and

$$\bar{G}^{(1)}_{h,\delta}(t) = \frac{1}{M} \sum_{j=0}^{M-1} G(t + j\delta) \tag{7.9b}$$

are used as the averaged values $\bar{G}$ and $\bar{g}$. Applying (7.8a) to (7.9b) leads to (10.1) in section 10.1 in the chapter of the third building block. As result, we get for the averaged Euler scheme the explicit iteration rule

$$y_{n+1} := y_n + h \left(\frac{1}{M} \sum_{j=0}^{M-1} G(t_n + j \cdot \delta) + H(y_n) \cdot \frac{1}{M} \sum_{j=0}^{M-1} g(t_n + j \cdot \delta) \right). \tag{7.10}$$

Like the explicit Euler method for ODEs, the averaged Euler scheme for path-wise RODEs (7.10) possesses an order of convergence of 1 [92] if M, i.e. the number of sub-samples, is chosen correctly.

Analogously, the explicit Heun method can be modified to the averaged Heun method. The application of the explicit Heun scheme to (7.7) results in

$$y_{n+1} = y_n + h \cdot \frac{1}{2} \left(F_\omega(t_n, y_n) + F_\omega(t_{n+1}, \underbrace{y_n + h \cdot F_\omega(t_n, y_n)}_{\text{averaged Euler}}) \right).$$

Hence, two evaluations of the right-hand side F_ω are necessary being still considered to be separable. In addition to the single-averaged functions $\bar{g}^{(1)}_{h,\delta}(t)$ and $\bar{G}^{(1)}_{h,\delta}(t)$, double-averaged functions $\bar{g}^{(2)}_{h,\delta}(t)$ and $\bar{G}^{(2)}_{h,\delta}(t)$ are necessary, defined as

$$\bar{g}^{(2)}_{h,\delta}(t) = \frac{2}{M^2} \sum_{i=0}^{M-1} \sum_{j=0}^{i} g(t + j \cdot \delta) = \frac{2}{M^2} \sum_{j=0}^{M-1} (M - j) \cdot g(t + j \cdot \delta) \tag{7.11a}$$

and

$$\bar{G}^{(2)}_{h,\delta}(t) = \frac{2}{M^2} \sum_{j=0}^{M-1} (M - j) \cdot G(t + j \cdot \delta). \tag{7.11b}$$

Applying (7.8a) to (7.11b) leads to (10.2) in third building block's section 10.1. Accordingly, the iteration rule for the averaged Heun method for the RODE (7.7) is

$$\begin{aligned} y_{n+1} := y_n + \frac{h}{2} \Bigg(& \underbrace{\bar{G}^{(1)}_{h,\delta}(t_n) + H(y_n) \cdot \bar{g}^{(1)}_{h,\delta}(t_n)}_{F_\omega(y_n,t_n)} + \\ & + \bar{G}^{(1)}_{h,\delta}(t_n) + H\Big(y_n + h\Big(\bar{G}^{(2)}_{h,\delta}(t_n) + H(y_n) \cdot \bar{g}^{(2)}_{h,\delta}(t_n)\Big)\Big) \cdot \bar{g}^{(1)}_{h,\delta}(t_n)\Bigg). \end{aligned} \tag{7.12}$$

In contrast to the averaged Euler scheme (7.10), we do not substitute the single- and double-averaged values for clarity. Again, if M is chosen correctly, the averaged Heun scheme for path-wise RODEs (7.12) and the explicit Heun scheme possess the same order of convergence, namely 2 [92].

The correct *sub-interval length* $\delta = \frac{h}{M}$, in the following also called *sub-step size* or *fine timestep size*, for the averaged Euler method depends linearly on h, thus, $\delta = h \cdot h = h^2$ and and $M = \lceil \frac{1}{h} \rceil$. Hence, the sub-step size is significantly smaller than the coarse timestep size h (for $h < 1$). For the averaged Heun method, the fine timestep size is even two orders of magnitude smaller than for the averaged Euler method, i.e. δ depends cubic on h. This leads to $\delta = h \cdot h^3 = h^4$ and and $M = \frac{1}{h^3}$ for the averaged Heun method. Due to the required sub-sampling, the averaged values introduce a severe increase in computational effort. This increase suggests the usage of GPUs to solve path-wise RODEs.

Plugging the single- and double-averaged values in the numerical solver is the task of the final building block presented in chapter 11. For the averaged Euler and Heun method, this process is demonstrated in section 11.1.

7.3.2. K-RODE-Taylor schemes

K-RODE-Taylor schemes are a class of solvers for RODE initial value problems derived by Jentzen [114] and Kloeden [122] providing arbitrary order of convergence K, converging for each $K > 0$ and having a global discretization error of $K - 1$. Hence,

K-RODE-Taylor schemes can also be used as high-order schemes. They base on Taylor expansions of the right-hand side function f of

$$\frac{\mathrm{d}x}{\mathrm{d}t} = F_\omega(t,x) := f(\omega(t),x), \qquad x(t_0) = x_0$$

in the smooth directions of variables X_t and ω for (almost) all $\omega \in \Omega$. Their flexibility for arbitrary K comes at the price of a slightly more complex notation and evaluations. First, we describe the K-RODE-Taylor schemes in their general form before we adapt them for the KT model in section 11.2.

Let $\alpha = (\alpha_1, \alpha_2) \in \mathbb{N}_0^2$ be a multi-index with magnitude $|\alpha| := \alpha_1 + \alpha_2$ and $\alpha! := \alpha_1!\alpha_2!$. The magnitude can be generalized by a weight $\theta \in ()0,1]$ such that $|\alpha|_\theta := \theta\alpha_1 + \alpha_2$ and for each $K \in \mathbb{R}_+$ with $K \geq |\alpha|_\theta$, let $|\alpha|_\theta^K := K - |\alpha|_\theta$. The underlying stochastic process determines the correct choice of θ. It is set to the supremum of the Hölder coefficients of the process' sample paths. In our case of RODEs, $\theta = \frac{1}{2}$ because the underlying process is the OU process (7.4). With K being the order of (the local error of) the scheme, we can define the sets of multi-indices

$$\mathcal{A}_K := \left\{\alpha = (\alpha_1,\alpha_2) \in \mathbb{N}_0^2 : \quad |\alpha|_\theta = \theta\alpha_1 + \alpha_2 < K\right\}.$$

The multi-index is introduced due to the Taylor expansion of f in directions X_t and ω. So, let partial derivatives with respect to the multi-index be denoted by $f_\alpha := (\partial_1)^{\alpha_1}(\partial_2)^{\alpha_2} f$, with $f_0 = f$. The K-RODE-Taylor scheme can be expressed via the explicit iteration rule

$$y_{n+1}^{K,h} := y_n^{K,h} + \sum_{\alpha \in \mathcal{A}_K} N_\alpha^{(K)}(t_{n+1}, t_n, y_n^{K,h}) \tag{7.13}$$

with

$$N_\alpha^{(K)}(t_{n+1}, t_n, y_n) := \frac{1}{\alpha!} f_\alpha(\omega(t_n), y_n) \int_{t_n}^{t_{n+1}} (\Delta\omega_s)^{\alpha_1} \left(\Delta M_{\Delta s}^{(|\alpha|_\theta^K)}(t_n, y_n)\right)^{\alpha_2} \mathrm{d}s, \tag{7.14a}$$

$$\Delta M_h^{(\ell)}(t_n, y_n) := \sum_{\alpha \in \mathcal{A}_\ell} N_\alpha^{(\ell)}(t_{n+1}, t_n, y_n), \tag{7.14b}$$

$$\Delta\omega_s := \omega(s) - \omega(t_n),$$

and

$$\Delta s = s - t_n.$$

Note the superscripts K and h of the numerical solution $y_n^{K,h}$ being used to distinguish solutions of different order of convergence. As visible in (7.14a) and (7.14b), the definition of the K-RODE-Taylor scheme (7.13) is recursive because the term $\Delta M_{\Delta s}^{(|\alpha|_\theta^K)}$ is of order $|\alpha|_\theta^K = K - |\alpha|_\theta < K$.

Since the derivation of specific K-RODE-Taylor schemes for concrete K requires considerable effort due to the recursivity, especially for larger K, we provide the explicit iteration rules for $K = 1, 3$, and 4. The formulas for $K = 3$ and 4 are taken from [188]. With $\Delta O_s := O_s - O_{t_n}$, the 1-RODE-Taylor scheme is

$$y_{n+1}^{1,h} = y_n^{1,h} + hf(O_{t_n}, z) + f_{(1,0)}(O_{t_n}, z) \int_{t_n}^{t_{n+1}} \Delta O_s \mathrm{d}s,$$

and the scheme for $K = 3$ reads

$$\begin{aligned} y_{n+1}^{3,h} = y_n^{3,h} &+ hf + f_{(1,0)} \int_{t_n}^{t_{n+1}} \Delta O_s \mathrm{d}s \\ &+ \frac{h^2}{2} f_{(0,1)} f + f_{(0,1)} f_{(1,0)} \int_{t_n}^{t_{n+1}} \int_{t_n}^{s} \Delta O_v \mathrm{d}v \, \mathrm{d}s \\ &+ \frac{h^3}{6} f_{(0,1)}^2 f + f_{(0,1)}^2 f_{(1,0)} \int_{t_n}^{t_{n+1}} \int_{t_n}^{s} \int_{t_n}^{v} \Delta O_w \mathrm{d}w \, \mathrm{d}v \, \mathrm{d}s. \end{aligned} \tag{7.15}$$

In the case of order $K = 4$, the corresponding K-RODE-Taylor scheme includes two additional terms and has the form

$$\begin{aligned} y_{n+1}^{4,h} = y_n^{4,h} &+ hf + f_{(1,0)} \int_{t_n}^{t_{n+1}} \Delta O_s \mathrm{d}s \\ &+ \frac{h^2}{2} f_{(0,1)} f + f_{(0,1)} f_{(1,0)} \int_{t_n}^{t_{n+1}} \int_{t_n}^{s} \Delta O_v \mathrm{d}v \, \mathrm{d}s \\ &+ \frac{h^3}{6} f_{(0,1)}^2 f + f_{(0,1)}^2 f_{(1,0)} \int_{t_n}^{t_{n+1}} \int_{t_n}^{s} \int_{t_n}^{v} \Delta O_w \mathrm{d}w \, \mathrm{d}v \, \mathrm{d}s \\ &+ \frac{h^4}{24} f_{(0,1)}^3 f + f_{(0,1)}^3 f_{(1,0)} \int_{t_n}^{t_{n+1}} \int_{t_n}^{s} \int_{t_n}^{v} \int_{t_n}^{w} \Delta O_x \mathrm{d}x \, \mathrm{d}w \, \mathrm{d}v \, \mathrm{d}s. \end{aligned} \tag{7.16}$$

Most of the K-RODE-Taylor schemes contain multiple integrals, for example (7.15) and (7.16). The multi-integrals can be rewritten as

$$\int_{t_n}^{t_{n+1}} \int_{t_n}^{x_d} \int_{t_n}^{x_{d-1}} \cdots \int_{t_n}^{x_1} g(z) \mathrm{d}z \, \mathrm{d}x_1 \ldots \mathrm{d}x_d = \int_{t_n}^{t_{n+1}} \frac{1}{d!} (t_{n+1} - z)^d g(z) \mathrm{d}z \tag{7.17}$$

which strongly simplifies the integration. Equivalence of left- and right-hand side of (7.17) is shown in [114]. The simplified multi-integral (7.17) can be approximated via a quadrature rule. It does not make sense to use a high-order quadrature rule because the stochastic process involved in f is typically only continuous. Thus, a low-order quadrature rule is sufficient. There are numerous candidates such as trapezoidal sums but we realize the approximation via Riemann sums. Then, the single integral is approximated via

$$\int_{t_n}^{t_{n+1}} \frac{1}{d!} (t_{n+1} - z)^d g(z) \mathrm{d}z \approx \delta \sum_{j=1}^{M} \frac{1}{d!} (t_{n+1} - z_j)^d g(z_j) \tag{7.18}$$

with $z_j := t_n + j\delta$. Evaluating the quadrature rule in parallel represents the third building block for the K-RODE-Taylor scheme demonstrated in section 10.2.

Similar to the averaged Euler and Heun scheme, M has to be chosen properly to achieve the desired order of convergence K. Here, the fine timestep size depends to the power of K on the coarse timestep size h, thus, for $K = 1$, $\delta = h \cdot h^1 = h^2$ and $M = \lceil \frac{1}{h} \rceil$. Analogously, for $K = 3$, $\delta = h \cdot h^3 = h^4$ and $M = \lceil \frac{1}{h^3} \rceil$ and for $K = 4$, $\delta = h \cdot h^4 = h^5$ and thus $M = \lceil \frac{1}{h^4} \rceil$. The enormous degree of sub-sampling leads to an extremely high computational load which can be tackled by GPUs.

7.3.3. Remarks on numerical schemes

To conclude this section, we would like to take up a discussion raised in [92, 172] concerning the choice of the numerical scheme in dependence of the number of fine timesteps. Both, the averaged schemes in subsection 7.3.1 and the K-RODE-Taylor schemes in subsection 7.3.2 require for every coarse timestep of length h several fine timesteps of length δ. For the averaged schemes, G and g have to be evaluated at the fine timesteps to retrieve single- ((7.9a) and (7.9b)) and double-averaged ((7.11a) and (7.11b)) values. Similarly, the K-RODE-Taylor schemes require a fine evaluation of f for the numerical quadrature rule, e.g. (7.18). Actually, both schemes calculate some sort of mean value. To obtain a certain order of convergence, δ has to be chosen properly. Let us assume we want to solve a path-wise RODE with an absolute error of order $\mathcal{O}(10^{-4})$. If we use the averaged Euler method, we have to choose $h = 10^{-4}$ and, thus, $\delta = 10^{-8}$. Using the averaged Heun method instead leads to $h = 10^{-2}$ and, thus, $\delta = 10^{-8}$. The explicit Euler method (without sub-sampling) with its order of convergence of $\frac{1}{2}$ could also be used with $h = 10^{-8}$. Finally, the 3-RODE-Taylor scheme would also do the job with $h = 10^{-1}$ and $\delta = 10^{-8}$. All four methods need the same amount of sub-samplings because all δ are the same order of magnitude. The best choice for such a scenario are higher order schemes as stated in [92]. Higher order schemes allow a larger choice of h and, thus, less evaluations of the whole right-hand side are necessary. In addition, it is easier to compute the averages and Riemann sums, respectively, instead of applying classical explicit schemes.

8. Building block 1: Pseudo random number generation

Various applications of Computational Science and Engineering (CSE) essentially rely on random numbers. Many models incorporate the realization or approximation of stochastic processes and Monte Carlo sampling, both holding for RODEs. Furthermore, fields such as performance modeling and cryptography fundamentally depend on random numbers. There are several sources for random numbers. Real randomness can be sampled from suited physical processes, for example from radioactive decay. Such physical processes suffer in general from low generation rates, thus, the amount of random numbers per time is low. Another, very popular, source for random numbers are pseudorandom number generators (PRNGs) [85]. PRNGs are deterministic rules generating sequences of random numbers which can be implemented on computers [41, 132]. Thus, PRNGs do not generate real randomness but random numbers fulfilling certain statistical criteria [143]. For simplicity, we omit the term pseudo in the following even if we restrict ourselves to pseudo random numbers.

There are a lot of PRNG categories each comprising numerous generators satisfying different statistical distributions (uniform, normal, exponential, etc.) with diverse capabilities, properties, and characteristics [133, 70, 229]. Almost all PRNGs produce uniformly distributed random numbers, in the following called uniform random numbers and analogously normal, exponential, etc. random numbers. If a random number from another distribution is required, an uniform random number is generated first and afterwards transformed to the desired distribution by a corresponding operation. In most cases, such operations try to approximate the inverse cumulative distribution function (CDF) of the aimed distribution. Examples to directly approximate the inverse normal CDF are given in, e.g. [47, 196]. Hence, generating random numbers complying non-uniform distributions is in most cases a two stages process. There are exceptional generators directly generating random numbers of a certain distribution but they are quite exotic and uncommon. The combination of uniform random number generation followed by transformation to target distribution is also called PRNG even if it is not a single step assimilation. In our implementations of such PRNGs, we always fuse the generation of the uniform random number and its transformation in one single kernel.

In this work, normal random numbers are required due to the solution of the OU process (7.5). So we focus on the generation of normal random numbers representing the first building block of our RODE HPC implementation and leaving out the generation of uniform random numbers. High-performance implementations of uniform generators, especially for GPUs, can be found in [23, 134]. The PRNGs introduced in this chapter are not necessarily limited to normal distribution: Every time one of the treated PRNGs is also capable to deal with other distributions, we mention this fact accordingly. There is already much experience in optimizing PRNGs, especially for CPUs. Yet when it comes to GPUs, alternative and lesser known generators fitting well on the accelerator's hardware are promising. Hence, we deal with three uncommon PRNGs which are typically not considered the first choice when it comes to the generation of normal random numbers due to their difficult implementation, high complexity, and bad performance because of immense computational intensity. In contrast to CPUs, these properties can be partly beneficial on GPUs. The three investigated PRNGs are the Ziggurat method, rational polynomials to approximate the inverse CDF of the normal distribution (in the following just called rational polynomials), and the Wallace method. We are not introducing any newly developed generators, and this chapter is also not a broad overview on general PRNGs. The three mentioned methods are optimized for GPUs exploiting intrinsic properties of the particular PRNGs making them competitive against or even superior to well established normal PRNGs on CPU and GPU. Results of this endeavor are published by us in [197, 198] and we transfer the articles' structure to this chapter. Statistical properties of PRNGs can be experimentally checked with test batteries such as Diehard [153] or TestU01 [135, 136]. Investigating those is, however, not within the scope of this thesis. A discussion on popular methods such as Mersenne Twister [159], Polar [155], and Box/Muller [38] in the context of GPUs is provided by us in [200] but is not content of this chapter.

The common parallelization approach for PRNGs is not to parallelize the generation of a single continuous sequence of random numbers but to assign one (sub-) sequence to every thread. Ergo, parallelizing a PRNG means generating multiple sequences in parallel. From a statistical point of view, that is not a problem at all because interleaved (sub-) sequences of random numbers are as random as concatenated (sub-) sequences of random numbers. There are also approaches to cooperatively work on one single sequence of random numbers called *counter-based* PRNGs [207, 58].

There is a dedicated section for every considered normal PRNG in this chapter: The setup and function of the Ziggurat method as well as its encouraging memory/runtime trade-off are explained in section 8.1. Together with rational polynomials discussed in section 8.2, they are representatives of transformation functions depending on a source of uniform random numbers. This does not hold for the Wallace method in section 8.3: It is a method to directly obtain normal random numbers without a preceding step. Section 8.4 concludes this chapter with benchmark and profiling results of the presented PRNGs and draws a comparison with state-of-the-art library functions for CPUs and GPUs. We suspend the notation for RODEs introduced in previous chapter 7 for this chapter because we reuse some of the identifiers.

8.1. The Ziggurat method

The Ziggurat method is a rejection method also used in MATLAB for its normal PRNG[1]. It is named after Ziggurats, massive buildings having the form of a terraced step pyramid

[1] `https://de.mathworks.com/company/newsletters/articles/normal-behavior.html`

of successively receding levels. In the best case, it realizes the transformation from uniform to normal distribution with just one table lookup and one multiplication. However, in all other cases, the transformation becomes much more expensive by evaluations of `sqrt()`, `exp()`, and `log()` or a restart of the method. Via a memory/runtime trade-off it is possible to increase the likelihood of the cheap cases making the Ziggurat method an interesting candidate for GPUs. It is not only capable to transform from uniform to normal distribution but to transform to every distribution with decreasing probability density function (PDF) such as the exponential distribution.

Marsaglia et al. first proposed the Ziggurat method in the original paper [156]. Over time, the method was improved in terms of simplicity and performance with the latest version presented in [157] building the base for our GPU implementation. We focus on the implementation and optimization for GPUs but a discussion on the statistical properties of the Ziggurat method can be found in [72, 138]. There are already several successful attempts to implement the Ziggurat method on various special purpose hardware, mainly on Field Programmable Gate Arrays (FPGAs). Examples of these attempts can be found in [256, 74, 67]. Thomas et al. present in [228] an extensive survey on several, also massively parallel architectures where the Ziggurat method turns out to be the best choice on CPUs but not on GPUs. We come to a different conclusion as shown in section 8.4. The memory/runtime trade-off is also exploited by Buchmann et al. [50] for their cryptosystem application but their implementation is limited to a normal distribution for integers.

The structure and notation of this section follows [197]. First, we introduce a Ziggurat-shaped approximation of the area under the normal distribution's PDF in subsection 8.1.1. This approximation is used for the transformation from uniform to normal distribution in subsection 8.1.2 detailing on cheap and expensive cases. Hence, the Ziggurat method requires uniform random numbers as input. The setup of the approximating Ziggurat is non-trivial, thus, we explain its construction in subsection 8.1.3. Finally, subsection 8.1.4 illustrates how the likelihood for cheap and expensive cases can be altered via the memory/runtime trade-off.

8.1.1. Definition of the Ziggurat

Figure 8.1 depicts how the Ziggurat approximates the area under the Gaussian function $f(x) = e^{-\frac{x^2}{2}}$ for $x \geq 0$. The Ziggurat consists of N vertically stacked, axis-aligned *strips*. f is the PDF of the normal distribution and N can be an arbitrary number ≥ 2. There are two different kinds of strips: On the one hand, there are $N-1$ rectangular shaped strips $R_i, i = 0, \ldots, N-2$, in the following simply called *rectangles*. On the other hand, there is one single *base strip* $R_{N-1} = R_B$ having a different shape and hatched from bottom left to top right in figure 8.1. The rectangles $R_0, \ldots, R_{N-2}$ are bounded from the left by $x = 0$, from the right by x_{i+1}, and from the bottom and top by y_i and y_{i+1}, respectively, with $f(x_i) = y_i$ and $0 = x_0 < x_1 < \ldots < x_{N-1} = r$. The base strip R_B is bounded from the left by $x = 0$, from the right by $f(x)$, from the bottom at $y = 0$, and from the top at $y = y_{N-1}$. v denotes the common area of every single strip R_i, so all

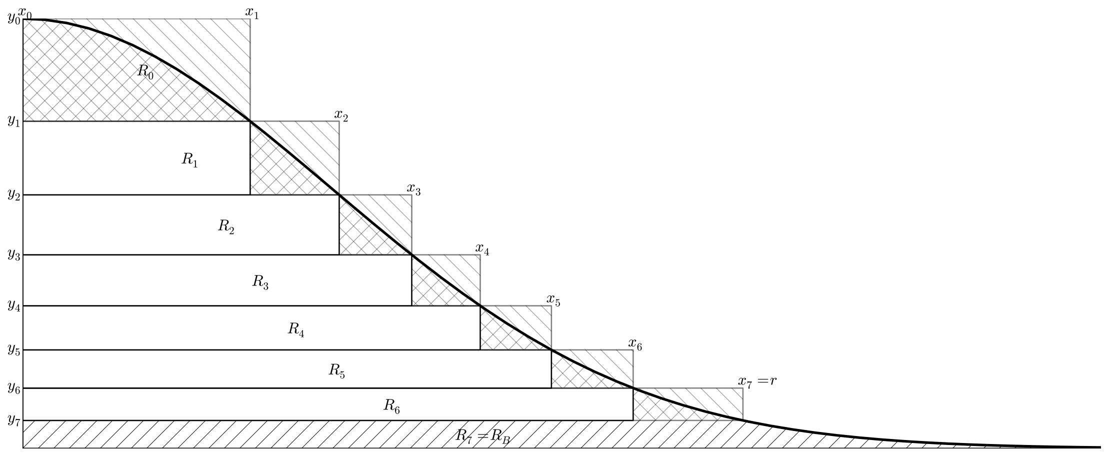


Figure 8.1.: Eight strips (seven rectangles and one base strip (hatched from bottom left to top right)) forming a Ziggurat. Central regions are not hatched, tail regions are hatched with diagonal crosses, and cap regions are hatched from bottom right to top left. This figure is taken from our contribution [198].

strips have the same area, even the non-rectangular R_B. All rectangles $R_0, \dots, R_{N-2}$ are further subdivided in three regions: The *central region* of R_i is an axis-aligned rectangle with upper-left corner $(0, y_i)$ and lower right corner (x_i, y_{i+1}). R_0 does not have such a central region because $x_0 = 0$. R_i's *tail region* is bordered by $x = x_i$ from the left, $y = y_{i+1}$ from the bottom, and f from right and top, thus, tail regions are no rectangles. Finally, R_i's *cap region* is the part of R_i which is not covered by the central and tail region, thus, bordered by f from left and bottom, from right by $x = x_{i+1}$, and from $y = y_i$ from top. While the central and tail region lie completely below the PDF, the cap region lies completely above it. Central regions are not hatched, tail regions are hatched with diagonal crosses, and cap regions are hatched from bottom right to top left in figure 8.1. Consequently, the union of the three regions results in the rectangles $R_0, \dots, R_{N-2}$.

8.1.2. Algorithmic description of the Ziggurat method

Algorithm 5 outlines the Ziggurat method. For the transformation of a uniform integer random number $u^{\text{uniform}}_{\texttt{int}}$ and its normalized floating-point value $u^{\text{uniform}}_{\texttt{float}} \in [0, 1]$, respectively, to a normal random number, u^{normal}, $u^{\text{uniform}}_{\texttt{int}}$ is used in two different ways: First, obviously, $u^{\text{uniform}}_{\texttt{int}}$ is the random number being transformed. Second, it is used to randomly select one particular strip R_k. If the number of strips of the Ziggurat N is chosen as a power of 2, thus $N = 2^n$, then the selection of a strip R_k can be done in a very efficient way by using the n least significant (or any other) bits of $u^{\text{uniform}}_{\texttt{int}}$. The selected strip has index k (line 4). Depending on $u^{\text{uniform}}_{\texttt{int}}$, one of the following four cases can occur:

Algorithm 5 The Ziggurat method

```
procedure ZIGGURAT(n, r)
    $u^{\text{uniform}}_{\texttt{int}} \leftarrow$ uniform integer number
    $u^{\text{uniform}}_{\texttt{float}} \leftarrow$ normalized $u^{\text{uniform}}_{\texttt{int}}$
    $k \leftarrow u^{\text{uniform}}_{\texttt{int}} \& (2^n - 1)$                          ▷ select a strip
    if $u^{\text{uniform}}_{\texttt{float}} \leq \frac{x_k}{x_{k+1}}$ then                          ▷ central
        $u^{\text{normal}} \leftarrow u^{\text{uniform}}_{\texttt{float}} \cdot x_{k+1}$
        return $u^{\text{normal}}$
    else
        while true do
            if $k = N - 1$ then                          ▷ normal tail of base strip
                repeat
                    $(u_1, u_2) \leftarrow$ pair of normalized uniform numbers
                    $(t_1, t_2) \leftarrow \left(\frac{-\ln(u_1)}{r}, -\ln(u_2)\right)$
                until $t_1^2 > t_2^2$
                $u^{\text{normal}} \leftarrow r + t_1$
                return $u^{\text{normal}}$
            else
                $u^{\text{normal}} \leftarrow u^{\text{uniform}}_{\texttt{float}} \cdot x_{k+1}$
                $t_1 \leftarrow$ normalized uniform number
                $t_2 \leftarrow t_1 \cdot (f(x_k) - f(x_{k+1}))$
                if $t_2 < f(u^{\text{normal}}) - f(x_{k+1})$ then                          ▷ tail
                    return $u^{\text{normal}}$
                else                          ▷ cap
                    $u^{\text{uniform}}_{\texttt{int}} \leftarrow$ uniform integer number
                    $u^{\text{uniform}}_{\texttt{float}} \leftarrow$ normalized $u^{\text{uniform}}_{\texttt{int}}$
                    $k \leftarrow u^{\text{uniform}}_{\texttt{int}} \& (2^n - 1)$
                    continue
```

(a) A central region is hit:
This occurs if $k \neq N - 1$ and $u^{\text{uniform}}_{\texttt{float}} \leq \frac{x_k}{x_{k+1}}$ (line 5).

(b) A tail region is hit:
This occurs if $k \neq N - 1$, the central region is not hit and $u^{\text{uniform}}_{\texttt{float}} \cdot (f(x_k) - f(x_{k+1})) < f(u^{\text{uniform}}_{\texttt{float}} \cdot x_{k+1}) - f(x_{k+1})$ (line 21).

(c) A cap region is hit:
This occurs if $k \neq N - 1$ and neither the central nor tail region are hit (line 23).

(d) The base strip it hit:
This occurs if $k = N - 1$ (line 10).

If cases (a) and (b) occur, the transformation can simply be done by the operation $u^{\text{normal}} = u^{\text{uniform}}_{\texttt{float}} \cdot x_{k+1}$ (lines 6 and 18, respectively), thus, a multiplication and a lookup for the value x_{k+1}. For the cap region case (c), the Ziggurat method is restarted (line 27) with a new u^{uniform} (lines 24–26) leading to additional computational costs. This behavior has no impact on the statistics of the Ziggurat method because the likelihood for a particular u^{uniform} not hitting a cap region is still uniform. It only decreases the number of possible samples. Finally, for the much more expensive case (d), two sub-cases have to be considered depending on $x = \frac{v \cdot u^{\text{uniform}}_{\texttt{float}}}{f(r)}$: If $x < r$, then $u^{\text{normal}} = x$. Otherwise, a treatment of the normal tail as described by Marsaglia [152] is necessary involving the generation of further uniform random numbers and the evaluation of `ln()`. Lines 11–16 outline the two sub-cases and Marsaglia's treatment of the normal tail.

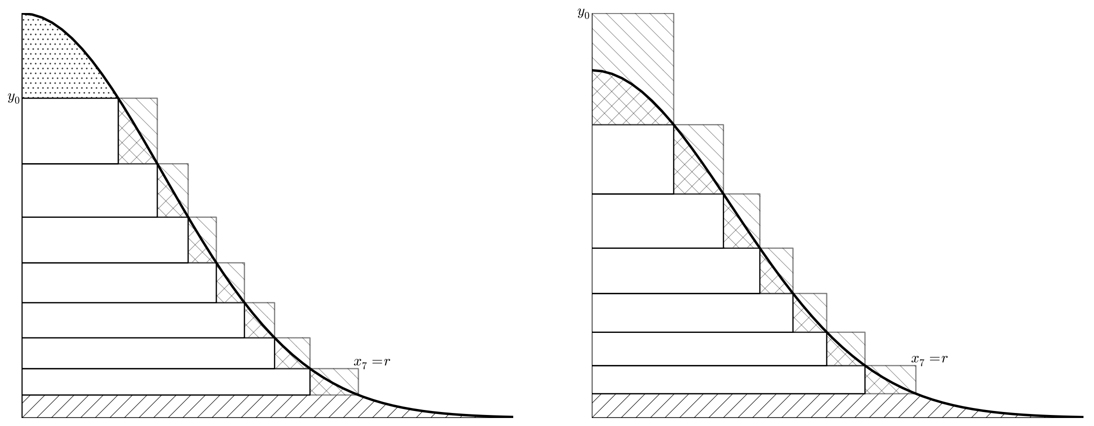
8.1.3. Setup of the Ziggurat

For now, a Ziggurat approximating the area under the normal PDF with N strips of area v was expected to be simply given as in subsection 8.1.1. Now, we determine how to setup the Ziggurat.

Algorithm 6 Setup of the Ziggurat

```
1: procedure SETUP(N, r)
2:     $x_{N-1} \leftarrow r$
3:     $v \leftarrow r \cdot f(r) + \underbrace{\int_r^\infty f(x)dx}_{\sqrt{\frac{\pi}{2}} \cdot \text{erfc}\left(\frac{r}{\sqrt{2}}\right)}$
4:     for $(N-2) \leq i \leq 1$ do
5:         $x_i = f^{-1}\left(\frac{v}{x_{i+1} + f(x_{i+1})}\right)$
6:     return $v - (x_1(1 - f(x_1))$
```

Determining the Ziggurat means finding all $x_i, i = 0, \ldots, N-1$ such that all strips have area v, $x_0 = 0$, and $f(x_0) = y_0 = 1$. Once the rightmost rectangle border $r = x_{N-1}$ is found, all other right rectangle borders $0 = x_0 < x_1 < \ldots < x_{N-2}$ can be determined by stacking rectangles with $v = r \cdot f(r) + \int_r^\infty f(x)dx$ on top of each other according to algorithm 6. The next x_i is computed via the formula $x_i = f^{-1}\left(\frac{v}{x_{i+1}} + f(x_{i+1})\right)$ where x_{i+1} is already known. r is figured out via a binary search for a given N: Initially, an arbitrary r is guessed and used in algorithm 6, thus, a Ziggurat is setup for this r. For every iteration of the binary search it is checked if 0 is returned, successfully terminating the binary search. In such a case, all strips have the correct area, even the top rectangle R_0, and so, the correct r is found. If a value < 0 is returned, then r was chosen too large as depicted by subfigure 8.2(a). Hence, a smaller guess for r is used for the next iteration of the binary search. Too small values of r, as illustrated by subfigure 8.2(b), lead to a mathematical problem during the execution of algorithm 6 because arguments > 1 can occur for $f^{-1}(x) : (0, 1] \rightarrow \mathbb{R}^+$. Then, the setup of the Ziggurat has to be canceled in time and for the next iteration of the binary search, a larger guess for r has



(a) (b)

Figure 8.2.: Two examples of a Ziggurat using eight strips with over- and underestimated r. If r is chosen too large as in subfigure 8.2(a), v becomes too small so there are not enough strips to approximate the area under f correctly. The not dealt space is filled with dots. Subfigure 8.2(b) shows a scenario where r is guessed too large so $N \cdot v$ is too big for the approximation. Both subfigures are taken from our contribution [198].

to be used. The original paper of the Ziggurat method [156] demonstrates how to setup the Ziggurat for exponential distribution.

We provide values r and v for a given N, all of them powers of 2, in table 8.1. The table's values are determined by the binary search procedure. We use these values in our own implementation leading to the results in section 8.4. The numbers in table 8.1 offer enough digits for double precision.

During runtime, the values x_i are calculated once at the beginning of execution and are stored afterwards in a lookup table. Hence, the size of such a lookup table depends linearly on N. Besides just storing the right rectangle edges x_i, it can make sense to also manage lookup tables for the ratios $\frac{x_i}{x_{i+1}}$ and the values y_i to save computation time. Here, it depends on the characteristics of the actual computing device if additional lookup tables are useful: CPUs benefit from this approach but on GPUs, the better strategy is calculating the ratios and y_i whenever they are needed because computations are fast while the size of fast memories is limited on GPUs.

The values x_i are constant for a given N. Thus, a single lookup table is sufficient even if multiple normal PRNGs using the Ziggurat method are running in parallel. The Ziggurat method can be tweaked by adding one entry to the lookup table storing the value of $\frac{v}{f(r)}$ in entry x_N. On the one hand, this ensures that the conditional statement in line 5 of algorithm 5 also can be evaluated for $k = N - 1$, and on the other hand, the non-tail case of the base strip can be treated like a central region.

N	r	v
$2 = 2^1$	1.329,233,128,110,321	$7.797,780,032,623,92 \cdot 10^{-1}$
$4 = 2^2$	1.914,928,263,803,744	$3.756,758,421,647,67 \cdot 10^{-1}$
$8 = 2^3$	2.338,371,698,247,252	$1.761,736,401,187,77 \cdot 10^{-1}$
$16 = 2^4$	2.675,536,765,737,614	$8.398,946,374,782,72 \cdot 10^{-2}$
$32 = 2^5$	2.961,300,121,264,019	$4.075,874,443,221,99 \cdot 10^{-2}$
$64 = 2^6$	3.213,657,627,158,896	$2.002,445,715,735,16 \cdot 10^{-2}$
$128 = 2^7$	3.442,619,855,896,652	$9.912,563,035,336,47 \cdot 10^{-3}$
$256 = 2^8$	3.654,152,885,361,008	$4.928,673,233,974,66 \cdot 10^{-3}$
$512 = 2^9$	3.852,046,150,368,391	$2.456,766,351,541,35 \cdot 10^{-3}$
$1024 = 2^{10}$	4.038,849,846,109,505	$1.226,324,646,353,08 \cdot 10^{-3}$
$2048 = 2^{11}$	4.216,370,409,511,898	$6.126,065,176,240,44 \cdot 10^{-4}$
$4096 = 2^{12}$	4.385,945,034,871,305	$3.061,541,032,784,63 \cdot 10^{-4}$
$8192 = 2^{13}$	4.548,600,609,949,139	$1.530,372,349,462,99 \cdot 10^{-4}$

Table 8.1.: Depending on the number of strips N of a Ziggurat, the second column gives rightmost rectangle edge r. Accordingly, the third column lists the area v of a particular Ziggurat strip. Enough digits are provided to use these numbers for double precision computations. This table is taken from our contribution [197].

8.1.4. Memory/runtime trade-off for the Ziggurat method

The Ziggurat method is fast if a central or tail region is hit because in these cases, only a table lookup for and a multiplication with x_{k+1} is required to perform the transformation from uniform to normal distribution. However, if a cap region or the base strip's normal tail is hit, the Ziggurat method becomes very slow, slower then most other approaches for transformation. Thus, we look for a mechanism to increase the likelihood of cases (a) or (b) while reducing the likelihood for cases (c) and hitting the normal tail of R_B.

On GPUs, the maximization of likelihood of one particular case becomes even more crucial due to their SIMT architecture. If not all threads of a warp execute the same case, warp divergence occurs because different threads handle different cases. Warp divergence leads to extended execution times and, thus, a decrease in performance.

The more strips N are used for the Ziggurat, the higher the likelihood gets to hit case (a). This relation is illustrated by figure 8.3 showing four different Ziggurats with $N = 4$, 16, 32, and 64. The larger N, the bigger the un-hatched area representing the likelihood for case (a) gets in comparison to the area of the whole Ziggurat. Since the size of the lookup table depends on N, we achieve a classical runtime/memory trade-off: The more strips are used, the faster the transformation gets due to cheaper calculations of the Ziggurat method (increased likelihood of hitting a central region) and reduced likelihood for warp divergence but the size of the lookup tables grows resulting in higher memory consumption.

To testify the observations of figure 8.3, table 8.2 lists the likelihoods to hit a central

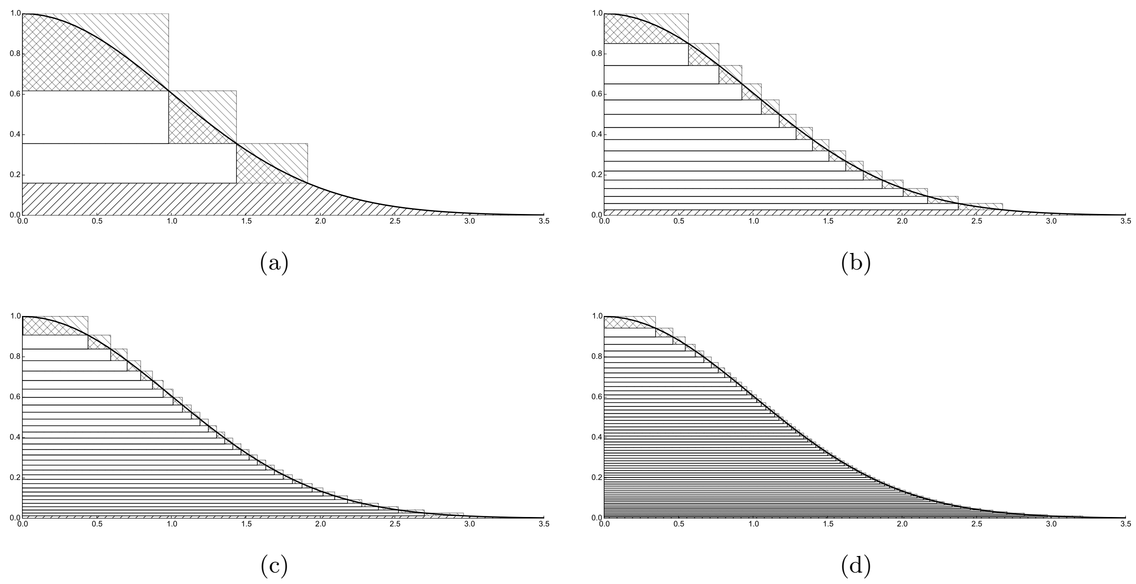


Figure 8.3.: Four Ziggurats using $N = 4$, 16, 32, and 64 strips to approximate the area under the Gaussian PDF. The fewer strips are used (e.g. subfigure 8.3(a)), the smaller the ratio of area covered by central regions (case (a)) becomes in comparison to the total area of the Ziggurat. By using more strips, almost the whole area is covered by central regions (cf. subfigure 8.3(d)). All four subfigures are taken from our contribution [198].

region (case (a)) or the non-tail area of the base strip in dependence of N. The optimization opportunity mentioned at the end of subsection 8.1.3 enables a fast transformation also for the non-tail area of R_B. The more strips are used, the bigger the corresponding values in the second column get and, thus, a fast transformation becomes more likely. These percentages can be used to calculate an indicator for the likelihood that not all threads of a warp deal with a central region, i.e. being an indicator for warp divergence, given in the last column of table 8.2. For larger N, this value decreases.

8.2. Rational polynomials

Instead of trying to approximate the area under the Gaussian bell function, as the Ziggurat method does, explicit functions can be used to directly approximate the inverse CDF. These explicit functions can then directly be applied as transformation functions to alter a uniform random number $u \in [0, 1)$ to the target distribution. Luu [147] gives an example for such explicit functions by using piecewise Chebyshev polynomials using CUDA for implementation, but for the inverse CDF of the gamma distribution. In this section, we are using rational polynomials as explicit functions for the inverse CDF of normal distribution. They can achieve a high degree of accuracy and are computationally

number of strips	likelihood to hit central region or non-tail area of the base strip	likelihood of warp divergence
$2 = 2^1$	$\sim 35.23\%$	$1 - 0.3523^{32} \approx 99.9\%$
$4 = 2^2$	$\sim 56.22\%$	$1 - 0.5622^{32} \approx 99.9\%$
$8 = 2^3$	$\sim 72.80\%$	$1 - 0.7280^{32} \approx 99.9\%$
$16 = 2^4$	$\sim 84.02\%$	$1 - 0.8402^{32} \approx 99.6\%$
$32 = 2^5$	$\sim 90.93\%$	$1 - 0.9093^{32} \approx 95.2\%$
$64 = 2^6$	$\sim 94.96\%$	$1 - 0.9496^{32} \approx 80.9\%$
$128 = 2^7$	$\sim 97.24\%$	$1 - 0.9724^{32} \approx 59.2\%$
$256 = 2^8$	$\sim 98.51\%$	$1 - 0.9851^{32} \approx 38.2\%$
$512 = 2^9$	$\sim 99.20\%$	$1 - 0.9920^{32} \approx 22.7\%$
$1024 = 2^{10}$	$\sim 99.57\%$	$1 - 0.9957^{32} \approx 12.9\%$
$2048 = 2^{11}$	$\sim 99.77\%$	$1 - 0.9977^{32} \approx 7.1\%$
$4096 = 2^{12}$	$\sim 99.88\%$	$1 - 0.9988^{32} \approx 3.8\%$
$8192 = 2^{13}$	$\sim 99.94\%$	$1 - 0.9994^{32} \approx 1.9\%$

Table 8.2.: The second column lists the likelihoods to hit a central region or the non-tail area of the base strip in dependence of N. These values can be utilized to estimate the likelihood for warp divergence caused by at least one thread of a warp not dealing with the just mentioned cheap scenario.

cheap to evaluate.

There is no optimal single set of coefficients for such a rational polynomial. Depending on the applied polynomial degree, the examined interval, and demands on numerical stability, different coefficient sets are suited. Furthermore, a piecewise definition of the approximating rational polynomial makes sense if high accuracy for all regions or complete $\mathbb{R}$ is desired. In such a scenario, the inner region around 0 could be approximated by one rational polynomial while another rational polynomials could be used for the tail regions of f. Generally, rational polynomials allow the approximation of every kind of CDF making them very flexible and widely applicable.

In our implementation of rational polynomials as transformation function, we use the coefficient set suggested by Wichura in [245]. We do not explicitly mention the coefficients but refer to the original paper. Wichura uses a piecewise rational polynomial with a dedicated function for the inner region $\left[-\frac{17}{40}, \frac{17}{40}\right]$ and two for the tail regions. An auxiliary variable $s = \sqrt{-\log(\min(u, 1-u))}$ decides for $s >$ or ≤ 5 over the best fitting rational polynomial at the outer regions to further increase accuracy. Nominator and denominator degree are 7 for all three rational polynomials. As already mentioned, there are alternative coefficient sets for approximations of the inverse normal CDF. For example, Beasley offers in [27] a piecewise rational polynomial with an higher accuracy in $\left[-\frac{7}{2}, \frac{7}{2}\right]$ but less precision in the tails of the distribution than Wichura's set. The actual accuracies and error bounds for the coefficient sets are given in the original publications [27, 245].

The piecewise definition of the approximation suggests a potential risk for warp divergence on the GPU. Depending on u, one of three available rational polynomials is selected to perform the transformation when using Wichura's coefficient set. A popular technique to overcome this issue is blending: Instead of dealing with just one branch of a conditional statement (in the context of our approximation, a branch corresponds to the evaluation of a particular rational polynomial), all branches are evaluated. Afterwards, the results of the branches are weighted by the outcome of the conditional statement causing the branching with weight 0 for `false` and weight 1 for `true`. Finally, all weighted results are summed up leading to the transformed value because the correct branch is multiplied by 1 and all wrong branches are multiplied by 0. This idea introduces many superfluous computations but completely avoids warp divergence. Since GPUs deal very well with computational intense problems but can suffer from the overhead caused by warp divergence, the application of blending is advisable. Furthermore, additions and multiplications accruing during the evaluation of polynomials using the Horner scheme [105] have an especially high throughput on GPUs. In the results section 8.4, benchmarks are given for a version using branching, thus, suffering from warp divergence, and a version using blending, thus, avoiding warp divergence.

8.3. The Wallace method

A completely different approach for a normal PRNG is given by Wallace [237]. Instead of bringing up a transformation operation requiring uniform random numbers as input, he describes a method to directly generate normal random numbers. It uses a chunk of previously computed normal random numbers to evolve them to a new chunk of normal random numbers. This operation can be repeated to generate a sequence of normal random numbers with unlimited length. Wallace's method utilizes a linear operator and gets along without any transcendental functions such as log() or sin() or conditional statements. This property makes the Wallace method a promising candidate for GPUs.

We use the vectorized version of the Wallace method presented in [42] and realized in the library *rannw* [45] as basis for our GPU code. It comes up with some ideas easily adaptable to GPUs leading to a high-performance implementation. Considering alternative computing devices than CPUs and GPUs, there is an implementation of the Wallace method for FPGAs [137]. Some very useful comments in terms of mathematical and historical background can be found in [44].

The idea behind the Wallace method is the usage of the *maximum entropy* property [113], which is $E(x^2) = 1$ for normal distribution. E generally denotes the expected value of a random number x. The Wallace method evolves a vector X of previously generated normal random numbers to a new vector Y of different normal random numbers. This transformation is linear and expressed by $Y = A \cdot X$. Both vectors, X and Y have length k, thus, A is a $k \times k$ matrix with additional mandatory properties: A has to be orthogonal to satisfy the maximum entropy property of the normal distribution because orthogonal matrices preserve the sum of squares. A *pool* consists of l of such vectors of size k, hence, $\nu = k \cdot l$ normal random numbers form a pool. To evolve a complete pool,

l transformation steps are necessary because A is applied to every k elements of the current pool. The transition from an old to an updated pool is called a *pass*. We use a 4×4 orthogonal matrix A, i.e. $k = 4$, but different values for k are also possible: [43, 44] deal with $k = 2$ leading to a rotation matrix in the plane A. Due to the linear operator A, the execution time of a pass depends quadratically on k , so k should not become too large. Even on GPUs, this issue can decrease the performance of the Wallace method in a way disqualifying it as a competitive PRNG.

Since the Wallace method relies on the maximum entropy property, it can also be used to sample from non-normal distributions. For example, the uniform distribution's maximum entropy property is $0 \leq x \leq 1$. This leads to the class of generalized Fibonacci generators [123]: $(u_1 + u_2) \mod 1$ is a uniform random number if u_1 and u_2 are uniform random numbers. Another example of a maximum entropy property is $E(x) = 1$ $(x \geq 0)$ for the exponential distribution.

The Wallace method introduced so far has several statistical flaws. To eliminate them, some modifications are indispensable. First, it is desired that every element of a pool has a contribution to all elements of succeeding pools evolved after some passes. An elegant way to satisfy this property is an alternating reinterpretation of the pool's storage scheme: During odd passes, the pool of normal random numbers being stored linearly in memory is read row-major order (the elements of a k vector lie continuously in memory) while during even passes, the pool is read column-major order (the elements of a k vector lie in memory separated by a stride). Hence, between two passes, an implicit transposition is performed if the pool is interpreted as a $k \times l$ matrix. For large pools, i.e. $l > 256$, additional measurements are crucial to guarantee an adequate mixing. A more general idea of the implicit transposition is the usage of a random odd stride to access the rows of the pool and to use a random offset $\neq 0$ for the first row of the pool. This makes it mandatory to take the adapted row index modulo l. Second, a further improvement of the statistical quality can be reached by using multiple different orthogonal operators instead of only one to perform the passes. For example, Wallace suggests in his original paper [237] four different matrices $A_1, \ldots, A_4$ randomly chosen in every pass instead of only one single A.

$$A_1 = \frac{1}{2}\begin{pmatrix} 1 & 1 & -1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 1 \end{pmatrix} \qquad A_2 = \frac{1}{2}\begin{pmatrix} 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ -1 & -1 & -1 & 1 \end{pmatrix}$$

$$A_3 = \frac{1}{2}\begin{pmatrix} 1 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \end{pmatrix} \qquad A_4 = \frac{1}{2}\begin{pmatrix} -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix}$$

$A_1, \ldots, A_4$ share the common property that they require seven additions and one multiplication with $\frac{1}{2}$ to execute the transformation. Third, the usage of orthogonal operators leads to the obvious defect of constant sum of squares of the numbers of one pool, thus, $||Y||_2 = ||A \cdot X||_2$. Instead, a chi-squared distribution χ^2_ν is aspired for the sum of squares. There are several solutions to overcome this issue but already a very simple one leads to fair results: One random element of every pool is used to approximate a variate V from χ^2_ν. Using this element, a scaling factor $\sqrt{\frac{V}{k \cdot l}}$ is determined and applied to the remaining $k \cdot l - 1$ elements of the pool. These three improvements are just some approaches to deal with such statistical defects. Further ideas and alternative solutions are given in [237, 42, 137, 44].

Regarding optimization of the Wallace method for GPUs, we follow the vectorization strategy in [42]. Vectorization is carried out along l direction, thus, every thread transforms k elements of the pool. Consequently, l threads work cooperatively to do a pass being the size of a thread block, and different thread blocks are working on different pools. Although the Wallace method directly produces normal random numbers and, thus, does not rely on a source of uniform random numbers, a uniform PRNG is required to deal with the above discussed statistical defects. Uniform random numbers are needed to resolve a stride and offset for the mixing of the pools and to randomly select A_i $(i = 1, \ldots, 4)$ but one uniform random number is enough to perform a whole pass.

8.4. Results

In the following, we first profile and benchmark the particular three normal PRNGs in subsection 8.4.1 to examine how well they are suited for GPUs. Afterwards, we compare the performance (in terms of generation rate) of the PRNGs in subsection 8.4.2 with each other to determine the best generator for our RODE implementation. To get a bigger picture of our PRNG implementations, we also compare to our results with state-of-the-art CPU and GPU random number libraries.

We do not investigate or implement any uniform PRNGs even though the Ziggurat method, rational polynomials, and the Wallace method depend on them but provide benchmark results for the uniform PRNGs and a corresponding discussion in subsection 8.4.2. The first two PRNGs require uniform random numbers as input to be transformed, the latter PRNG needs uniform random numbers to deal with statistical issues. Instead, we use cuRAND's [181, 183] XORWOW shift-register generator [154] as source of uniform random numbers. Except the binary rank test, all tests of Diehard [153] are passed by XORWOW. Therefore, its statistical properties such as long periods, a uniform distribution, and a hard predictability of the sequence are sufficient for our scenario. Furthermore, it offers high generation rates and has a small memory footprint. The benchmark results incorporate the runtimes of the uniform PRNG but not the setup times summed up in table 8.4. Thus, we do not identify the pure execution times of the transformation from uniform to normal distribution. Specifications for the Tesla M2090, the Tesla K40m, and the GTX 750 Ti used for the tests are given in table 2.1.

When it comes to the actual implementation of the presented PRNGs, the choice of the utilized GPU memory matters. Since the same lookup table of the Ziggurat can be used for all threads, it can either be stored in local memory or in shared memory. The same holds for the states of the uniform PRNG to improve the Wallace method.

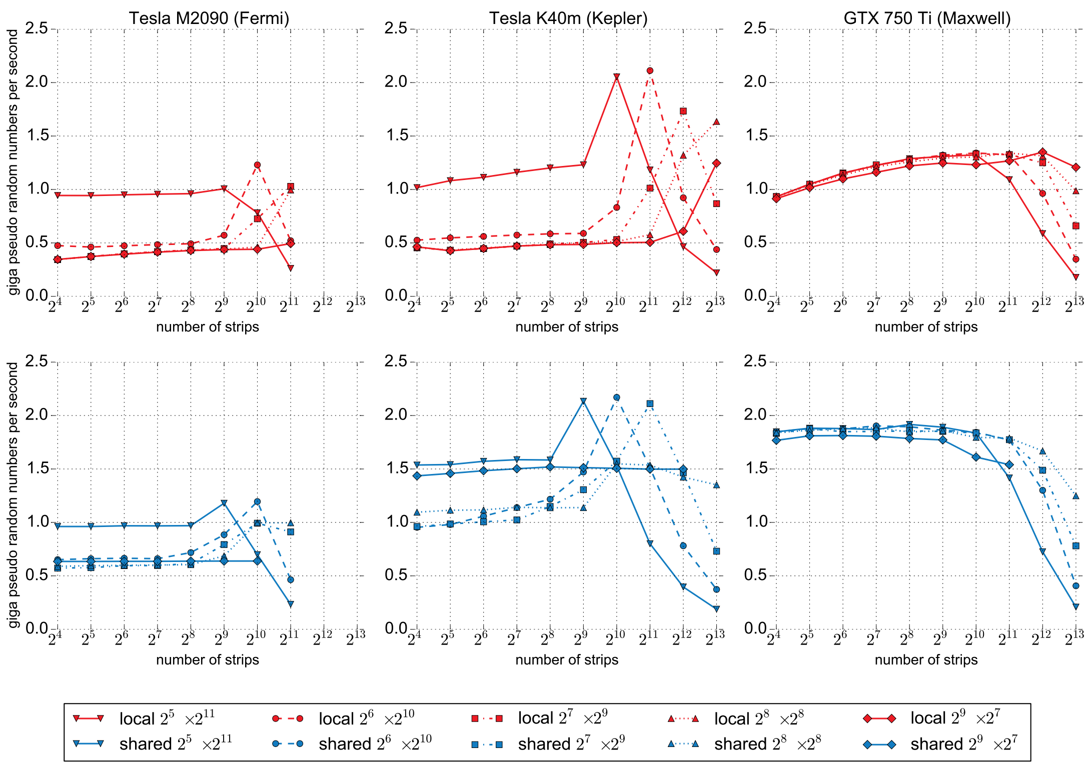


Figure 8.4.: Performance in GPRNs/s over number of strips N of different versions of the Ziggurat method measured on three different NVIDIA GPUs. First row of plots shows the local memory version colored in red, second row shows the shared memory version colored in blue. Different grid configurations ("threads per block" × "blocks per grid"), ranging from $2^5 \times 2^{11}$ to $2^9 \times 2^7$, are depicted by different lines. This plot is taken from our contribution [198].

Local memory is actually global memory with caching showing best performance if the same value is broadcasted to all threads. However, not all threads are dealing with the same strip of the Ziggurat and different threads are dealing with different k vectors of a pool. Hence, serialization may occur. Shared memory is fast on-chip memory with low latency but its performance can suffer from bank conflicts. We analyze both, a local and a shared memory version of the Ziggurat and the Wallace method. The pools of the Wallace method are always kept in shared memory, independent on where the states of the uniform PRNG are held. In the plots of subsection 8.4.1, results of the local memory version are colored in red and of the shared memory version in blue, respectively. Results for the investigated normal PRNGs can also be found in our work [198].

To test the PRNGs, 1GByte of single precision normal random numbers is generated during each run, corresponding to $2^{28} = 268{,}435{,}456$ `float` numbers. From a statistical

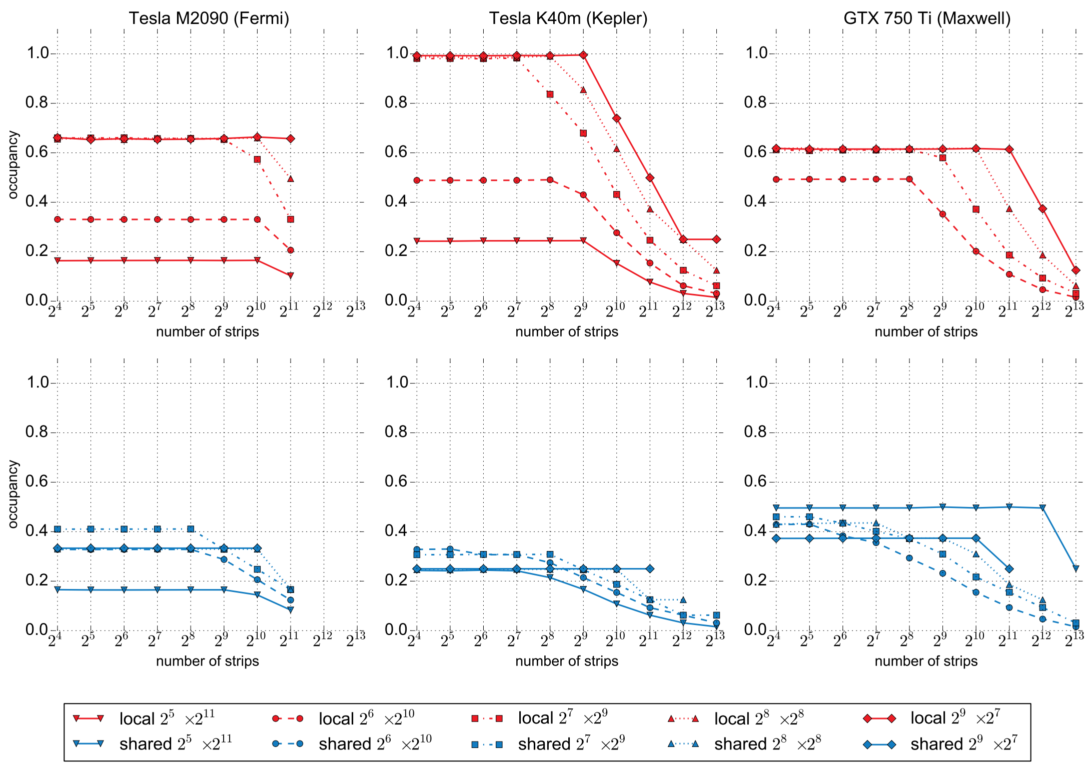


Figure 8.5.: Achieved occupancy over number of strips N of different implementations of the Ziggurat method measured on three different NVIDIA GPUs. Color coding, line captions, and markers are identical to figure 8.4. This plot is taken from our contribution [198].

point of view, it does not make sense to use double precision if the input is generated by cuRAND's XORWOW because it only delivers 32bit uniform random numbers. Thus, there are just up to 2^{32} distinct numbers possible to form. This limitation can easily be overcome by concatenating two successive 32bit random numbers to obtain one 64bit random number. We measure performance in giga (10^9) pseudo random numbers per second (GPRNs/s), use it as a unit for generation rate and put it on the ordinates of the following plots. Depending on the PRNG, we are either interested in the influence of the number of used strips or the grid configuration on the performance, so we assign these units to the abscissas. The grid configuration, or parallel setup, is denoted by "threads per block" × "blocks per grid" and ranges from $2^5 \times 2^{11}$ to $2^9 \times 2^7$ always leading to 2^{16} threads per grid. Therefore, every thread produces $2^{12} = \frac{2^{28}}{2^{16}}$ normal random numbers.

8.4.1. Evaluation of particular pseudo random number generators

To measure how well the memory/runtime trade-off works for the Ziggurat method, we vary the number of strips and assign them to the ordinates of figures 8.4 and 8.5. While

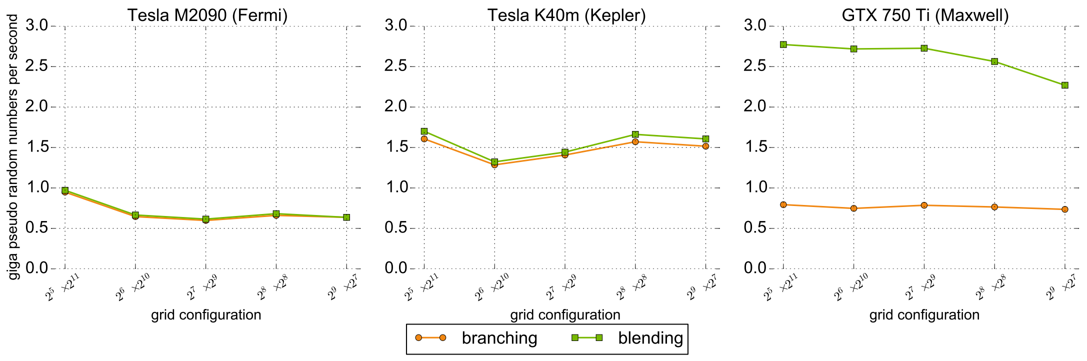


Figure 8.6.: Performance in GPRNs/s over grid configuration ("threads per block" × "blocks per grid") for rational polynomials measured on three different NVIDIA GPUs. The version with conditional statements is colored in green, the version using blending instead is colored in orange. This plot is taken from our contribution [198].

figure 8.4 illustrates the generation rate in PRNG/s for the Ziggurat method, figure 8.5 depicts the corresponding occupancies. In both figures, different grid configurations are indicated by different lines. Results originating from the local memory version (the lookup table is stored in local memory) are colored in red in the top rows of plots and results of the shared memory version are colored in blue in the bottom rows. Due to the limited size of shared memory, especially on older GPU architectures, some results are missing for some combinations of memory version, grid configuration, and number of strips. For example, there are no results for $N > 2^{11}$ strips on Fermi or for $N = 2^{13}$ strips using the shared memory version with grid configuration $2^9 \times 2^7$ on Kepler.

On Teslas M2090 and K40m, a single peak of performance is recognizable for almost all versions and configurations. This peak can also be retrieved on the GTX 750 Ti, but less significantly. High warp divergence limits the maximum performance if fewer strips than the peak performance configuration are used as the memory/runtime trade-off suggests. As figure 8.5 shows, the occupancy drops if more strips than the peak performance configuration are used being the limiter for higher strip numbers than the peak performance configuration. Until this point, the occupancy stays constant. In general, the occupancy of the shared memory version is lower than the one of the local memory version due to its higher shared memory consumption. However, in most cases, the shared memory version performs better than the local memory version. On the Tesla M2090, best performance is obtained with the local memory version using 1024 strips and a grid configuration of $2^6 \times 2^{10}$ leading to 1.23 GPRNs/s. The shared memory version gives the best performance on the Tesla K40m and the GTX 750 Ti with 2.17 GPRNs/s on Kepler (1024 strips and grid configuration $2^6 \times 2^{10}$) and 1.91 GPRNs/s on Maxwell (256 strips and grid configuration $2^5 \times 2^{11}$).

Different to the Ziggurat method, we are interested how the performance of the rational polynomials is influenced by the grid configuration. Thus, the grid configuration is

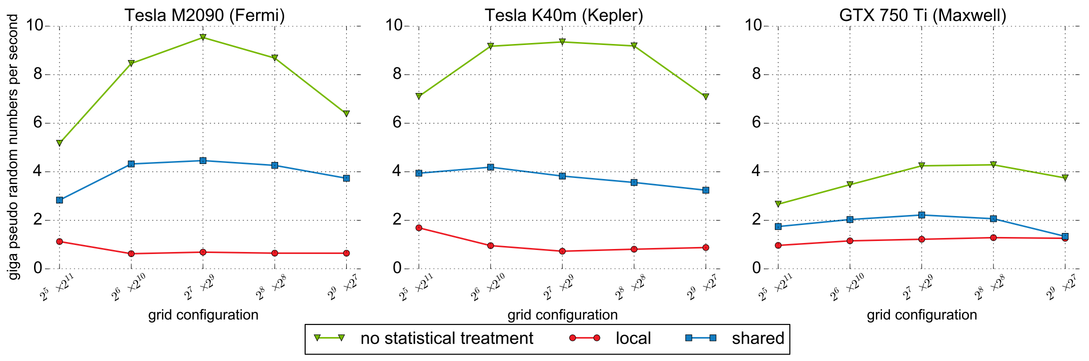


Figure 8.7.: Performance in GPRNs/s over grid configuration ("threads per block" × "blocks per grid") for the Wallace method measured on three different NVIDIA GPUs. Like in figures 8.4 and 8.5, the local and shared memory versions are colored in red and blue, respectively. Results from a shared memory version without any treatment of the statistical flows is plotted in purple. This plot is taken from our contribution [198].

assigned to abscissas and performance in GPRNs/s to ordinates of figure 8.6. The version using conditional statements is represented by green lines, the more advanced version avoiding branching by orange lines.

In contrast to the Ziggurat method, there is only a minor influence of the grid configuration and of the occupancy on the performance of rational polynomials. The version using blending always performs better then the default version using branching but the difference varies much depending on the utilized GPU. On the Teslas M2090 and K40m, there is only a span of up to 1.03× (grid configuration $2^8 \times 2^8$) and 1.06× (grid configuration $2^9 \times 2^7$), respectively. The GTX 750 Ti is much more sensitive to this kind of optimization: An improvement of up to 3.64× (grid configuration $2^6 \times 2^{10}$) can be measured when using the blending version. Obviously, the overhead to synchronize and join diverged threads of a warp is much bigger on Maxwell than on Fermi and Kepler. Best achieved performance is 0.97 GPRNs/s on the Tesla M2090, 1.70 GPRNs/s on the Tesla K40m, and 2.77 GPRNs/s on the GTX 750 Ti each using grid configuration $2^5 \times 2^{11}$.

The axes assignment in figure 8.7, showing the generation rates for the Wallace method, is the same like the results of the rational polynomials. Results of the local memory version keeping the states of the uniform PRNG in cached global memory are colored in red, the shared memory version in colored in blue. Both versions utilize a uniform PRNG to deal with the statistical weaknesses of the Wallace method as explained in section 8.3. In addition, we added in purple results of a version without any special statistical treatment. The normal random numbers obtained by this version are improper to be used in any application but it shows how compelling the idea of the Wallace method of using a linear orthogonal operator can be.

Similar to the Ziggurat method, there is a single peak of performance for all versions on all GPUs. Using more threads per block than the peak performance configuration lowers the occupancy and, thus, the performance. Pools are always kept in shared memory and in our parallelization approach for GPUs, the pool size depends on the number of threads per block. The shared memory version always outperforms the local memory version leading to a highest generation rate of 4.46 GPRNs/s on the Tesla M2090, 4.18 GPRNs/s on the Tesla K40m, and 2.21 GPRNs/s on the GTX 750 Ti. Optimal grid configurations are $2^7 \times 2^9$ on the Teslas and $2^6 \times 2^{10}$ on the GTX 750 Ti. Comparing the results of the local (red) and shared (blue) memory version with the purple line demonstrates the overhead of the statistical treatment: Omitting it leads to an acceleration of 2.14× on Tesla M2090, of 2.44× on Tesla K40m, and of 1.93× on GTX 750 Ti.

On a first look, the orthogonal linear operator A to update the pools seems expensive but it is chosen as cheap as possible in terms of number of mathematical operations. $A_1, \dots, A_4$ only cause seven additions and one multiplication per K vector. Hence, the Wallace method neither has to be computationally expensive nor compute-bound as verified by table 8.3 (see row "Wallace").

method	implementation	Tesla M2090	Tesla K40m	GTX 750 Ti
Ziggurat	local	0.02%	0.02%	0.38%
	shared	0.04%	0.02%	0.58%
rational polynomial	branching	0.29%	0.08%	1.65%
	blending	0.88%	0.26%	17.87%
Wallace	no treatment	0.50%	0.13%	1.44%
	local	0.06%	0.01%	0.42%
	shared	0.23%	0.04%	0.70%

Table 8.3.: Achieved single precision FLOPS efficiency (achieved FLOPS rate over theoretical peak FLOPS rate) of different normal PRNGs (rows) implemented in different versions on three different NVIDIA GPUs (columns). Grid configuration (and for the Ziggurat method, number of strips N) leading to best performance is used (see results of the particular PRNGs). This table is taken from our contribution [198].

Table 8.3 lists the achieved single precision FLOPS efficiencies of the Ziggurat method, rational polynomials, and the Wallace method. Configurations leading to best performance for the particular PRNGs are applied, cf. peaks in figures 8.4, 8.6, and 8.7 to obtain the efficiencies. None of the three normal PRNGs is compute-bound on any of the three GPUs whose theoretical maximum FLOPS rates can be found in table 2.1. The FLOPS rates are actually quite low which is not a problem at all because none of the discussed normal PRNGs is computationally expensive and still, the generation rates are very high. All PRNGs are also not memory- but latency-bound. Hence, the PRNGs of this chapter benefit from GPU architectures dealing well with latency-bound problems by better instruction scheduling and lower instruction latencies. Such an architecture is Maxwell's SMM [178] design and that is the reason why the mid-range consumer GPU GTX 750 Ti can keep up with the theoretically much more powerful Teslas relying on less sophisticated architectures [177].

method	implementation	Tesla M2090	Tesla K40m	GTX 750 Ti
Ziggurat	local	24.063ms	61.889ms	52.176ms
	shared	24.042ms	26.801ms	34.417ms
rational polynomial		22.089ms	18.332ms	31.631ms
Wallace	local	3617.30ms	3193.11ms	1311.23ms
	shared	3619.79ms	4015.01ms	2841.78ms

Table 8.4.: Setup times to initialize the states of different normal PRNGs (rows) on three different NVIDIA GPUs (columns). Parameters leading to best performance are used, cf. peaks in figures 8.4, 8.6, and 8.7. This table is taken from our contribution [198].

Finally, table 8.4 lists the setup times of all three normal PRNGs essential before random numbers can be fabricated. For the Ziggurat method, setup times include the computation of the Ziggurat approximation according to subsection 8.1.3 and the initialization of the uniform PRNGs. The initialization of the uniform PRNGs is also compulsory for rational polynomials because both PRNGs are just transformation operations. Besides the initialization of the uniform PRNGs to improve statistical properties, the Wallace method has to create an original pool of normal random numbers. For rational polynomials, the state initialization is identical for both versions (branching and blending), hence only one value is indicated. Since the shared memory version and the version without any special treatment for statistical defects of the Wallace method handle the same data structures in the same kinds of memory, only one setup time is given. The setup times for the Ziggurat method and rational polynomials are negligible. This does not hold for the Wallace method because the generation of the original pools composed of $\nu = k \cdot l$ normal random numbers takes some time.

8.4.2. Performance comparison of pseudo random number generators

The particular results of the normal PRNGs are used to draw a comparison between them. In addition, we match the outcomes with generation rates obtained from other well-established random number libraries for CPU and GPU. For the Ziggurat method (local memory version on the Tesla M2090, shared memory version on the Tesla K40m and the GTX 750 Ti), rational polynomials (version using blending), and the Wallace method (shared memory version) we use those parameters (number of strips, grid configuration) achieving best performance on the corresponding GPUs. These parameters are the same as for tables 8.3 and 8.4.

To make a fair comparison with a state-of-the-art random number generator library for the CPU, we benchmark the capabilities of two Intel Xeon E5-2680 v2 representing a high-end CPU system. The Math Kernel Library (MKL) is used as CPU library. Since it is hard to find any absolute generation rates for the MKL, we wrote our own CPU code to benchmark the MKL and highly optimized it by using AVX vectorization and

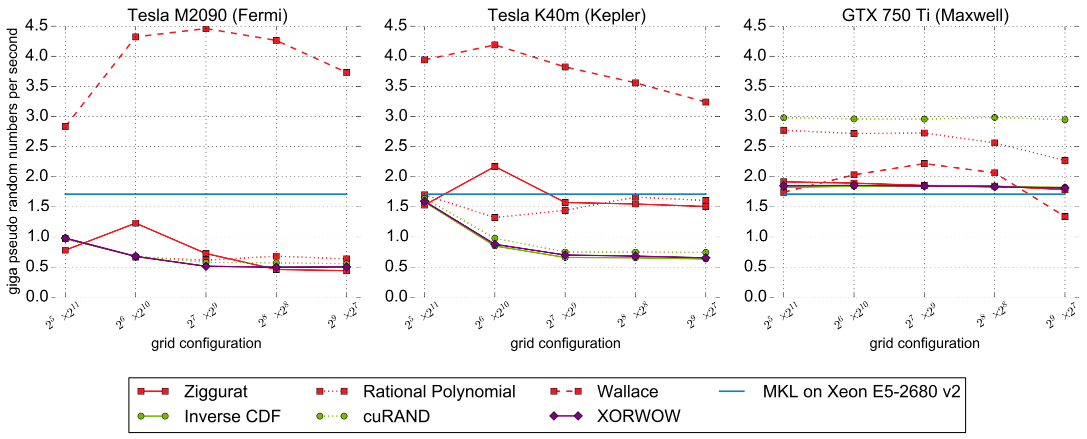


Figure 8.8.: Performance in GPRNs/s over grid configuration ("threads per block" $\times$ "blocks per grid", ranging from $2^5 \times 2^{11}$ to $2^9 \times 2^7$) of various PRNGs. Three different NVIDIA GPUs are used for the GPU PRNGs. The CPU reference colored in blue is taken on two Intel Xeon E5-2680 v2 using the MKL. The GPU references Box/Muller and `normcdfinvf()` are colored in green. Best performing parametrizations of the normal PRNGs presented in sections 8.1–8.3 are colored in red. Finally, the pure uniform random number generation result from XORWOW is colored in purple. This plot is taken from our contribution [198].

OpenMP for multithreading with 20 threads. Sources such as [111] only offer speed-up factors, e.g. in comparison to the C function `rand()`, without mentioning absolute values for the reference. Our CPU code bases on the MKL 11.3 and is translated by the Intel C++ Compiler (ICC) 16.0. According to our experience, the combination of MKL's 59bit multiplicative congruential generator (uniform random number generation) and inverse CDF of the Gaussian function (transformation to normal distribution) delivers best performance on the two Intel CPUs with 1.71 GPRNs/s.

Besides the XORWOW generator, cuRAND also provides normal random number generators. Hence, we use cuRAND as state-of-the-art GPU library, namely its implementation of the Box/Muller method [38] for transformation to normal distribution. Furthermore, CUDA itself offers a built-in approximation of the inverse normal CDF similar to our rational polynomials called `normcdfinvf()`. Both, the Box/Muller method and `normcdfinvf()` are transformation operations relying on uniform random numbers supplied by XORWOW. Speaking of XORWOW, its performance is also added to figure 8.8 to get an impression how big its share in total runtime of the normal PRNGs is.

Figure 8.8 shows the performance results of all mentioned PRNGs drawn by different lines using the fastest results from subsection 8.4.1. GPRNs/s is plotted on ordinates, grid configuration on the abscissas. On the Teslas, the Wallace method clearly shows best performance exceeding the second best candidate, the Ziggurat method, by factors of 3.62× (Tesla M2090) and 1.93× (Tesla K40m). Rational polynomials provide worst performance of the three uncommon methods on these two GPUs. All methods presented in sections 8.1–8.3 (red lines) always achieve at least similar performance to the GPU libraries (green lines) and the Wallace method outperforms them by 4.53× (Tesla M2090) and 2.55× (Tesla K40m). The performance of cuRAND's Box/Muller and CUDA's `normcdfinvf()` seems to be limited by XORWOW (purple lines) because generation rates are almost identical. The same holds for rational polynomials on the Tesla M2090.

On the GTX 750 Ti, the relation between the results of the PRNGs is totally different. Best performance is carried out by cuRAND's Box/Muller implementation being 1.08× faster than the second best method, the rational polynomials. Hence, rational polynomials have the highest generation rates of the three methods presented in this chapter, 1.25× faster than the Wallace method and 1.44× faster than the Ziggurat method.

A direct comparison between our CPU implementation with our best performing GPU implementations shows that the GPUs exceed the CPUs by factors of 2.61× (Tesla M2090), 2.45× (Tesla K40m), and 1.74× (GTX 750 Ti), respectively. These are common factors being expected when comparing highly-optimized CPU with highly-optimized GPU code.

In many cases, our implementations of the Ziggurat method and rational polynomials perform better than the pure XORWOW generator. This also holds for cuRAND's Box/Muller implementation on the GTX 750 Ti. At a first sight, this is quite surprising because all these methods require a uniform random number from XORWOW before they can transform it to normal distribution. Thus, the performance of the Ziggurat method, rational polynomials, and cuRAND's Box/Muller should be actually limited by the performance of the uniform PRNG. We assume that this discrepancy can be explained by a higher degree of instruction level parallelism stemming from the fusion of uniform random number generation and transformation to normal distribution in one single kernel, what we do.

Now, we have several normal PRNGs at hand whose characteristics fit very well on GPUs: The memory/runtime trade-off allows to adapt the Ziggurat method to be efficiently implemented on GPUs and to tailor it for particular GPUs. GPUs offer high computational performance, thus, also methods such as (piecewise) rational polynomials with high nominator and denominator degree to approximate the inverse normal CDF utilizing blending deliver high performance. The Wallace method benefits from this computational performance, too, even if special measurements have to be taken to deal with statistical pitfalls. With such high-performance normal PRNGs available, we now proceed to the parallel realization of the OU process, the second building block of our RODE solver layout, in chapter 9 which incorporates normal random numbers.

9. Building block 2: Ornstein-Uhlenbeck process

The OU process (7.4) is, besides the *geometric Brownian motion*, one of the simplest but also important stochastic processes. It is incorporated in RODEs, thus, its realization is essential for our general RODE solver but also for numerous other stochastic applications. The solution of the OU process (7.5) is a strictly sequential formula where the next element $O_{t_{n+1}}$ is determined from O_{t_n}. This property makes it hard to parallelize the solution of the OU process, especially when multiple levels of parallelism should be exploited as we plan to do with our GPU implementation.

In this chapter, we first demonstrate how to map the explicit formula to realize a path of the OU process to the operation of building prefix sums in section 9.1. For the various types of prefix sums, there are successful strategies for parallelization. Section 9.2 illustrates how to use these strategies to parallelize the OU process also applicable but not limited to GPUs. Here, our contribution is not the parallelization of prefix sums but in the mapping of the OU process to them. Finally, we measure in section 9.3 the number of generated OU process elements per time on the three NVIDIA GPUs already used throughout part III. Identifiers of chapter 7 are utilized in the following.

9.1. From the Ornstein-Uhlenbeck process to prefix sum

To find a strategy to realize multiple elements of the OU process (7.5) in parallel, we first have a look how the OU process evolves. We are interested in the dependence of the i-th element $O_{t_{n+i}}$ on the first element O_{t_n}. If there is a direct relation without requiring $O_{t_{n+i-1}}, \ldots, O_{t_{n+1}}$, then an independent calculation of distinct elements is possible.

$$\begin{aligned}
O_{t_{n+1}} &\overset{(7.5)}{=} \mu_X O_{t_n} + \sigma_X n_1^{(1)} \\
O_{t_{n+2}} &= \mu_X O_{t_{n+1}} + \sigma_X n_1^{(2)} = \\
&= \mu_X \left(\mu_X t_n + \sigma_X n_1^{(1)}\right) + \sigma_X n_1^{(2)} = \\
&= \mu_X^2 O_{t_n} + \sigma_X \left(\mu_X n_1^{(1)} + n_1^{(2)}\right) \\
O_{t_{n+3}} &= \mu_X O_{t_{n+2}} + \sigma_X n_1^{(3)} = \\
&= \mu_X \left(\mu_X O_{t_{n+1}} + \sigma_X n_1^{(2)}\right) + \sigma_X n_1^{(3)} = \\
&= \mu_X \left(\mu_X \left(\mu_X O_{t_n} + \sigma_X n_1^{(1)}\right) + \sigma_X n_1^{(2)}\right) + \sigma_X n_1^{(3)} =
\end{aligned}$$

$$= \mu_X^3 O_{t_n} + \sigma_X \left(\mu_X^2 n_1^{(1)} + \mu_X n_1^{(2)} + n_1^{(3)} \right)$$

$$\vdots = \vdots$$

$$O_{t_{n+i}} = \mu_X^i O_{t_n} + \sigma_X \sum_{k=1}^{i} \left(\mu_X^{i-k} n_1^{(k)} \right) \tag{9.1}$$

$n_1^{(i)}$ denotes the i-th normal random number of $\mathcal{N}(0,1)$. In the final general evolution step (9.1), the first summand $\mu_X^i O_{t_n}$ just depends on i and the very first element O_{t_n}. Furthermore, $\sigma_X = \sqrt{\frac{c\tau}{2}(1-\mu_X^2)}$ is constant for a given timestep length h. Ignoring μ_X^{i-k} for a moment simplifies the sum in the second summand to $\sum_{k=1}^{i} n_1^{(k)}$[1]. This simplified operation is exactly a *prefix sum* or *scan* [34].

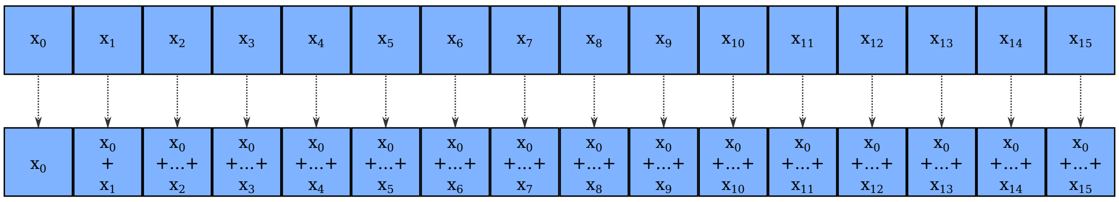


Figure 9.1.: Illustration of the inclusive prefix sum. The sum over the first $i+1$ continuous elements of the input sequence including x_i are assigned to x_i of the output sequence, i.e. $x_i = \sum_{k=1}^{i} x_k$.

Basically, there are two types of prefix sums, just slightly differing from each other: *exclusive prefix sum* (also called *prescan*) and *inclusive prefix sum*. Exclusive prefix sums assign the sum of the first i continuous elements of a given sequence of numbers $x_0, \ldots, x_{n-1}$ to the i-th element. This sum goes up to x_i but not x_i itself, i.e. $x_i = \sum_{k=0}^{i-1} x_k$. Inclusive prefix sums determine the sum of the first $i+1$ elements of the sequence including x_i, i.e. $x_i = \sum_{k=1}^{i} x_k$. Figure 9.1 depicts an inclusive prefix sum with a sequence length of 16. Having an inclusive prefix sum, the corresponding exclusive prefix sum is easily obtained by shifting all elements of the inclusive prefix sum one position to the right and setting the very first element x_0 to 0.

The general evolution step of the OU process (9.1) requires inclusive prefix sums, thus, we do not consider exclusive prefix sums in the following and mean inclusive prefix sum when mentioning the terms prefix sum or scan. A modification of the default prefix sum is necessary to be usable in the OU process, namely the re-integration of μ_X^{i-k} in the sum. Hence, we get $x_i = \sum_{k=1}^{i} \left(\mu_X^{i-k} x_k \right)$.

To keep pseudocodes and figures clear and simple, we restrict ourselves to realizations of the OU process with powers of 2 elements in the explanations. However, all methods can be generalized to any arbitrary number of elements and also our implementation does not suffer from this limitation.

[1]The normal random numbers in the sum are already computed in the previous building block step, cf. chapter 8.

9.2. Parallel prefix sum

The major advantage of mapping the OU process to a prefix sum is the possibility to exploit existing parallelization approaches of scans. We can adapt these approaches to parallelize the realization of the OU process. Parallel versions of the prefix sum as described in [126] are called *parallel prefix sum* or *parallel scan*, respectively. There are also parallel strategies targeting GPUs, cf. [100, 99, 216]. We use these strategies to parallelize the OU process.

Algorithm 7 Up-sweep phase

```
1: procedure UP
2:     for $d = 1$; $d \leq \log_2(n)$; $d$++ do
3:         for $i = 0$; $i < \frac{n}{2^d}$; $i$++ do
4:             $x_{(i+1)2^d-1} \leftarrow x_{(i+1)2^d-1} + x_{\left(i+\frac{1}{2}\right)2^d-1}$
```

Algorithm 8 Down-sweep phase

```
1: procedure DOWN
2:     for $d = \log_2(n) - 1$; $d \geq 0$; $d$-- do
3:         for $i = 0$; $i < \frac{n}{2^d} - 1$; $i$++ do
4:             $x_{\left(i+\frac{3}{2}\right)2^{d+1}-1} \leftarrow x_{(i+1)2^{d+1}-1} + x_{\left(i+\frac{3}{2}\right)2^{d+1}-1}$
```

Parallel implementations not using more operations in total than a serial implementation are called *work efficient*. One possible and the most popular work efficient parallel version of inclusive prefix sums subdevides the task in an *up-sweep phase* and a *down-sweep phase*. Both phases have a tree-like structure. The up-sweep phase is described by algorithm 7 and visualized by figure 9.2. It is the same operation like building the sum over all elements of the sequence in parallel but the intermediate results are essential for the down-sweep phase and, thus, for the determination of the prefix sum. Algorithm 8 expresses the down-sweep phase which is depicted by figure 9.3. Simply spoken, the down-sweep phase updates all those values of the sequence which are not already correct after the up-sweep phase by adding the missing terms. The particular iterations of the inner loops of algorithms 7 and 8, i.e. the loops over i, are independent from each other, hence, they can be executed in parallel. A thread sums up at least 2 elements per outer iteration, i.e. loop over d. Since the stopping criterion of the inner loop depends on the index of the outer loop, not all threads are doing work all the time. Actually, during the up-sweep phase, more and more threads get idle while during the down-sweep phase, the number of busy threads increases. This leads to the tree-like structure and $\mathcal{O}(\log(n))$ parallel operations are necessary instead of $\mathcal{O}(n)$ serial operations. Further details concerning standard parallel prefix sum on GPUs are explained in [100].

For now, we just illustrated how the parallelization for the prefix sum works but for the realization of the OU process, powers of μ_X have to be multiplied at correct positions to regain the factor μ_X^{i-k} omitted so far. During the up-sweep phase, $\mu_X^{2^d}$ has to be

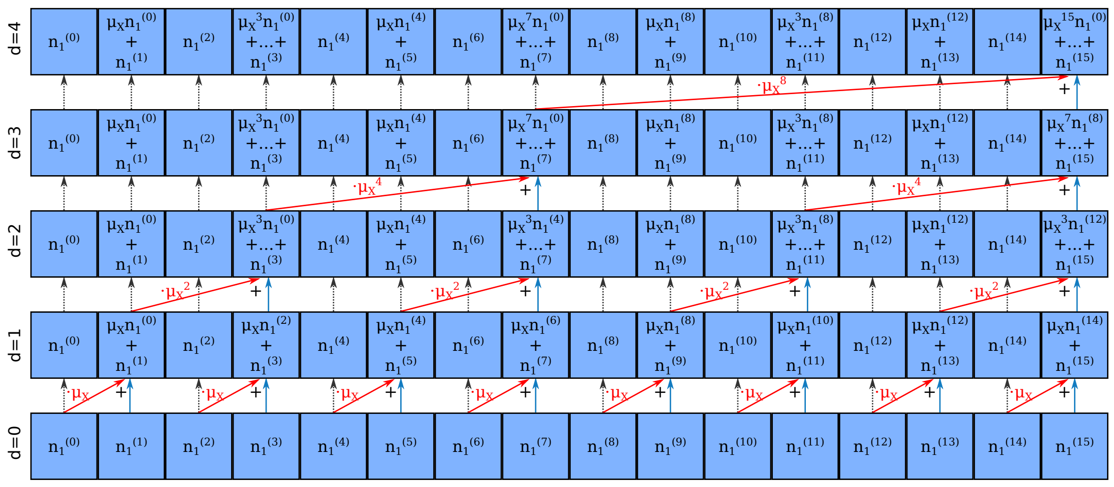


Figure 9.2.: Illustration of the up-sweep phase according to Algorithm 7 with a sequence length of 16. Every row represents the state of the sequence after one iteration of the outer loop. Blue and red arrows (ignoring the red powers of μ_X) indicate an addition of 2 elements of the sequence, gray dotted arrows mark unaltered elements. To extend the parallel prefix sum for the OU process, powers of μ_X have to be multiplied to the correct summands as expressed by the red factors. At least one addition is assigned to one thread. This figure is taken from our contribution [199].

multiplied to every $x_{(i+1)2^d-1}$. During the down-sweep phase, $\mu_X^{2^d}$ has to be multiplied to every $x_{(i+1)2^{d+1}-1}$. In figures 9.2 and 9.3, this augmentation of the default parallel prefix sum is represented by red powers of μ_X. Re-interpreting the parallel prefix sum makes it a quasi parallel-in-time method [81] for the OU process because the OU process evolves in time and we realize multiple elements of it simultaneously.

To reduce the number of accesses to slow global memory, a chunk of normal random numbers originating from building block 1 is loaded in shared memory before a path of the OU process is computed. This measurement increases performance because multiple elements are modified multiple times during the up- and down-sweep phase. If every thread of a thread block sums up 2 elements, only a fraction of the available shared memory is utilized. Currently, a thread block can consist of up to 2^{10} threads, thus, 2^{10} threads per block times 2 elements per thread times 8 bytes for every element in double precision results in only 16KByte memory consumption. Today's GPU architectures offer at least 48KByte of shared memory per multiprocessor (cf. table 2.1), thus, every thread could also deal with more than 2 elements. On the one hand, this action leads to an increase in computational intensity per thread being preferable on GPUs due to their high $\frac{\text{FLOP}}{\text{byte}}$ ratio. On the other hand, shared memory consumption per thread block is increased directing to less active blocks per multiprocessor, thus, lowering occupancy. Figure 9.4 shows two different setups of the parallel realization of the OU process: Every thread block consists of 2 threads. In subfigure 9.4(a), each thread deals with 2 elements

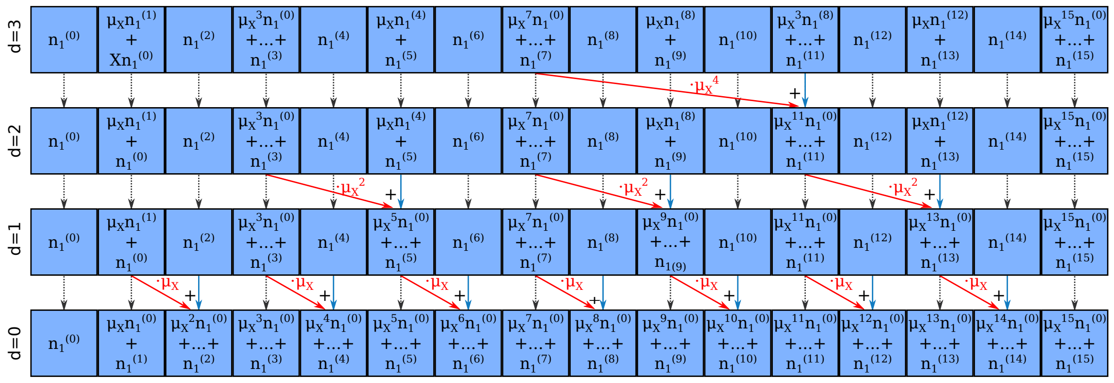


Figure 9.3.: Illustration of the down-sweep phase according to Algorithm 8 with a sequence lenth of 16. Row indexing, arrow coloring, the meaning of the powers of μ_X, and the principal assignment of mathematical operations to threads are the same as for figure 9.2. This figure is taken from our contribution [199].

requiring 4 thread blocks to handle 16 elements. If every thread deals with 4 elements as depicted in subfigure 9.4(b), 2 thread blocks process 16 elements.

In most cases, more elements of an OU process path are realized than fit in a single shared memory. Hence, multiple thread blocks have to be started. Our GPU implementation of the OU process uses three different kernels, executed consecutively, to enable a correct calculation:

- `scanInclusiveOUKernel()`:
  Realizes up- and down-sweep phase specified by algorithms 7 and 8 in a straight forward way.

- `scanOUFixKernel()`:
  Adds the results from the previous $i-1$ blocks to the i-th block. This fix is only necessary if more than one block is launched.

- `realizeOUProcessKernel()`:
  Extends the modified parallel prefix sum to the complete parallel OU process. σ_X is multiplied to the result of the modified parallel prefix sum and the addition of $\mu_X^i O_{t_n}$ eventually gives the final general evolution step (9.1).

There are much more sophisticated versions of parallel prefix sum on GPUs than the strategy explained so far. The usage of *warp shuffle functions* [69, 244] for example, available since the Kepler architecture for NVIDIA GPUs, allow more refined approaches [145]. However, this idea only works if the input elements of the prefix sum are integers and it only leads to a speed-up if a short bit representation of the elements is possible.

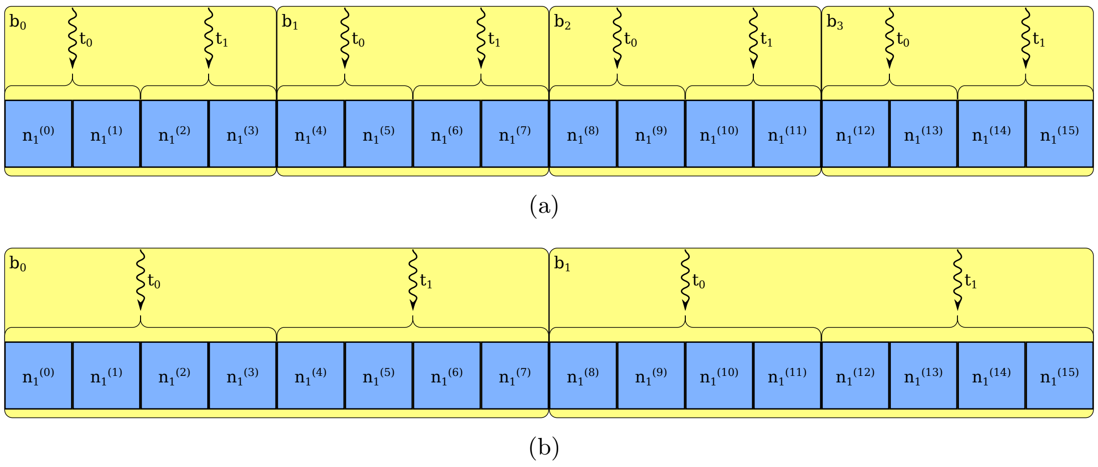


Figure 9.4.: Two versions of assignment of threads to elements to process. In both cases, thread blocks consist of 2 threads (indicated by curly arrows). Subfigure 9.4(a) depicts the case of 2 elements per thread resulting in 4 thread blocks for a sequence of 16 elements. For the same number of elements, 2 thread blocks are necessary if every thread deals with 4 elements as illustrated by subfigure 9.4(b). Both subfigures are taken from our contribution [199].

9.3. Results

Figure 9.5 shows the benchmark results of our parallel implementation of the OU process basing on parallel prefix sum. Benchmarks are conducted on the same three GPUs as used for benchmarking in the previous chapter 8: Tesla M2090, Tesla K40m, and GTX 750 Ti. In all runs, 2^{26} elements of the OU process are realized in parallel. We are interested in the influence of the number of elements per thread on performance, thus, we assign this value to the ordinates. The rate of realized elements of the OU process is used as performance indicator measured in giga (10^9) realizations of the OU process per second (GROUPs/s) assigned to the abscissas. Different lines represent different applied “threads per block” parallel setups ranging from 2^7 to 2^{10}. Red lines represent single precision results, double precision outcomes are colored in blue. Runtimes refer to the realization of the OU process only including all three kernels `scanOUFixKernel()`, `scanOUFixKernel()`, and `realizeOUProcessKernel()` but do not incorporate random number generation.

For many combinations of GPU model, elements per thread, number of threads per block in one dimension, and number of blocks per grid in one dimension, there are no results because hardware limitations avoid a successful execution. So for instance, realizing 2^{26} elements of the OU process using 2^7 threads per block and every thread handling 2^3 elements would require $\frac{2^{26}}{2^7 \cdot 2^3} = 2^{16}$ blocks per grid. Due to its compute capability, the Tesla M2090 only supports $2^{16} - 1$ blocks per grid per dimension. Or using 2^9 threads per block and every thread handling 2^4 elements in double precision

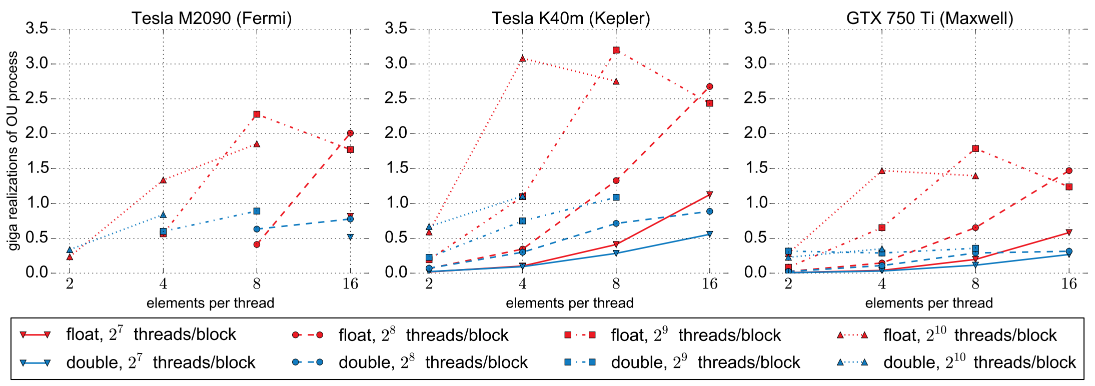


Figure 9.5.: Performance in GROUPs/s over number of elements per thread of our parallel OU process implementation on three different NVIDIA GPUs. Single precision results are colored in red, double precision results are colored in blue. Different “threads per block” configurations ranging from 2^7 to 2^{10} are depicted by different lines. Due to hardware limitations, there are not results for all combinations of GPU, elements per thread, and number of threads per block. This plot is taken from our contribution [199].

would require $2^{10} \cdot 2^4 \cdot 8\text{Bytes} = 128\text{KByte}$ of shared memory. None of the utilized GPUs offers that amount of shared memory per multiprocessor.

Even though there are only few results per line, one single peak of performance is recognizable for almost all results in single precision. For example, the 2^{10} threads per block version shows best performance on the Tesla K40m when assigning 4 elements to every thread or on the GTX 750 Ti, 16 elements per thread is the best choice when running 2^8 threads per block. Low computational intensity is the reason for worse performance when using less elements per thread than the peak performance configuration. Low occupancy due to higher shared memory consumption is the reason for worse performance when using more elements per thread than the peak performance configuration. Using more elements per thread almost always leads to better performance when executing operations in double precision. On the Tesla K40m, 4 elements per thread and 2^{10} threads per block result in the best double precision performance on this particular GPU. For all other combinations of GPU and precision, the best performing configuration is always 8 elements per thread and 2^9 threads per block. Peak performance values of the Tesla M2090 are 2.28 GROUPs/s and 0.89 GROUPs/s for single and double precision, respectively, with according occupancy 1 and 0.33. The highest realization rates are accomplished on the Tesla K40m with 3.20 GROUPs/s in single (occupancy 0.75) and 1.11 GROUPs/s in double precision (occupancy 0.5). The corresponding performance of the GTX 750 Ti is 1.78 GROUPs/s (occupancy 1) and 0.36 GROUPs/s (occupancy 0.5), respectively. Configurations for all utilized GPUs leading to best performance of the OU process are clearly listed in table 12.1.

The major bottleneck of the parallel OU process is its tree-like algorithmic structure. Hence, approximately half of the threads are idle during execution. However, our approach of mapping the OU process to prefix sum and exploiting its parallelization strategies is the first successful attempt to parallelize this stochastic process also applicable to other parallel computing devices and architectures. The next building block determines different averaged values of continuous sub-sequences of a realized OU process path in parallel and is presented in the next chapter 10.

10. Building block 3: Averaging

To complete the transition from fine timestepping with timestep size δ to coarse timestepping with timestep size h, different kinds of averaged values of the elements of the OU process (cf. chapter 9) have to be determined. Hence, averaging forms the third building block of our RODE solver pipeline. The averager type depends on the actual numerical solver for the RODE. Even if we start with averaged values for the KT model, we see that this leads to the determination of general, problem-independent averages. These averages are not limited to the solution of RODEs but can also be assigned to many other domains. Averaging is the only memory-bound building block of our RODE solver. We continue using the identifiers from chapter 7.

In section 10.1 we show that the calculation of single and double averaged values (7.9b) and (7.11b) simply leads to the computation of a (modified) universal averaged value. The same holds for the approximation of multi-integrals by Riemann sums for the K-RODE-Taylor schemes (7.18) demonstrated in section 10.2. We conclude this chapter with section 10.3 containing benchmarks and profilings of our averager implementations for GPUs highlighting the performance of this particular building block.

10.1. Single & double averaging

The averaged Euler scheme (7.10) incorporates the single averaged value (7.9b). Rewriting $\bar{G}^{(1)}_{h,\delta}(t = t_n)$ and applying the KT model's $G(t)$ gives

$$\begin{aligned}
\bar{G}^{(1)}_{h,\delta}(t_n) &\overset{(7.9b)}{=} \frac{1}{M}\sum_{j=0}^{M-1} G(t_n + j\delta) \\
&\overset{(7.8a)}{=} \frac{1}{M}\sum_{j=0}^{M-1} -O_{t_n+j\delta}\begin{pmatrix} 1 \\ 2\xi_g\omega_g - 1 \end{pmatrix} \\
&= -\frac{1}{M}\underbrace{\begin{pmatrix} 1 \\ 2\xi_g\omega_g - 1 \end{pmatrix}}_{\text{KT model-specific}}\sum_{j=0}^{M-1} O_{t_n+j\delta}.
\end{aligned} \tag{10.1}$$

$\begin{pmatrix} 1 \\ 2\xi_g\omega_g - 1 \end{pmatrix}$ is specific to the utilized KT model but the remainder of (10.1) is just the average of M continuous elements of the OU process. Depending on M, this average is by far the computationally most expensive part of $\bar{G}^{(1)}_{h,\delta}(t)$.

We come to a similar conclusion when investigating the averaged Heun scheme (7.12): It incorporates the double averaged value (7.11b). Applying the KT model to $\bar{G}^{(2)}_{h,\delta}(t = t_n)$ leads to

$$\begin{aligned}\bar{G}^{(2)}_{h,\delta}(t_n) &\overset{(7.11b)}{=} \frac{2}{M^2}\sum_{j=0}^{M-1}(M-j)G(t_n+j\delta)\\ &\overset{(7.8a)}{=} \frac{2}{M^2}\sum_{j=0}^{M-1} -(M-j)O_{t_n+j\delta}\begin{pmatrix}1\\ 2\xi_g\omega_g-1\end{pmatrix}\\ &= -\frac{2}{M^2}\underbrace{\begin{pmatrix}1\\ 2\xi_g\omega_g-1\end{pmatrix}}_{\text{KT model-specific}}\sum_{j=0}^{M-1}(M-j)O_{t_n+j\delta}.\end{aligned} \tag{10.2}$$

Again, the KT model-specific factor can be separated resulting in an average value of M continuous elements of the OU process. In contrast to the average value of the averaged Euler scheme, the average value of the averaged Heun scheme has to be extended by a factor $M - j$ but this factor is still independent from the actual model. Building the average value is the computationally most expensive part of $\bar{G}^{(2)}_{h,\delta}(t = t_n)$.

Actually, we also have to analyze the averaged values (7.9a) and (7.11a). Since for the KT model $g(t) \overset{(7.8b)}{:=} 1$, also the averaged values become 1 and, thus, can be neglected.

Parallelizing the averagers is trivial due to the associativity of the add operation. Since there are in general much more elements to be summed up than working threads, a parallel averager works in two steps exemplified by figure 10.1: First, an exclusive equally-sized segment of the values to be averaged is assigned to every thread which computes the sum of the segment (multiplying each value with $M - j$ if double averages are requested). Second, the global sum of all segments is determined in a tree-like manner, also called a *reduce* operation. Finally, the global sum has to be divided by M or $-\frac{M^2}{2}$ for single or double averages, respectively. Regarding numerical stability, it can make sense to not perform the division at the end of the averaging process but already sooner.

10.2. Tridiagonal averaging

The simplified multi-integrals (7.17) originating from the K-RODE-Taylor schemes, each over a sub-interval $[t_n, t_{n+1}]$, are approximated by us via Riemann sums (7.18). Neither the multi-integrals nor the Riemann sums are problem-dependent, cf. equation (7.14a), but they just depend on elements of the OU process. Rearranging the Riemann sums gives

$$\int_{t_n}^{t_{n+1}} \frac{1}{d!}(t_{n+1}-u)^d \Delta O_t \mathrm{d}u \overset{(7.18)}{\approx} \delta \sum_{j=1}^{M} \frac{1}{d!}\left(t_{n+1}-u_j\right)^d \Delta O_t$$

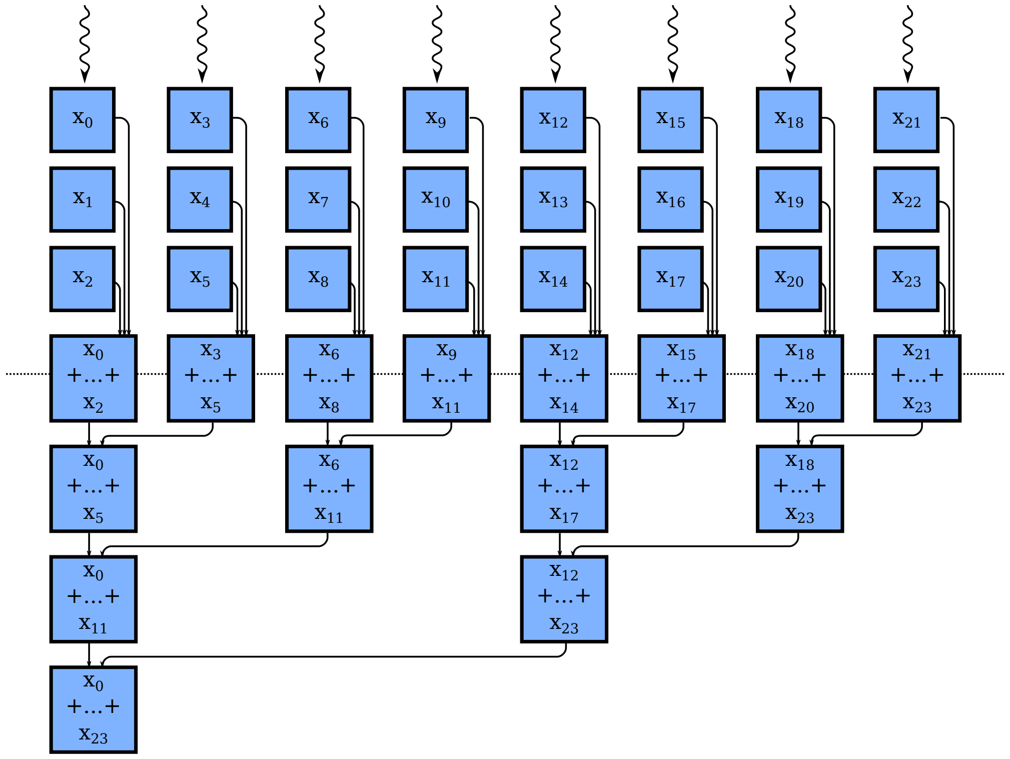


Figure 10.1.: Parallel averaging of a sequence of input numbers. First, every thread sums up a segment of equal size (here, consisting of 3 elements, e.g. $[x_9, \ldots, x_{11}]$) depicted in the upper part of the figure. Second, the partial results of the particular threads are reduced in parallel in a tree-like manner to get the overall sum depicted in the lower part. This figure is taken from our contribution [199].

$$\begin{aligned} &= \frac{\delta}{d!} \sum_{j=1}^{M} \left(t_{n+1} - (t_n + j\delta)\right)^d \left(O_{t_n+j\delta} - O_{t_n}\right) \\ &= \frac{\delta}{d!} \sum_{j=1}^{M} (h - j\delta)^d (O_{t_n+j\delta} - O_{t_n}). \end{aligned} \tag{10.3}$$

Similar to the averaged values $\bar{G}^{(1)}_{h,\delta}(t_n)$ and $\bar{G}^{(2)}_{h,\delta}(t_n)$ in the previous subsection 10.1, the Riemann sum leads to the determination of a modified average value. The same two step parallelization approach as used for the averaged schemes and depicted by figure 10.1 can be reused for the parallel computation of the Riemann sums. The corresponding kernel just has to be modified in a minor way to incorporate the factor $(h - j\delta)^d$ (h, δ, and d are constant during the calculation of a particular Riemann sum) and to sum over the difference ΔO_t instead of just the realized elements of the OU process.

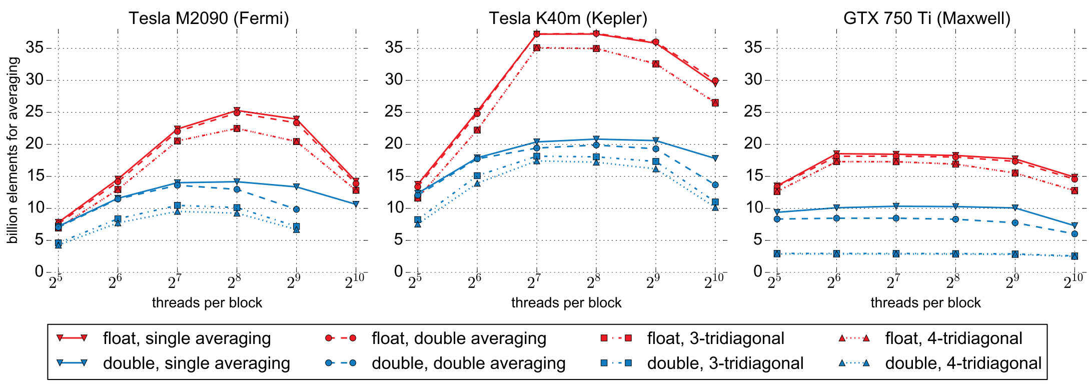


Figure 10.2.: Performance in billion elements for averaging over “threads per block” of different averagers measured on three different NVIDIA GPUs. Single precision results are colored in red, double precision results are colored in blue. The averagers for the methods averaged Euler (single averaging), averaged Heun (double averaging), 3-RODE-Taylor (3-tridiagonal) scheme and 4-RODE-Taylor (4-tridiagonal) scheme are depicted by different lines. Due to hardware limitations, some results on Fermi for computing in double precision using 2^{10} threads are missing. This plot is taken from our contribution [199].

10.3. Results

For averaging, we measure performance in billion input elements processed per second in dependence of “threads per block” grid configuration. Hence, we assign billion elements for averaging to the abscissas and threads per block, ranging from 2^5 to 2^{10}, to the ordinates of figures 10.2 and 10.3. Once again, the Teslas M2090 and K40m and the GTX 750 Ti are employed for benchmarks and profilings. Four different averagers are tested, each noted by a dedicated line and line marker: The averager originating from the single averages $\bar{G}^{(1)}_{h,\delta}(t_n)$ for the averaged Euler scheme (cf. (10.1)), the averager originating from the double averages $\bar{G}^{(2)}_{h,\delta}(t_n)$ for the averaged Heun scheme (cf. (10.2)), and the tridiagonal averagers used in the K-RODE-Taylor schemes for $K = 3$ and $K = 4$ (cf. (10.3)). Red lines represent runs using single precision operations, double precision runs are colored blue. Runtimes refer to the averaging process only but do not incorporate the generation of the input elements.

As in chapter 9, 2^{26} elements, provided by an OU process, are processed per run. Every thread block computes an average value over 2^{14} elements, thus, $\frac{2^{26}}{2^{14}} = 2^{12}$ thread blocks are started. Depending on the number of launched threads per block, every thread sums up between 2^4 and 2^9 elements during the first step. There are always much more threads than processing elements. Our implementation of the averagers also supports the joint computation of a single average value by multiple or even all thread blocks but this feature is not benchmarked in this section. The first reason for this cutback is

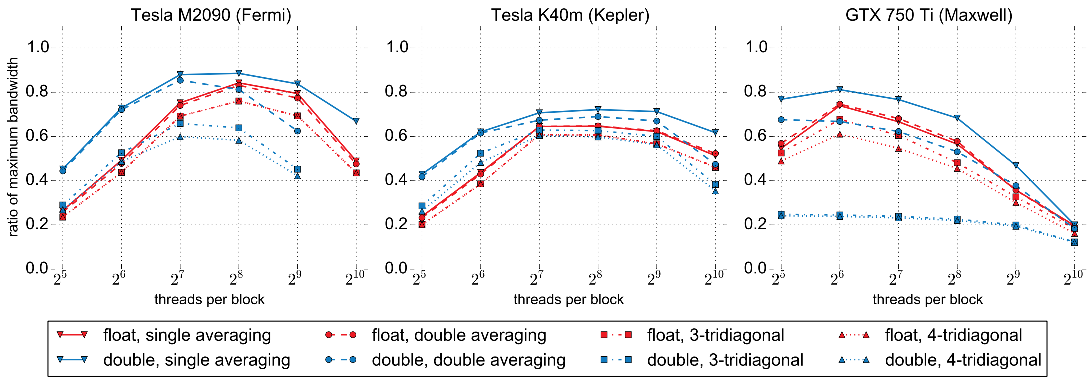


Figure 10.3.: Share of utilized memory bandwidth in peak memory bandwidth over “threads per block” of different averagers measured on three different NVIDIA GPUs. Color coding, line captions, and markers are equal to figure 10.2. This plot is taken from our contribution [199].

the setting of solving RODEs where multiple coarse timesteps, each requiring at least a single averaged value, have to be calculated instead of only one. The second reason is the observation of just small discrepancies in benchmarks and profilings between the one thread block per average value version and the multiple thread blocks per average value version.

Analyzing the benchmark results in figure 10.2 shows a familiar picture: Two limiters, one for low threads per block numbers and one for high threads per block numbers, lead to a single peak of performance on almost all GPU architectures. In this case, the non-optimal ratio of warps executing computations and warps performing memory transfers is the common limiting source. This prevents the scheduler from optimally hiding communication with computation. Performance of the averagers of the averaged Euler and Heun schemes is quite similar because they only differ in a factor $M - j$ per element. The same holds for the Riemann sums: If the number of elements per sum is the same like in our benchmark scenario, the particular tridiagonal sums just vary in the exponent d with a negligible influence on performance. Parallel setups leading to best performance on all utilized GPUs of the particular averagers are clearly listed in table 12.1.

Averaging in the way we parallelize it on the GPU is a memory-bound problem. During the first step, warps can make full advantage of coalesced memory access with a very low computational intensity. Coalesced memory access is possible due to the associativity of the add operation allowing an arbitrary optimization of the memory access pattern. In contrast, the runtime of the second step is relatively short. Let us assume a setup where 2^6 threads determine the average value of 2^{14} elements. Then, each thread sums up $\frac{2^{14}}{2^6} = 256$ elements during the first step but just 6 parallel operations are necessary to realize the reduction in the second step. Since every input element is only treated once, it does not make sense to utilize shared memory but to directly read from global memory. Figure 10.3 depicts the share of the actual memory bandwidth utilization to the peak memory bandwidth (cf. table 2.1). For single and double precision, the best carried out memory bandwidth utilizations are 84.2% and 88.5% (both running 2^8 threads per block) on the Tesla M2090, 64.7% and 72.1% (both running 2^8 threads per block) on the Tesla K40m, and 74.6% (running 2^6 threads per block) and 81.1% (running 2^7 threads per block) on the GTX 750 Ti, respectively. These values, all achieved by averager (10.1), confirm the memory-bondage of averaging and could even be improved by *vectorized memory access* [144].

The peaks and slopes of figures 10.2 (processing rate) and 10.3 (memory bandwidth utilization) do not perfectly correlate. There are two reasons explaining this behavior: First, the computational intensity for the averages of the averaged Euler and Heun schemes is lower than the intensity of the Riemann sums of the K-RODE-Taylor schemes while the amount of transferred data is the same. Second, computations in double precision require to transfer twice the amount data than single precision.

Exploiting the high memory bandwidth of GPUs makes the memory-bound task of averaging a very fast and efficient operation on these parallel computing devices. Up to 25.29, 37.22, and 18.45 billion elements are treated per second on the Teslas M2090 and K40m and the GTX 750 Ti, respectively. With normal PRNGs, realization of the OU process, and averaging, we now have everything at hands to do the transition from fine timestepping to coarse timestepping in chapter 11. This fourth building block finally enables us to determine a path-wise solution of an RODE.

11. Building block 4: Coarse timestepping for the right-hand side

The groundwork of the first three building blocks presented in chapters 8–10 enables the path-wise solution of RODEs with coarse timestepping. The averaged values, basing on the OU process, again basing on normal random numbers, are now used in the fourth building block. This final building block is the only application-specific stage of our RODE solver approach and we demonstrate it for the KT model. That means we combine terms of chapters 7 and 10 to set a path-wise numerical solution of reasonable order of convergence. To get the overall solution, the expected value of the multiple path-wise solutions is still needed.

At the beginning, we specify the averaged Euler and Heun schemes for the KT model (7.6) in section 11.1 and for the K-RODE-Taylor scheme in section 11.2 for $K = 1$, 3, and 4. In contrast to the first three building blocks, this chapter does not contain a results section. Experiments in section 12.2 show that the final building block itself actually has a negligible share in total runtime, thus, we neither optimize, parallelize, nor analyze it in any way. Using a different RODE than the KT model may increase the computational effort of the fourth building block. Then, a more detailed analysis of the coarse timestepping method can be beneficial.

11.1. Averaged schemes

The averaged Euler scheme (7.10) for the KT model (7.6) has the following form:

$$\begin{aligned}
\begin{pmatrix} z_1 \\ z_2 \end{pmatrix}_{n+1} = Z_{n+1} &\overset{(7.10)}{=} Z_n + h \cdot F_\omega(Z_n, t_n) \\
&\overset{(7.7)}{=} Z_n + h \cdot (\bar{G}^{(1)}_{h,\delta}(t_n) + \bar{g}^{(1)}_{h,\delta}(t_n) \cdot H(Z_n)) \\
&\overset{(7.8c)}{=} Z_n + h \cdot \left(\underbrace{\bar{G}^{(1)}_{h,\delta}(t_n)}_{(10.1)} + \underbrace{\bar{g}^{(1)}_{h,\delta}(t_n)}_{=1} \cdot - \begin{pmatrix} (z_2)_n \\ 2\zeta_g \omega_g (z_2)_n - \omega_g^2 (z_1)_n \end{pmatrix} \right)
\end{aligned}$$

It determines the numerical solution for the next coarse timestep $Z_{n+1} \in \mathbb{R}^2$. The functions $\bar{G}^{(1)}_{h,\delta}(t_n)$ and $\bar{g}^{(1)}_{h,\delta}(t_n)$ are single averages, H is the deterministic part of the KT model, h denotes the coarse, and δ the fine timestep size. The constants ζ_g and ω_g are introduced in section 7.2.

Analogously, incorporating double averaged values $\bar{G}^{(2)}_{h,\delta}(t_n)$ and $\bar{g}^{(2)}_{h,\delta}(t_n)$, the averaged Heun scheme (7.12) for the KT model (7.6) reads

$$\begin{aligned}
Z_{n+1} &\overset{(7.12)}{=} Z_n + \frac{h}{2}(F_\omega(Z_n, t_n) + F_\omega(Z_{n+1}, t_{n+1})) \\
&\overset{(7.7)}{=} Z_n + \frac{h}{2}\Big(\bar{G}^{(1)}_{h,\delta}(t_n) + \bar{g}^{(1)}_{h,\delta}(t_n) \cdot H(Z_n) \\
&\qquad + \bar{G}^{(1)}_{h,\delta}(t_n) + \bar{g}^{(1)}_{h,\delta}(t_n) \cdot H(Z_{n+1})\Big) \\
&= Z_n + \frac{h}{2}\left(\underbrace{\bar{G}^{(1)}_{h,\delta}(t_n)}_{(10.1)} + \underbrace{\bar{g}^{(1)}_{h,\delta}(t_n)}_{=1} \cdot \underbrace{H(Z_n)}_{(7.8c)}\right. \\
&\qquad \left. + \underbrace{\bar{G}^{(1)}_{h,\delta}(t_n)}_{(10.1)} + \underbrace{\bar{g}^{(1)}_{h,\delta}(t_n)}_{=1} \cdot \underbrace{H\left(Z_n + h\left(\underbrace{\bar{G}^{(2)}_{h,\delta}(t_n)}_{(10.2)} + \underbrace{\bar{g}^{(2)}_{h,\delta}(t_n)}_{=1} \cdot \underbrace{H(Z_n)}_{(7.8c)}\right)\right)}_{(7.8c)}\right)
\end{aligned}$$

with $Z_{n+1} \in \mathbb{R}^2$ being the numerical solution at the next coarse timestep t_{n+1}.

11.2. K-RODE-Taylor schemes

The 1-RODE-Taylor scheme using Riemann sums for quadrature has the form

$$\begin{aligned}
Z^{1,h}_{n+1} &= Z^{1,h}_n + hf + f_{(1,0)} \int_{t_n}^{t_{n+1}} \Delta O_s \mathrm{d}s \\
&\overset{(7.18)}{\approx} Z^{1,h}_n + hf + \underbrace{f_{(1,0)}\, \delta \sum_{j=1}^{M} (O_{t_n+j\delta} - O_{t_n})}_{(10.3) \text{ with } d=0}
\end{aligned}$$

with $Z_{n+1} \in \mathbb{R}^2$ representing the numerical solution at the next coarse timestep t_{n+1}.

After setting up the 3-RODE-Taylor scheme according to (7.13), the multi-integrals are simplified by (7.17) and approximated by (7.18). This leads to the numerical approximation

$$\begin{aligned}
Z^{3,h}_{n+1} \overset{(7.15)}{=} Z^{3,h}_n &+ hf + f_{(1,0)} \int_{t_n}^{t_{n+1}} \Delta O_s \mathrm{d}s \\
&+ \frac{h^2}{2} f_{(0,1)} f + f_{(0,1)} f_{(1,0)} \int_{t_n}^{t_{n+1}} \int_{t_n}^{s} \Delta O_v \mathrm{d}v \mathrm{d}s \\
&+ \frac{h^3}{6} f^2_{(0,1)} f + f^2_{(0,1)} f_{(1,0)} \int_{t_n}^{t_{n+1}} \int_{t_n}^{s} \int_{t_n}^{v} \Delta O_w \mathrm{d}w \mathrm{d}v \mathrm{d}s
\end{aligned}$$

$$
\begin{aligned}
&\overset{(7.17)}{=} Z_n^{3,h} + hf + f_{(1,0)} \int_{t_n}^{t_{n+1}} \Delta O_t \mathrm{d}t \\
&\quad + \frac{h^2}{2} f_{(0,1)} f + f_{(0,1)} f_{(1,0)} \int_{t_n}^{t_{n+1}} (t_{n+1} - (t_n + j\delta)) \Delta O_t \mathrm{d}t \\
&\quad + \frac{h^3}{6} f_{(0,1)}^2 f + f_{(0,1)}^2 f_{(1,0)} \int_{t_n}^{t_{n+1}} \frac{1}{2} (t_{n+1} - (t_n + j\delta))^2 \Delta O_t \mathrm{d}t \\
&\overset{(7.18)}{\approx} Z_n^{3,h} + hf + f_{(1,0)} \underbrace{\delta \sum_{j=1}^{M} (O_{t_n + j\delta} - O_{t_n})}_{\text{(10.3) with } d=0} \\
&\quad + f_{(0,1)} f \frac{h^2}{2} + f_{(0,1)} f_{(1,0)} \underbrace{\delta \sum_{j=1}^{M} (h - j\delta)(O_{t_n + j\delta} - O_{t_n})}_{\text{(10.3) with } d=1} \\
&\quad + \frac{h^3}{6} f_{(0,1)}^2 f + f_{(0,1)}^2 f_{(1,0)} \underbrace{\frac{\delta}{2} \sum_{j=1}^{M} (h - j\delta)^2 (O_{t_n + j\delta} - O_{t_n})}_{\text{(10.3) with } d=2}
\end{aligned}
$$

with third order of convergence.

Analogously, the 4-RODE-Taylor scheme is derived by introducing a fourth term. To keep the derivation compact, we skip the intermediate step with the multi-integrals and indicate the final approximation:

$$
\begin{aligned}
Z_{n+1}^{4,h} \overset{(7.16)(7.17)(7.18)}{\approx} & Z_n^{4,h} + hf + f_{(1,0)} \underbrace{\delta \sum_{j=1}^{M} (O_{t_n + j\delta} - O_{t_n})}_{\text{(10.3) with } d=0} \\
& + \frac{h^2}{2} f_{(0,1)} f + f_{(0,1)} f_{(1,0)} \underbrace{\delta \sum_{j=1}^{M} (h - j\delta)(O_{t_n + j\delta} - O_{t_n})}_{\text{(10.3) with } d=1} \\
& + \frac{h^3}{6} f_{(0,1)}^2 f + f_{(0,1)}^2 f_{(1,0)} \underbrace{\frac{\delta}{2} \sum_{j=1}^{M} (h - j\delta)^2 (O_{t_n + j\delta} - O_{t_n})}_{\text{(10.3) with } d=2} \\
& + \frac{h^4}{24} f_{(0,1)}^3 f + f_{(0,1)}^3 f_{(1,0)} \underbrace{\frac{\delta}{6} \sum_{j=1}^{M} (h - j\delta)^3 (O_{t_n + j\delta} - O_{t_n})}_{\text{(10.3) with } d=3} .
\end{aligned}
$$

In general, K Riemann sums are necessary for a K-RODE-Taylor scheme.

The right-hand side of the KT model is

$$f \overset{(7.6)}{=} \begin{pmatrix} -(z_2 + O_t) \\ -2\zeta_g\omega_g(z_2 + O_t) + \omega_g^2 z_1 + O_t \end{pmatrix}.$$

The partial derivatives $f_{(1,0)}$ and $f_{(0,1)}$ of f are parts of the Jacobian of $f(O_t, z_1, z_2)$

$$\nabla f(\omega, Z) = \nabla f(O_t, z_1, z_2) = \left(\underbrace{\begin{array}{c} \frac{\partial f^1}{\partial O_t}(x) \\ \frac{\partial f^2}{\partial O_t}(x) \end{array}}_{=:\, f_{(1,0)}} \middle| \underbrace{\begin{array}{cc} \frac{\partial f^1}{\partial z_1}(x) & \frac{\partial f^1}{\partial z_1}(x) \\ \frac{\partial f^2}{\partial z_1}(x) & \frac{\partial f^2}{\partial z_1}(x) \end{array}}_{=:\, f_{(0,1)}} \right) \Bigg|_{x=(O_t, z_1, z_2)}$$

with $f, Z \in \mathbb{R}^2$ and $\omega \in \mathbb{R}$. Hence,

$$f_{(1,0)} = \begin{pmatrix} -1 \\ -2\zeta_g\omega_g + 1 \end{pmatrix} \in \mathbb{R}^2, \quad f_{(0,1)} = \begin{pmatrix} 0 & -1 \\ \omega_g^2 & -2\zeta_g\omega_g \end{pmatrix} \in \mathbb{R}^{2\times 2}.$$

The solution of the KT model introduced so far is $Z = (z_1, z_2)^T$. The actual physical values of the earthquake namely ground position x_g, velocity $\dot{x}_g$, and acceleration $\ddot{x}_g$ are finally determined via

$$x_g = z_1,$$

$$\dot{x}_g = -(z_2 + O_t),$$

and

$$\ddot{x}_g = -2\zeta_g\omega_g\dot{x}_g - \omega_g^2 x_g$$

with $x_g, \dot{x}_g, \ddot{x}_g \in \mathbb{R}$.

This finishes the final building block and, thus, the central elements of our RODE solver. The final building block is the only application-specific step in the presented solver pipeline. Hence, depending on the actual application, a parallelization of the coarse timestepping can make sense being not necessary for the KT model as discussed, amongst other things, in the next chapter 12.

12. Results of the full random ordinary differential equations solver

With an efficient GPU implementation of the four building blocks (generation of normal random numbers, realization of the OU process, averaging of the elements of the OU process, and the solution of the right-hand side of the application RODE using coarse timestepping) at hand, it is possible to analyze and benchmark the interaction of them. The first three building block chapters 8–10 already provide profiling and benchmark results for the particular building blocks in their final sections (cf. sections 8.4, 9.3, and 10.3). In this chapter, we apply the best performing configurations (number of Ziggurat strips, parallel setup, elements per thread, etc.) to the building blocks and evaluate the performance of our entire RODE solver approach. This includes the study of a single path-wise solution as well as the investigation of the expected value of multiple path-wise solutions.

For building blocks two and three (cf. chapters 9 and 10, respectively), only one concrete parallel concept to realize the corresponding task (realization of the OU process and averaging) was given, thus, these particular approaches are executed here. Out of the three normal random number generators presented in chapter 8, the Ziggurat method is used throughout this chapter. To keep it compact, only one normal PRNG is utilized; it could be replaced by rational polynomials or the Wallace method without any limitations. The RODE application is still the KT model. Regarding the RODE solver, we conduct measurements with the averaged Euler scheme as representative for the averaged schemes and the 3-RODE-Taylor scheme as representative for the K-RODE-Taylor schemes and omit results for the averaged Heun scheme and K-RODE-Taylor schemes for $K \neq 3$. For the single-GPU evaluations, the Teslas M2090 and K40m and the GTX 750 Ti are utilized in sections 12.1 and 12.2. The GPU clusters JuDGE, Hydra, and TSUBAME2.5 (cf. table 2.5) are employed for the multi-GPU evaluations in section 12.3.

The remainder of this chapter is structured as follows: First, the best performing configurations for the first three building blocks being used for profiling and benchmarking in the subsequent sections are summed up in section 12.1. In addition, profiling values such as FLOPS rate relative to maximum peak performance and occupancy resulting from these configurations are provided. Second, the share of every GPU kernel in total runtime for a path-wise solution of the RODE is listed in section 12.2 which, hence, profiles the first two levels of parallelism originating from the building blocks. In section 12.3, the scalability of the Monte Carlo approach, i.e. the third level of parallelism, is evaluated on three GPU clusters. Finally, an empirical analysis of the statistical properties of the presented RODE solver is given in section 12.4 by comparing the solutions of different solvers using various amounts of path-wise solutions.

12.1. Configurations of choice for the building blocks

building block	configuration parameter	Tesla M2090		Tesla K40m		GTX 750 Ti	
		`float`	`double`	`float`	`double`	`float`	`double`
1. PRNG	t./b.	2^6	-	2^6	-	2^5	-
	#strips	2^{10}		2^{10}		2^8	
	implementation	local mem.		shared mem.		shared mem.	
2. OU process	t./b.	2^9			2^{10}	2^9	
	elements/thread	2^3			2^2	2^3	
3. averaging	t./b. (single)	2^8				2^6	2^7
	t./b. (double)	2^8	2^7	2^8		2^7	
	t./b. (3-trid.)		2^7			2^6	
	t./b. (4-trid.)					2^6	2^5

Table 12.1.: Optimal configuration parameters leading to best performance of the first three building blocks. Configuration parameters vary depending on the utilized GPU and floating-point precision (`float` or `double`). The abbreviation t./b. stands for "threads per block" and specifies the parallel setup. Single and double refers to the averagers for the averaged Euler and Heun scheme, respectively. 3-tridiag. and 4-tridiag. denote the Riemann sums for the 3- and 4-RODE-Taylor scheme.

To achieve the best overall performance for our entire RODE solver, the best performing configurations of the particular building blocks are applied. The performance is measured in GPRNs/s, GROUPs/s, and processed elements per second for PRNG, OU process, and averaging, respectively. These configurations for the first three building blocks are already discussed in the corresponding result sections and listed in table 12.1. The configuration parameters for the PRNG refer to the Ziggurat method.

Using these optimal configuration parameters leads to the profiling values in table 12.2 determined by the command line profiler `nvprof` [182]. FLOPS rate relative to maximum peak performance and theoretical and measured occupancy are determined for all kernels of the first three building blocks. Table 2.1 lists the peak FLOPS rates being the reference value for the relative FLOPS rates. Having a closer look on these rates testifies that none of the building blocks is compute-bound. Only the GTX 750 Ti performing double precision operations shows significant relative FLOPS rates but this behavior is related to the very low double precision performance of this GPU. Instead, the kernels of the PRNG and OU process building block are latency-bound, the averaging is memory-bound. The measured occupancies correlate well with the expected theoretical occupancies. Since benchmarks for the normal PRNG are only carried out in single precision, no corresponding double precision values are given in tables 12.1 and 12.2.

building block	k.	perfor. value	Tesla M2090		Tesla K40m		GTX 750 Ti	
			`float`	`double`	`float`	`double`	`float`	`double`
PRNG	1a	% peak	0.01%	-	0.01%	-	0.01%	-
		occ. theo.	0.33		0.28		0.5	
		occ. meas.	0.32		0.27		0.46	
	1b	% peak	0.02%	-	0.02%	-	0.58%	-
		occ. theo.	0.33		0.16		0.3	
		occ. meas.	0.33		0.15		0.29	
OU process	2a	% peak	0.14%	0.21%	0.03%	0.06%	0.98%	11.89%
		occ. theo.	0.66	0.33	0.5		0.75	0.25
		occ. meas.	0.66	0.33	0.49		0.73	0.25
	2b	% peak	0.28%	1.60%	0.03%	0.59%	0.79%	47.69%
		occ. theo.	1					
		occ. meas.	0.91	0.97	0.89	0.91	0.87	0.89
	2c	% peak	0.44%	0.38%	0.06%	0.11%	1.05%	27.84%
		occ. theo.	1					
		occ. meas.	0.89	0.83	0.87	0.82	0.86	0.91
aver-aging	3a	% peak	0.14%	0.22%	0.06%	0.10%	1.15%	21.34%
		occ. theo.	1	0.83	1		0.72	0.81
		occ. meas.	0.99	0.8	1		0.71	0.77
	3b	% peak	1.28%	1.78%	0.31%	0.69%	5.88%	48.86%
		occ. theo.	1	0.5	0.86	0.56	0.72	0.4
		occ. meas.	0.99	0.5	0.87	0.56	0.71	0.4

Table 12.2.: Profiling values for the first three building blocks using the optimal parameters from Table 12.1: FLOPS rate relative to maximum peak performance (% peak) and theoretical (occ. theo.) and actually measured (occ. meas.) occupancy. The values are broken down kernel-wisely (k.) for three different GPUs. Row 1a lists values for the initialization of the PRNG (`initStatesNormalKernel()`) while the actual kernel for the generation (`getRandomNumbersNormalKernel()`) is listed in row 1b. Explanations of the three kernels `scanExclusiveOUKernel()` (denoted by 2a), `scanOUFixKernel()` (denoted by 2b), and `realizeOUProcessKernel()` (denoted by 2c) for the OU process can be found in section 9.3. The averager for the averaged Euler scheme (`singleAverageKernel()`) and the Riemann sums for the 3-RODE-Taylor scheme `tridiagIntegralApproximationKernel()` are given in rows 3a and 3b. This table is taken from our contribution [199].

12.2. Profiling of single path-wise solutions

In this section, the share each kernel of our RODE solver contributes in total runtime is examined depending on the problem size. All four building blocks are considered for a single path-wise solution of the RODE, thus, one OU process is realized to profile the first two levels of parallelism. Results for an averaged Euler scheme scenario are depicted in figure 12.1 and for an 3-RODE-Taylor scheme scenario in figure 12.2 conducted on the Tesla M2090, Tesla K40m, and the GTX 750 Ti. Runtimes are normalized to 1, thus, the runtime of all four building blocks adds up to 1 and the runtimes of the particular kernels are divided by this total runtime. Hence, the shares of the particular kernels are easier to identify independent of the actual problem size. The problem size is denoted by "seconds of simulated time" $\times$ "coarse timesteps per second" (i.e. $\frac{1}{h}$) $\times$ "fine timesteps per coarse timestep" (i.e. $\frac{h}{\delta}$) and ranges from 2^3 seconds up to 2^{10} seconds (which is a relatively big problem size) of simulation time. The execution of the kernels is configured according to table 12.1.

In contrast to the building blocks two, three, and four, the normal random number

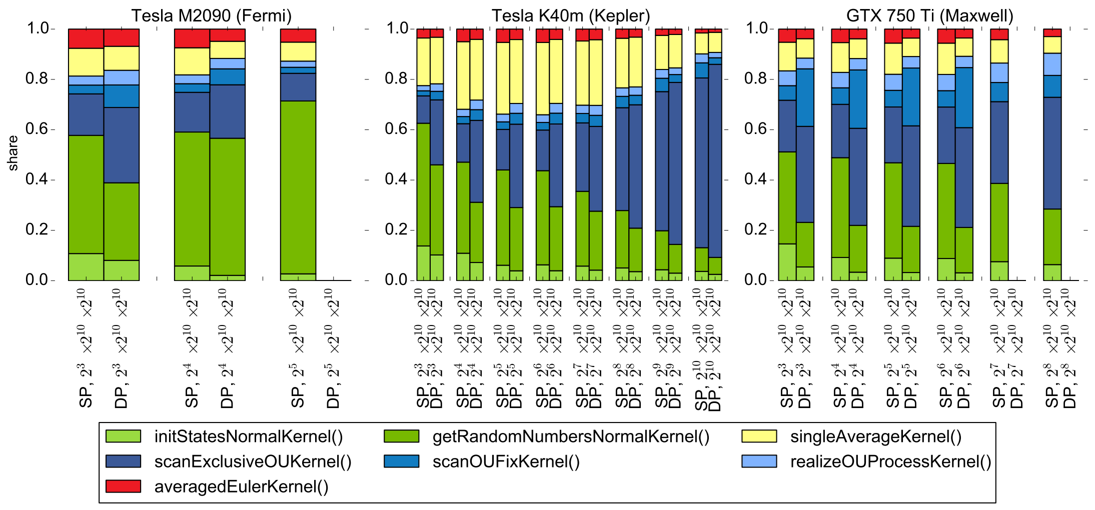


Figure 12.1.: Shares in total runtime of all necessary kernels to determine a path-wise solution of the KT model applying the averaged Euler scheme on three different GPUs. Results depend on problem size denoted by "seconds of simulated time" $\times$ "coarse timesteps per second" $\times$ "fine timesteps per coarse timestep" and are conducted in single (SP) and double precision (DP). All values are normalized. The generation of normal random numbers also includes times for initialization of the PRNGs and is shown in green. The realization of the OU process executing three kernels is depicted in blue. The share of averaging is colored in orange and the red bar corresponds to the coarse timestepping of the averaged Euler scheme. This figure is taken from our contribution [199].

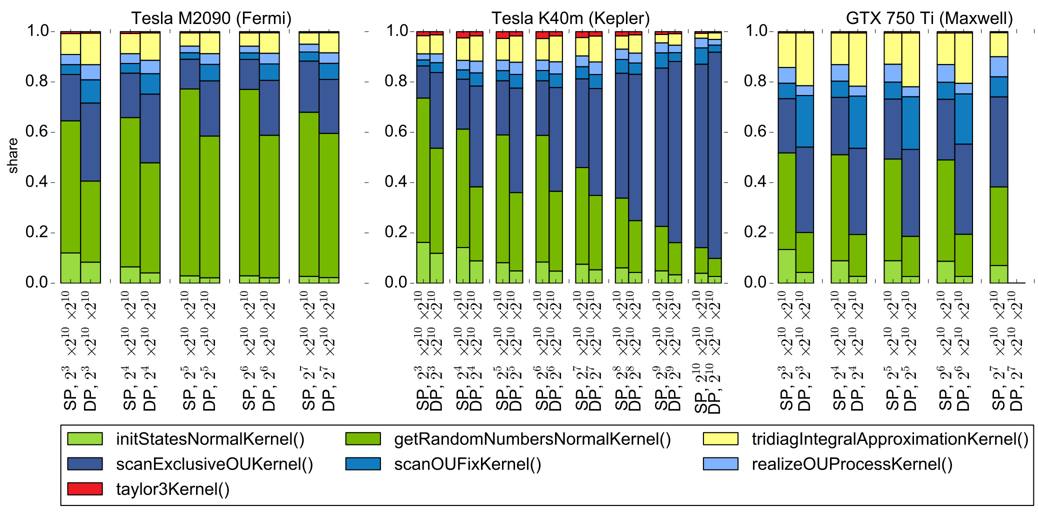


Figure 12.2.: Shares in total runtime of all necessary kernels to determine a path-wise solution of the KT model applying the 3-RODE-Taylor scheme on three different GPUs. Axes assignments and color coding are equal to figure 12.1. This figure is taken from our contribution [199].

generation is always carried out in single precision as in section 8.4, also for the double precision setups. That is the reason why its share in the single precision benchmark is always higher than in the corresponding double precision benchmarks because on the GPU, operations in double precision are always more expensive than in single precision. For both the averaged Euler and the 3-RODE-Taylor scheme scenario the first three building blocks have the largest shares in total runtime. They are 77.2% for normal random number generation (3-RODE-Taylor scheme on the Tesla M2090 in single precision simulating 2^4 seconds), 87.0% for the realization of the OU process (3-RODE-Taylor scheme on the Tesla K40m in double precision simulating 2^{10} seconds), and 28.7% for averaging (averaged Euler scheme on the Tesla K40m in double precision simulating 2^4 seconds). With a contribution between 0.3% (3-RODE-Taylor scheme on the GTX 750 Ti in single precision simulating 2^8 seconds) and 7.6% (averaged Euler on Tesla M2090 in single precision simulating 2^3 seconds) in total runtime, the share of the coarse timestepping is negligible[1]. There are two reasons for this ratio: First, the initial three building blocks operate on fine timesteps δ, thus, their computational load is much higher than the final building block which operates only on coarse timesteps h. That especially holds for the 3-RODE-Taylor scheme because there $\delta = h^4$ instead of $\delta = h^2$. Second, the final solution of the KT model (7.3) is very cheap in comparison to other RODEs because $\ddot{u}_g$ is only one-dimensional. Therefore, the parallelization and optimization for GPUs of the KT model's final building block is unnecessary.

[1]Actually, 7.6% is a considerable share but it refers to a serial implementation while all other building blocks are parallelized on and optimized for GPUs.

On all utilized GPUs, independent of single or double precision operations, the share of the OU process becomes larger the bigger the problem size gets. This phenomenon can be best observed on the Tesla K40m. Since our implementation of the OU process bases on parallel prefix sums, its scalability is limited by the parallel scan's tree-like execution pattern (cf. figures 9.2 and 9.3). Such an algorithmical limitation does not apply to normal random number generation and averaging (its second step to reduce the partial sums is negligible), so they scale much better for larger problem sizes.

12.3. Scalability of the multi-path solution

In this section, the third level of parallelism of the RODE solver, i.e. the Monte Carlo approach to combine multiple path-wise solution to a global solution by identifying the expected value, is examined. This level of parallelism is mapped to multiple GPUs and its scaling behavior is benchmarked. Every GPU determines an own path-wise solution with a dedicated realization of the OU process which can be done embarrassingly parallel because no communication is required for this operation. Only at the end, one global communication step is necessary, more specifically, a global reduce operation realized via the `MPI_Reduce()` operation. Since the problem size is the same for all GPUs during the entire runtime, no load balancing is required, neither at the beginning nor during

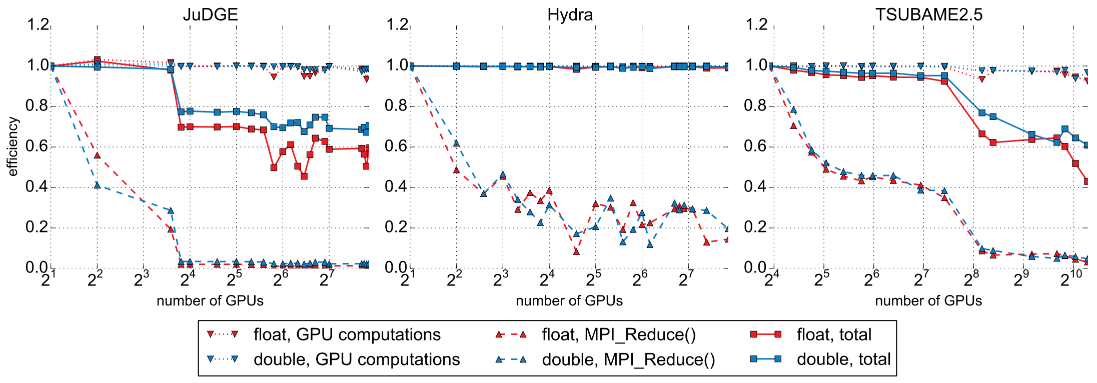


Figure 12.3.: Parallel efficiency of the entire multi-path RODE solver using the averaged Euler scheme depending on the number of utilized GPUs/MPI ranks on three GPU clusters. Single precision results are colored in red, double precision results are colored in blue. Dotted lines depict the parallel efficiency of all four building blocks carried out on the GPU. Dashed lines indicate the parallel efficiency of the final global reduction step. Solid lines give the combined parallel efficiency for the the entire RODE solver. The baseline value for parallel efficiency is 2 GPUs for JuDGE and Hydra and 15 GPUs for TSUBAME2.5. The largest run uses 232 GPUs on JuDGE and Hydra and 1248 GPUs on TSUBAME2.5. This figure is taken from our contribution [199].

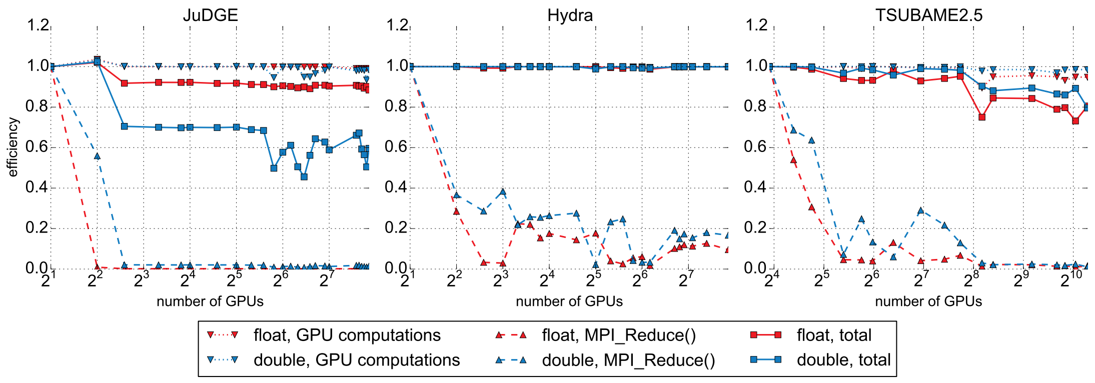


Figure 12.4.: Parallel efficiency of the entire multi-path RODE solver using the 3-RODE-Taylor scheme depending on the number of utilized GPUs/MPI ranks on three GPU clusters. Axes assignments, color coding, line captions, and markers are equal to figure 12.3. This figure is taken from our contribution [199].

the execution. Such an approach is comparable to weak scaling: The problem size (here, the problem size corresponds to the number of path-wise solutions) is altered in the same way (increased or decreased) as the degree of parallelism (here, the degree of parallelism corresponds to the number of utilized GPUs). However, in contrast to classical weak scaling, the degree of parallelism influences the statistical properties of the Monte Carlo solution instead of spatial or temporal size or resolution. In such a context, a strong scaling scenario is also conceivable: A constant number of path-wise solutions is determined by a varying number of GPUs. The benchmarks in this section are only dealing with the weak scaling-like scenario. Strong scaling-like scenarios are not considered because according to our experience, the scaling behavior is quite similar to the weak scaling-like scenarios and, thus, does not provide any new insights.

Benchmarks to analyze the scaling behavior are carried out on JuDGE, Hydra, and TSUBAME2.5. Technical data for these three GPU clusters is listed in table 2.5. This table also gives information regarding the utilized CUDA version for the building blocks and MPI implementation for the Monte Carlo level of parallelism. JuDGE is equipped with Teslas M2050 which are quite similar to the Tesla M2090. Hence, the best performing parameter configurations given in table 12.1 in column Tesla M2090 also hold for the Tesla M2050 and are used throughout this section. Hydra and TSUBAME2.5 are equipped with Teslas K20x which are quite comparable to the Tesla K40m. Accordingly, the values of table 12.1's Tesla K40m column are used throughout this section because they are also the best performing parameter configurations for the Tesla K20x due to their similar architecture. One MPI rank is started per GPU, so the number of MPI ranks per node is equal to the number of GPUs per node. All GPUs per node are utilized. Table 12.3 lists the problem sizes of the utilized test scenarios denoted by “seconds of simulated time” × “coarse timesteps per second” × “fine timesteps per coarse timestep".

	JuDGE	Hydra	TSUBAME 2.5
averaged Euler scheme	$2^4 \times 2^{10} \times 2^{10}$	$2^9 \times 2^{10} \times 2^{10}$	
3-RODE-Taylor scheme	$2^7 \times 2^5 \times 2^{15}$	$2^9 \times 2^5 \times 2^{15}$	

Table 12.3.: Problem sizes for multi-GPU benchmarks denoted by "seconds of simulated time" × "coarse timesteps per second" × "fine timesteps per coarse timestep".

Figures 12.3 and 12.4 depict the parallel efficiencies (assigned to the ordinates) achieved on the three GPU clusters for the two different RODE solver scenarios (see rows in table 12.3). The baseline for parallel efficiency is 1 node/2 GPUs for JuDGE and Hydra and 5 nodes/15 GPUs for TSUBAME2.5 and the largest runs on JuDGE and Hydra utilize 232 GPUs/MPI ranks and 1248 GPUs/MPI ranks on TSUBAME2.5 (assigned to the abscissas). Three times are measured: (1) The accumulated runtime of the four building blocks, (2) the runtime for the final global reduction operation calling `MPI_Reduce()`, and (3) the total runtime of the entire multi-path solver summing up the first two runtimes indicated by dotted, dashed, and solid lines, respectively. Experiments are run in single (red lines) and double precision (blue lines) where the normal random number generation of the double precision scenario is carried out in single precision. While the execution times of the four building blocks do not differ much on the particular GPUs, there is a significant variation in runtime of the `MPI_Reduce()` operation due to its usually tree-like implementation. Hence, benchmarks are run 100 times, afterwards, runtimes are averaged and, finally, the worst average of all nodes is taken for figures 12.3 and 12.4.

Comparing the parallel efficiencies of the four building blocks (dotted lines) with parallel efficiencies of the global reduce operation (dashed lines) reveals an almost perfect

system	#GPUs	averaged Euler scheme		3-RODE-Taylor scheme	
		`float`	`double`	`float`	`double`
JuDGE	232	59.6%	70.6%	88.5%	59.6%
Hydra		98.7%	99.3%	81.0%	99.2%
TSUBAME2.5	213	94.4%	96.4%	95.2%	98.2%
	1248	42.9%	60.9%	80.5%	79.5%

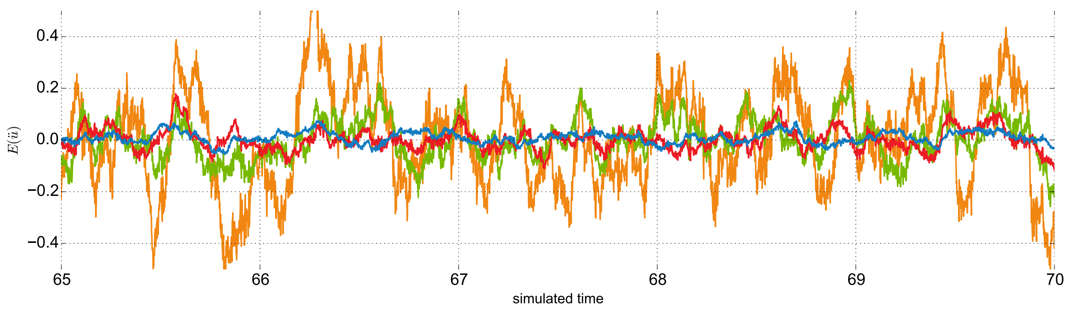
Table 12.4.: Selected values of parallel efficiency for total runtime on the three benchmarked GPU clusters (rows) applying the averaged Euler and 3-RODE-Taylor scheme (columns). Values given for JuDGE and Hydra are the largest runs on these two GPU clusters each using 232 GPUs. For TSUBAME2.5, results of a run of similar size (213 GPUs) are listed. In addition, the overall largest run is done on TSUBAME2.5 using 1248 GPUs.

scaling of the path-wise solution of the RODE but a very poor scalability to resolve the expected value to optain the multi-path solution. Depending on the chosen numerical RODE solver (cf. table 12.3), the reduce operation has a varying share in the total execution time: In the averaged Euler scheme scenarios, 2^{10} fine timesteps are realized per coarse timestep while in the 3-RODE-Taylor scheme, 2^{15} fine timesteps are computed. So the computational effort of the latter solver for the four building blocks is much higher than for the averaged Euler scheme and, thus, the global reduction has less influence in total runtime. Furthermore, the scalability of MPI operations depends on the actual MPI implementation (see row “MPI” in table 2.5) and the network topology of the cluster being a fat tree for Hydra and TSUBAME2.5 showing a significantly better MPI performance than JuDGE. Table 12.4 lists the parallel efficiencies of total execution time of the largest runs performed on the corresponding GPU clusters. To compare the results of JuDGE and Hydra with the ones of TSUBAME2.5, a run of similar size (213 GPUs) on this cluster is also given. The most remarkable values are the 99.3% and 99.2% achieved parallel efficiency on Hydra for the two numerical RODE solvers using double precision. Very good values 80.5% and 79.5% using single and double precision, respectively, for the 3-RODE-Taylor scheme are obtained in the overall largest runs using 1248 GPUs on TSUBAME2.5. These values demonstrate the outstanding scalability of our RODE solver approach on GPU clusters.

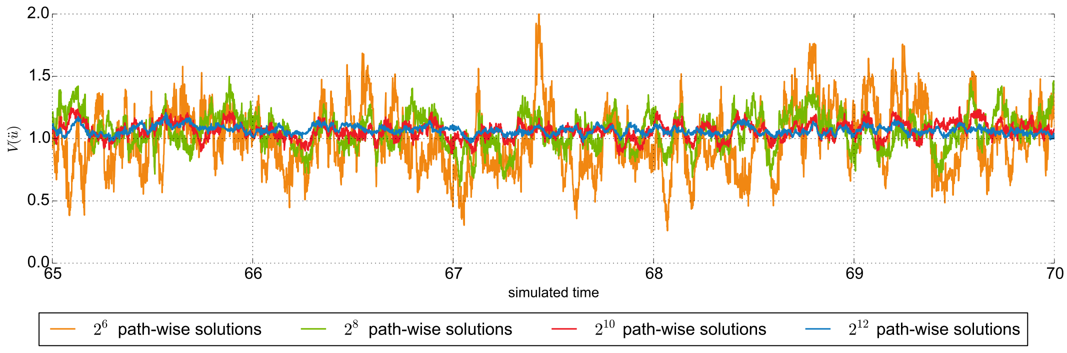
12.4. Statistical evaluation of the multi-path solution

The main goal of this work is the development of an efficient solver for RODEs on GPU clusters exploiting their multiple levels of parallelism. The performance of our RODE solver is discussed in the previous sections 12.2 and 12.3. With such a high-performance implementation at hand it is possible to experimentally analyze the statistical properties of an RODE solution. This section is limited to an experimental study giving qualitative instead of quantitative statements and omits a mathematical discussion. The expected value E and the variance V of the path-wise solutions are the most interesting statistical values because E corresponds to the final RODE solution and V can be used as an indicator how many path-wise solutions are required to obtain a reasonable solution.

Once again, we use the KT model for demonstration. For the statistical experiments, the same scenarios as listed in table 12.3 on the Hydra cluster are investigated using double precision. $E(\ddot{u})$ and $V(\ddot{u})$ for the averaged Euler scheme using different numbers of path-wise solutions are plotted in figure 12.5. Figure 12.6 depicts the statistical values for the 3-RODE-Taylor scheme scenario. $E(\ddot{u})$ at a certain point of simulation time is determined by calculating the averaged value of all considered path-wise solutions at the same point of simulation time. $V(\ddot{u})$ is computed accordingly. Both figures show the arbitrary interval $[65, 70]$ of simulated time. The number of path-wise solutions is varied between $2^6 = 64$ and $2^{12} = 4096$ and indicated by different lines. Since h differs by a factor 2^5 between the averaged Euler and the 3-RODE-Taylor scheme, there is a clearly visible discrepancy between the sampling rates in figures 12.5 and 12.6. The same sequences of normal random numbers are used for both solvers leading to the same



(a) Expected value



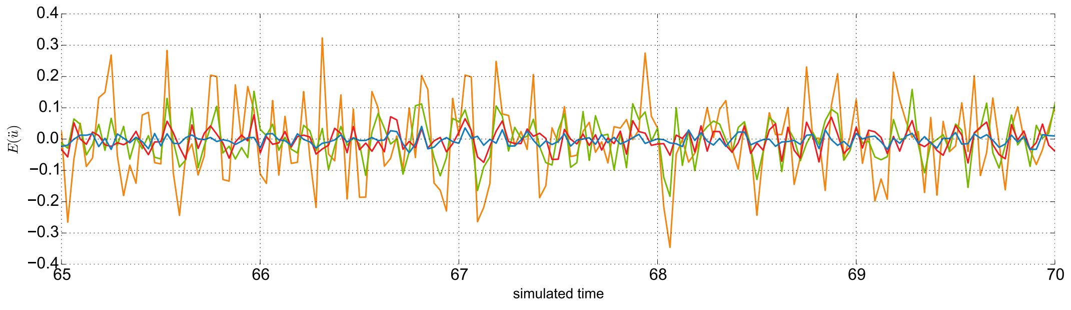
(b) Variance

Figure 12.5.: Expected value $E(\ddot{u})$ (subfigure 12.5(a)) and variance $V(\ddot{u})$ (subfigure 12.5(b)) of path-wise solutions for the KT model obtained by the averaged Euler scheme in the arbitrary time interval $[65, 70]$. Initial value Z_0 is 0, $h = \frac{1}{2^{10}}$, and $\delta = \frac{1}{2^{20}}$. Different lines represent different numbers of used path-wise solutions to determine a global solution ranging from 2^6 to 2^{12}.

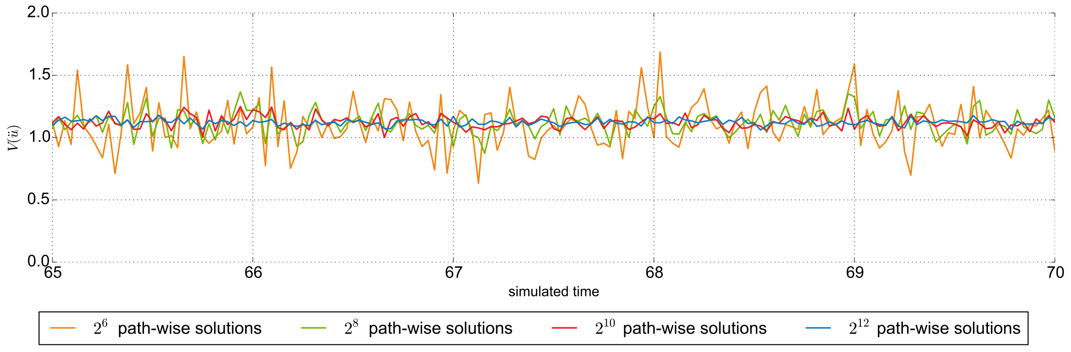
realizations of the OU process because in both cases $\delta = \frac{1}{2^{20}}$. Solutions involving more path-wise solutions reuse the random number sequences of solutions incorporating less path-wise solutions. $Z_0 = 0$ is used as initial start value.

Subfigure 12.5(a) shows a decreasing amplitude for $E(\ddot{u})$ the larger the number of involved path-wise solutions gets because they start to cancel out each other. While $V(\ddot{u})$ varies much for lower number of path-wise solutions when comparing different points in simulation time, the variance becomes “smoother” for larger path-wise solution numbers in subfigure 12.5(b). There is only a minor difference in smoothness between the 2^{10} (red) and 2^{12} (blue) path-wise solutions setup but still a significant difference between the 2^8 (green) and 2^{10} setups. Hence, combining 2^{10} path-wise solutions from the averaged Euler scheme for the KT model using the above given h and δ is a meaningful choice because it is a good compromise between low cancellation and smooth variance.

A similar behavior is observed in figure 12.6 plotting E and V for the 3-RODE-Taylor



(a) Expected value



(b) Variance

Figure 12.6.: Expected value $E(\ddot{u})$ (subfigure 12.6(a)) and variance $V(\ddot{u})$ (subfigure 12.6(b)) of path-wise solutions for the KT model obtained by the 3-RODE-Taylor scheme in the arbitrary time interval $[65, 70]$. Initial value Z_0 is 0, $h = \frac{1}{2^5}$, and $\delta = \frac{1}{2^{20}}$. Different lines represent different numbers of used path-wise solutions to determine a global solution ranging from 2^6 to 2^{12}.

scheme scenario. The more path-wise solutions are combined, the smaller the magnitude of the expected value (see subfigure 12.6(a)) and the smoother the variance (see subfigure 12.6(b)) over simulated time get. In contrast to the averaged Euler scheme scenario, already the Monte Carlo solution combining 2^8 path-wise solutions shows a similar variance behavior like solutions comprising more path-wise solutions. So using less path-wise solutions in comparison to the averaged Euler scheme scenario is acceptable and preferable because fewer path-wise solutions lead to less computational effort.

Figure 12.7 illustrates a direct comparison of the expected values stemming from the different solver scenarios each using $2^{10} = 1024$ path-wise solutions. The green and blue lines correspond to the values in figures 12.5 and 12.6 while the red line represents an averaged Euler scenario with $h = \frac{1}{2^5}$ and $\delta = \frac{1}{2^{20}}$ satisfying the order of convergence criterion in subsection 7.3.1. Comparing the red and blue lines reveals almost the same shape of E. Both use the same $h = \frac{1}{2^5}$ and the same sequences of random numbers and,

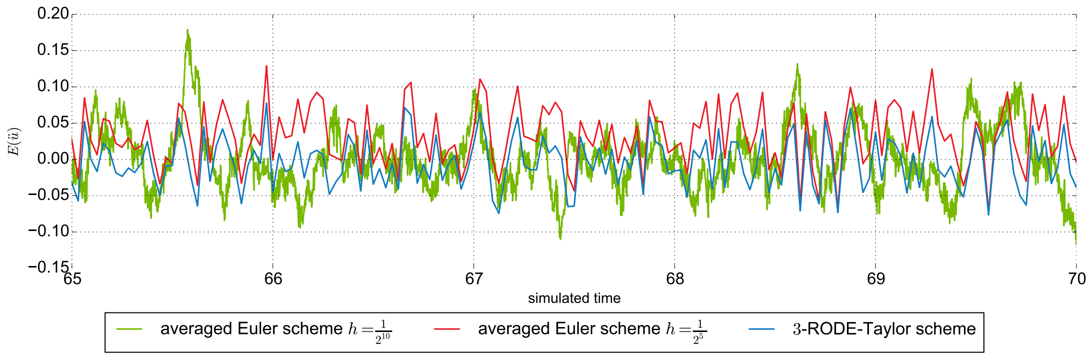


Figure 12.7.: Comparison of expected values $E(\ddot{u})$ using 2^{10} path-wise solutions for the KT model over the arbitrary time interval $[65, 70]$ obtained by three different solver scenarios: The green and red expected values stem from solutions obtained by the averaged Euler method with $h = \frac{1}{2^{10}}$ and $h = \frac{1}{2^5}$, respectively. The blue lines corresponds to solutions from the 3-RODE-Taylor scheme $h = \frac{1}{2^5}$. $\delta = \frac{1}{2^{20}}$ for all three RODE solutions.

thus, the only difference is the usage of single averages (10.1) for the averaged Euler scheme instead of the Riemann sums (10.3) for the 3-RODE-Taylor scheme. Furthermore, the green line also has a similar shape in many sub-intervals of simulated time in comparison to the solutions using $h = \frac{1}{2^5}$ although the temporal resolution is much finer. More importantly, the minimum and maximum values of all solver scenarios are quite comparable which is a relevant qualitative statement: The KT model models a white-noise driven excitement, so depending on the realization of the white-noise, solutions are not deterministic. Instead, information such as maximum ground acceleration is much more relevant and can be obtained from the KT model.

Besides the statistical properties, another insight of chapter 12 is that the execution times of normal random number generation, the realization of the OU process, and its averaging have by far the most dominant shares in total runtime while the share of the fourth building block is negligible, especially when applying the 3-RODE-Taylor scheme as shown in section 12.2. For these shares, the first three building blocks are already parallelized on and optimized for the GPU but the actual coarse timestepping solver for the KT model is executed in a sequential way. In section 12.3 we exemplify the excellent scalability of our RODE solver obtaining a parallel efficiency of up to 99.3% on Hydra utilizing 232 GPUs and 80.5% on TSUBAME2.5 employing 1248 GPUs. These values can be achieved because the share of the four building blocks in total execution time is vastly larger than the final reduction step to determine the overall solution. This terminating step is the only one requiring global communication among multiple GPUs, all other steps are restricted to dedicated GPU.

Concluding part III

We present a common approach to numerically solve arbitrary RODEs. It is the first attempt to deal with RODEs at a large scale, not just on GPU clusters but on high-performance systems in general. The algorithms and parallelization techniques presented so far are not limited to GPU clusters but are also applicable to other parallel computing devices. RODEs are a special class of ODEs incorporating a stochastic process. Our approach consists of four subsequent building blocks to determine a path-wise solution of the RODE and a Monte Carlo step combining multiple path-wise solutions to obtain the actual solution. While the four building blocks are mapped to the GPUs of the cluster without any communication, the final step requires communication among these GPUs. Only the GPUs carry out actual computations, CPUs just perform MPI communication. All functionality of the building blocks is realized by newly developed CUDA kernels, no external libraries are utilized.

The introduction to RODEs (containing their relation to SODEs, their numerical solution, and the presentation of the KT model as an example for an RODE) is followed by the first building block dealing with normal random number generation. Three established normal PRNGs (the Ziggurat method, rational polynomials, and the Wallace method) with properties making them high-performance candidates on GPUs are explained. Benchmarks state that our GPU-optimized implementation of these normal PRNGs can deliver up to 4.46 GPRN/s (Wallace method on the Tesla M2090) outperforming state-of-the-art normal PRNGs for the CPU (MKL) by up to 2.61× and state-of-the-art GPU libraries (cuRAND) by up to 4.53×. The normal random numbers are necessary for the realization of the OU process being the fundamental stochastic process of RODEs. Our implementation for GPUs leads to the first successful parallelization of this stochastic process. The central idea for parallelization is the mapping of the OU process to prefix sums and afterwards exploiting existing parallelization strategies for prefix sums. Due to the tree-like structure of these strategies, perfect scaling is not possible but realization rates of up to 3.20 and 1.11 GROUPs/s in single and double precision, respectively, are achievable on a Tesla K40m. The numerical solvers for RODEs, all doing some kind of sub-sampling, require different kinds of averages of continuous sequences of the OU process. GPUs are suited very well for this task averaging up to 37.22 billion single precision 20.82 billion double precision elements per second on a Tesla K40m. Up to the third building block, none of the operations is specific to a particular RODE but are required to solve RODEs in general. Furthermore, normal random number generation, the realization of the OU process, and averaging are not restricted to RODEs but are applied in many other fields such as performance modeling and image processing. The only RODE-specific building block is the path-wise solution of RODEs with coarse timestepping handling the averaged values from the previous building block step. Its share in total runtime of the four building blocks is negligible, thus, in contrast to the first three building blocks, neither an optimization nor a parallelization for GPUs is necessary if the KT model is applied.

Except the final building block, the building blocks are parallelized using two levels of parallelism being mapped to the two levels of hardware parallelism of GPUs. The determination of the expected value of a bunch of path-wise solutions, being the actual solution of the RODE, forms a third level of parallelism being mapped to multiple GPUs. Depending on the parameter configuration of the building blocks, the largest share in runtime of normal random number generation is 77.2%, 87.0% for the realization of the OU process, 28.7% for averaging, and 7.6% for coarse timestepping. Since the particular path-wise solutions all have equal load during the entire runtime and no communication is needed to determine them, our RODE solver scales extremely well with a parallel efficiency of up to 99.3% utilizing 232 GPUs on Hydra and 80.5% employing 1248 GPUs on TSUBAME2.5. The RODE solver developed throughout this part of the thesis makes it possible to tackle RODEs of reasonable size: For example, a high-order numerical solver such as the 3-RODE-Taylor scheme with $h = \frac{1}{2^5}$ and $\delta = \frac{1}{2^{20}}$ can be utilized to solve the KT model for dozens of minutes of simulated time incorporating thousands of path-wise solutions. The entire solution process just takes some seconds of wall-clock time.

Similar to the SBTH algorithm in part II, none of the operations of the RODE solver applied to the KT model is compute-bound. However, GPUs perform very well executing these operations being latency- (normal random number generation and realization of the OU process) and memory-bound (averaging) because the GPU's warp schedulers hide latency with computations of other warps and memory bandwidth of GPUs is very high (see tables 2.1 and 2.2). So far, we omitted the computational performance of the CPUs in the utilized GPU clusters and just used them for communication purposes. The final part IV also incorporates these computing devices leading to a hybrid implementation of the LBM, an alternative approach to describe fluid dynamics, fully exploiting the heterogeneity of GPU clusters. In contrast to the RODE solver, permanent communication between all computing devices, CPU and GPUs, is indispensable.

Part IV.

Scalability on heterogeneous systems of the lattice Boltzmann method

One of the most relevant applications of scientific computing, especially in industry, is the simulation of the behavior of fluids called computational fluid dynamics (CFD). Performing numerical fluid simulations is computationally very demanding, especially in 3D, due to steadily rising demands in spatial, and hence temporal, resolution, larger and more complex domains, and the tracking of phenomena such as turbulence. There are several models describing the nature of fluids, the most prominent ones are the Euler and Navier-Stokes equations [171]. An alternative model is the lattice Boltzmann method (LBM) which has very convenient properties when it comes to parallelization and intrinsically uses Cartesian grids. On the basis of the LBM, best practices and techniques to achieve scalability on large-scale heterogeneous systems are the central object of this subsequent part. As a result, a holistic hybrid reference implementation applying these practices and techniques is provided and discussed.

From a modeling point of view, the LBM is an alternative approach to describe the physical values of a fluid such as flow velocities and fluid densities. Instead of discretizing these values in time and space, the LBM introduces virtual particles and deals with the likelihood these particles move along discretized directions. It can be shown that for a small time scale and a corresponding small spatial scale, the LBM leads to the macroscopic Euler and Navier-Stokes equations, respectively, as a formal limit [55, 73]. Hence, the LBM is a different access to these differential equations.

From a hardware point of view, this part demonstrates how to fully exploit the computational resources of large-scale heterogeneous systems to satisfy the computational demands of CFD on the basis of the LBM. Again, multiple levels of parallelism are applied. On the lowest level, this means the parallel implementation and optimization of the LBM for CPUs and GPUs. Moreover, a domain decomposition to evenly distribute work among different computing devices is given. Such a domain decomposition raises the need for a communication scheme to exchange data of subdomain boundaries forming the highest level of parallelism. A further contribution is a tailored performance model predicting the runtime and scalability of the presented hybrid LBM implementation and identifying its bottlenecks.

To fully exploit large-scale heterogeneous systems, scalability is indispensable. It is achieved by, among other things, hiding communication times with computations and by keeping these communication times as short as possible. So, the processing of the domain is subdivided in two phases: Initially, the data required by neighboring MPI processes is processed which is then communicated while the remaining computations are executed. For both operations, the same kernels, and thus functionality, are applied but on different parts of the simulation domain at different times.

We follow an approach where all different kinds of computing devices perform the same actions, namely the operations of the LBM, thus, the functionality of the CPU and GPU kernels is the same. Alternatively, different tasks can be tackled by the most suited computing device (cf. [165, 82, 240]) or only a minor workload is assigned to the CPU (cf. [219]). In contrast, we aim to fully exploit all available computing devices at all times. Utilizing heterogeneous systems requires hybrid programming. The code for the CPU is written in C++ utilizing OpenMP to simultaneously run on multiple CPU cores.

GPU kernels are developed with CUDA and the distributed memory parallelization is achieved with MPI. The usage of OpenCL is obvious because CPU and GPU kernels implement the same functionality. However, since the support of OpenCL on the latest GPU clusters is very poor or even missing, we choose CUDA instead of OpenCL.

There are various aspects in the context of the LBM and HPC not being covered in this work. We do not modify the LBM in any way but apply it as given in literature. Only regular and static grids are considered to simulate single-phase flows. Such kinds of grids do not require load-balancing during runtime and a static resource assignment is possible in advance. Discussions on adaptive mesh refinement (AMR) for the LBM are provided in [202, 76]. Dynamic grids for LBM are dealt within [175]. The LBM for multi-phase flows is object of investigation in [217, 226], for free surface flows in [205, 124, 252]. Multi-phase flow in combination with AMR is investigated in [231]. The major goal of this part is to show how to achieve scalability on large-scale heterogeneous systems with the LBM. Hence, scenarios are limited to benchmark applications such as lid-driven cavity [39, 86] instead of simulations dealing with free surface flows [230] or complex domains [108].

The hybrid LBM code used for demonstration in the following four chapters bases on work by Schreiber [209, 210] whose GPU kernels are used. They are ported to and parallelized for the CPU by us. A first multi-GPU version of Schreiber's code using MPI was implemented by Bakhtiari [20] and extended by us with CPU kernels, simultaneous memory copies, non-blocking communication, and communication hiding. The performance model bases on Feichtinger's work [77, 78] but is refined by us to model our approach of domain decomposition and communication.

The notation from previous parts II and III does not apply to this part IV. Some of the identifiers are reused to formulate the LBM. Since all GPU kernels are written in CUDA, analysis and benchmarks are carried out on NVIDIA GPUs. None of the techniques and concepts presented in the following are limited to CUDA, NVIDIA hardware, or GPU clusters but can also be adopted to parallel computing devices in general and heterogeneous setups. To represent a broad variety of large-scale heterogeneous systems, experiments are carried out on the GPU clusters Hydra, TSUBAME2.5, and Piz Daint. They differ in the number of CPUs and GPUs per node, cores per CPU, GPU architecture, and software such as CUDA version and MPI implementation (cf. table 2.5).

The remainder of this part is structured as follows: Chapter 13 provides a short introduction to the LBM, explains the basic idea behind it, and recaps a procedure to implicitly synchronize the particular steps of the LBM to enable embarrassing parallelism. These concepts are necessary to follow the parallelization of the LBM in chapter 14 including optimizations for the CPU and GPU and introducing an efficient and scalable communication scheme. A brief overview on performance modeling in general and the tailored performance model for our LBM implementation are given in chapter 15. Strong and weak scaling scenarios are used to experimentally validate the performance model and to obtain benchmark results for single- and multi-computing device setups in chapter 16 showing scalability on up to 2048 GPUs and 24,576 CPU cores.

13. The lattice Boltzmann method and its serial implementation

The LBM is a popular, typically memory-bound, procedure of CFD for fluid simulation briefly sketched in this chapter. Major principles and important terms of the LBM are presented to have a solid basis for the parallelization techniques listed in the following chapter 14. This chapter is written from a computer scientist point of view with a focus on data structures instead of mathematics including implementation aspects not related to parallelization. A derivation of the LBM is given by [103]. We use the nomenclature from [174] whose topical structure we follow in the next two sections. For a deeper insight in the LBM, we refer to [56, 251, 6, 224].

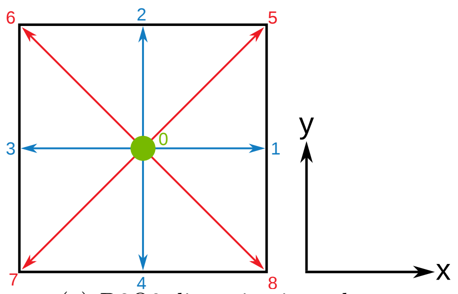
Instead of particle-based (e.g. molecular dynamics) and continuum (e.g. Navier-Stokes) methods, the LBM is a grid-based mesoscopic method. Historically, it originates from cellular and lattice gas automata [107] and is a particular simplified discretization of the Boltzmann equation [103]. The *(lattice) cells* of the Cartesian grid are squares in 2D and cubes in 3D, thus, the regular grid generation is simple. Physical values such as flow velocities and fluid densities are not directly assigned to the lattice cells but can be derived from *probability densities*, also called *density distribution functions*, densities of virtual particles discretized along specific *lattice velocities* associated with each lattice cell. Due to this kind of discretization, updates can be performed local[1] (only requiring data from the current and directly neighboring lattice cells) because the lattice velocities are chosen such that virtual particles move at most to adjacent lattice cells. This makes it very easy to parallelize the LBM cell-wise. The current values of the probability densities form the state of the simulation.

A more detailed look on probability densities and discretized directions is given in section 13.1. Following this, section 13.2 explains the modeling of fluid dynamics by alternating *collision*, also called *relaxation*, and *propagation*, also called *streaming*, steps. Finally, a memory-efficient storage scheme for the probability densities is presented in section 13.3.

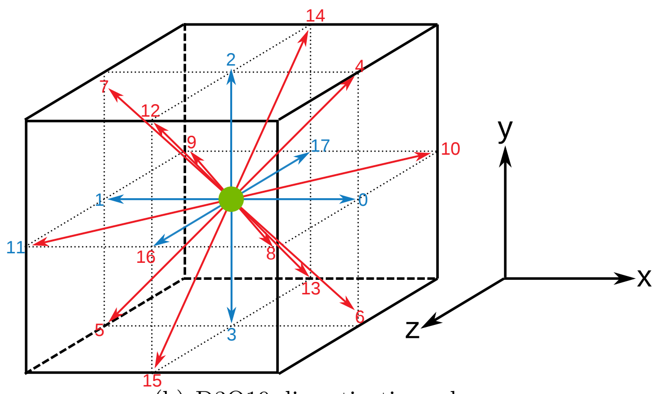
13.1. Discretization schemes

The LBM assigns probability densities $f_i(\mathbf{x}, t), i = 1, \dots, q$ to the center of every lattice cell. So, q values have to be managed per cell. They model the probability that a virtual particle moves along the direction of lattice velocity $\mathbf{c}_i$ within a small region around $\mathbf{x}$ at time t. The lattice velocities $\mathbf{c}_i$ are chosen such that the virtual particles traverse

[1]The property of local updates requires a suited collision operator, cf. section 13.2.



(a) D2Q9 discretization scheme



(b) D3Q19 discretization scheme

Figure 13.1.: Discretization schemes for the LBM using $q = 9$ probability densities in 2D (subfigure 13.1(a)) and $q = 19$ probability densities in 3D (subfigure 13.1(b)), respectively. Blue arrows represent probability densities with lattice speed $\mathbf{c}_i = 1$, red arrows with lattice speed $\mathbf{c}_i = \sqrt{2}$, and green dots with lattice speed $\mathbf{c}_i = 0$. The orientation of the corresponding coordinate systems is indicated in the right parts of the subfigures.

exactly one lattice cell per timestep dt. Depending on the number d of dimensions and q of probability densities, the notation DdQq is used to describe the actual discretization scheme. For example, figure 13.1 depicts the D2Q9 and D3Q19 discretization scheme. In this work, the D3Q19 discretization scheme is applied. For clarity, the D2Q9 scheme is used for illustration throughout the remainder of this thesis. There are also alternative discretization schemes such as D3Q15 or D3Q27 with different numerical properties and performance behavior [160].

The macroscopic quantities fluid density $\rho(\mathbf{x}, t)$ and flow velocity $\mathbf{u}(\mathbf{x}, t)$ are obtained from the probability densities via

$$\rho(\mathbf{x}, t) = \sum_{i=0}^{q} f_i \tag{13.1a}$$

and

$$\rho(\mathbf{x}, t)\mathbf{u}(\mathbf{x}, t) = \sum_{i=0}^{q} f_i \mathbf{c}_i \tag{13.1b}$$

with $\mathbf{x}$ a position in space and t a point in time. Due to the discretization of the spatial domain in cells and the usage of probability densities, all quantities are dimensionless. Instead, the spacial mesh size is scaled to one and so is the timestep.

13.2. Collision & propagation

The LBM evolves the system by alternating collision and propagation steps modeling the diffusion and convection of the *lattice Boltzmann equation*

$$f_i(\mathbf{x} + \mathbf{c}_i dt, t + dt) = f_i(\mathbf{x}, t) + \Delta_i(f - f^{eq}). \tag{13.2}$$

Separating the lattice Boltzmann equation (13.2) in diffusion and convection leads to the two subsequent algorithmic steps collision

$$f_i^*(\mathbf{x}, t) = f_i(\mathbf{x}, t) + \Delta_i(f - f^{eq}) \tag{13.3a}$$

and propagation

$$f_i(\mathbf{x} + \mathbf{c}_i dt, t + dt) = f_i^*(\mathbf{x}, t). \tag{13.3b}$$

Collision steps model the interaction of the virtual particles expressed by the *collision operator* $\Delta_i(f - f^{eq})$. Performing collisions requires computations. Propagation steps describe movement of the particles and, thus, lead to memory copy operations from and to other cells. For the collision steps it is assumed that the thermodynamical system only slightly deviates from its equilibrium state given by the *discretized equilibrium functions* f_i^{eq}. There are various formulations for f_i^{eq}, e.g. given in [102] and [10] but we stick to the standard polynomial form

$$f_i^{eq}(\rho, \mathbf{u}) = w_i \rho \left(1 + \frac{\mathbf{c}_i \mathbf{u}}{c_s^2} + \frac{(\mathbf{c}_i \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}}{2c_s^2}\right) \tag{13.4}$$

with c_s denoting the speed of sound and w_i being *lattice weights.* Constraints which have to be fulfilled by the weights w_i are given in [174].

There are various collision models for $\Delta_i(f - f^{eq})$. We use the most common choice for a collision operator, the Bhatnagar-Gross-Krook (BGK) scheme [31], also called single-relaxation-time scheme (SRT), denoted by

$$\Delta^{BGK}(f - f^{eq}) = -\frac{1}{\tau}(f - f^{eq}) \tag{13.5}$$

with τ the relaxation time directly related to the kinematic[2] viscosity $\nu = c_s^2 dt(\tau - 0.5)$ of the fluid. The relaxation time has to be chosen properly such that a positive viscosity results, i.e. $\tau > 0.5$. To keep the simulations stable, it is common to use a $\tau \in (0.5, 2)$. A further example for a collision operator is the multiple-relaxation-time (MRT) scheme [71]. Both, the BGK and the MRT collision operator allow local updates only involving the local and directly neighboring cells.

One further aspect not discussed in this chapter is the treatment of boundary conditions by the LBM. They have an influence on the function of collision and propagation steps. Our implementation supports some boundary conditions such as no-slip conditions but since this work focuses on optimization and parallelization on large-scale heterogeneous systems, we refer to [88, 209, 174].

[2]The kinematic viscosity is the ratio of the dynamic viscosity μ to the density of the fluid ρ.

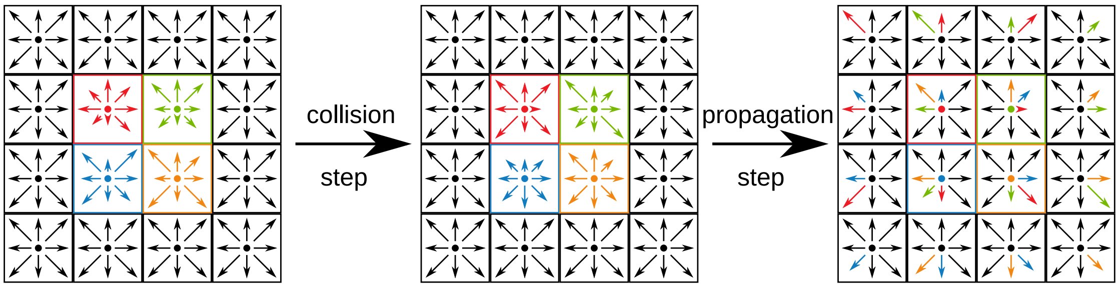


Figure 13.2.: By memory accesses affected probability densities of four processed cells (colored in red, green, blue, and orange) using the A-B memory layout pattern illustrating the unaltered procedure of collision and propagation steps. Different lengths of same-color arrows in the same direction represent an update of the probability densities due to a collision step.

13.3. Memory layout pattern

After briefly introducing the LBM in the previous two sections, this chapter is continued with its serial implementation. In particular, a storage-efficient memory layout pattern enabling coalesced memory access and caching is presented. Further aspects regarding the serial implementation of the LBM are discussed in [241].

In figures 13.2 and 13.3, different values of the probability densities are indicated by different lengths of the lattice velocities. Figure 13.2 depicts one step of the LBM consisting of a collision and a propagation. As illustrated by the four colors red, green, blue, and orange, a parallelization of the LBM assigns one thread to one cell as a most fine-granular approach. Same color means handled by the same thread. To avoid race conditions, this straight-forward parallelization approach requires two dedicated instances of all probability densities in memory because hazards such as reading an already updated probability density by a different thread is possible. The proposed solution of keeping two copies of the data is called *A-B memory layout pattern*. Data is always read from one copy and written to the other before source and destination sets are switched. This resolves the race conditions at the cost of halved usable memory.

A much more memory-efficient approach is the *A-A memory layout pattern* proposed in [19] which only uses one copy of data in memory and, thus, does not introduce any memory overhead. Race conditions are avoided by a rearrangement of the collisions and propagations of two consecutive (first an odd, then an even) steps of the LBM. α-steps as depicted by subfigure 13.3(a) are just the collision of the odd regular LBM steps while β-steps as shown in subfigure 13.3(b) implicitly combine the propagations of the odd regular LBM steps and the collisions and propagations of the even regular LBM steps. Hence, both, α- and β-steps read from and write to the same memory locations of the probability densities. During α-steps, a thread only deals with data corresponding to its assigned cell. A contrary behavior is performed during β-steps where a thread only deals with data corresponding to adjacent cells (except the probability densities with lattice

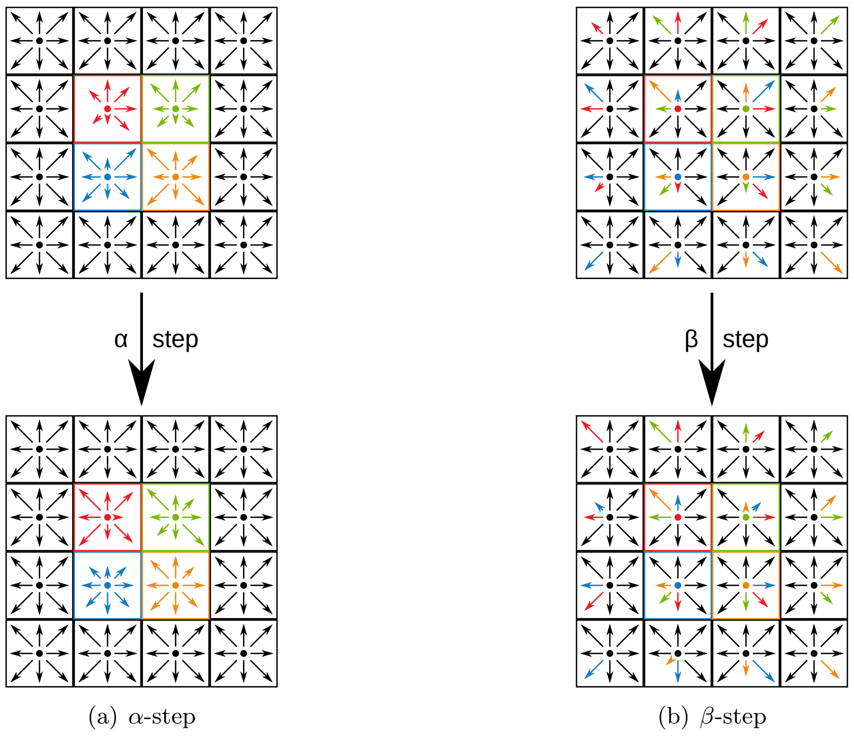


Figure 13.3.: By memory accesses affected probability densities of four processed cells (colored in red, green, blue, and orange) using the A-A memory layout pattern. Different lengths of same-color arrows in the same direction represent an update of the probability densities due to a collision step. Subfigure 13.3(a) illustrates the α-step consisting of one collision step, subfigure 13.3(b) depicts the β-step consisting of two propagation and one collision step.

speed 0). So, synchronization is only required between α- and β-steps but not within them. The major drawback of the A-A memory layout pattern is that after odd numbers of steps, the simulation is not in a valid state because more collision than propagation steps are executed. Only after β-steps, the simulation is in a consistent state. The A-A memory layout pattern is used for our LBM implementation. A deeper investigation on the A-A memory layout pattern as well as additional memory layout patterns for the LBM are given in [250]. When it comes to higher geometrical flexibility and local mesh refinement, the esoteric twist (EsoTwist) [84] is a more sophisticated alternative to the A-A memory layout pattern.

To achieve high performance, the arrangement of the probability densities in memory is crucial. Intuitively, $f_i, i = 1, \dots, q$ is stored cell-wise avoiding coalesced memory access on the GPU and efficient caching on the CPU. Hence, storing the probability densities of one particular direction continuously in memory is the better strategy applied to both, the CPU and GPU kernels of our implementation. While the first strategy corresponds to an array of structures (AoS) design, the second represents an example of a structure of arrays (SoA).

Probability densities are interpreted as a three-dimensional array and continuously stored in memory. The indices `i`, `j`, and `k` are used to iterate in x-, y-, and z-direction. `i` is chosen to be the fastest, `j` the second fastest, and `k` the least fastest running index, thus, for a fixed `j` and `k`, data in x-direction is continuously aligned in memory. This choice is not limited to the A-A memory layout pattern but is generally applicable.

With these information about the LBM at hand, its parallelization on different levels of a large-scale heterogeneous cluster is presented in the next chapter.

14. Parallelization of the lattice Boltzmann method

With the foundations of the LBM at hand, our contributions in terms of parallelization of the LBM on large-scale heterogeneous systems, i.e. GPU clusters, are presented in this chapter. Due to the complexity of these systems such as different types of computing devices each requiring different approaches of parallelization and numerous communication links, there are various possibilities for optimization. The hardware side offers multiple levels of parallelism and due to different types of computing devices, not all of them behave in the same way and require different programming models. This also increases the complexity for the application developer.

Before discussing the parallelization techniques applied by us, an overview on already existing successful attempts incorporating more than one parallel technology is given, not only restricted to the LBM. Neglecting different types of computing devices, Debudaj-Grabysz et al. [68] and Rabenseifner et al. [192] present classical examples of hybrid parallelization using OpenMP for the shared memory and MPI for the distributed memory parallelization. These techniques can also be utilized for the LBM as shown by Linxweiler in [141]. Considering single-GPU setups, first attempts were carried out by Tölke et al. [232] using the D3Q13 discretization scheme. Bailey et al. [19] could experimentally validate that a GPU implementation of the D3Q19 discretization scheme delivers superior performance in comparison to an optimized implementation for the CPU. Since the LBM is memory-bound, most optimizations target memory issues, e.g. presented in [184, 93] and already mentioned in sections 13.1 and 13.3. Considering only one heterogeneous computing node, Feichtinger et al. present a patch-based LBM approach to assign work to the CPU and to the GPU. The LBM can also be incorporated as the fluid solver in fluid-structure interaction (FSI) applications which benefit from heterogeneous computing nodes [233], too. There, similar to our implementation, both computing devices are executing the same functionality. An implementation of the LBM for small-scale GPU systems is reported by Obrecht et al. in [185]. Moving to large-scale systems, communication has to be overlapped by computations as shown by Wang et al. in [239]. The uncommon D2Q37 discretization scheme is applied by Calore et al. [53] who demonstrate how multi-layered boundaries are exchanged between multiple GPUs. Xiong et al. were the first who also use OpenMP for multi-core CPU parallelization in addition to the usage of multiple GPUs [253]. In this work, only static resource assignment is applied but Hagan et al. [94] provide a general load balancing mechanism for GPU clusters.

All these works deal with aspects captured by us in this chapter but none of them provides the full level of complexity of our implementation. The combination of the techniques listed in this chapter allows the full utilization of all computing capabilities of a large-scale GPU cluster. It is not limited to one particular GPU cluster but can be generalized to common GPU clusters such as those listed in table 2.5. The major goal is scalability being achieved as successfully confirmed in chapter 16 without losing sight of computing device-level performance.

To better orientate in 3D, the terms "left", "bottom", and "back" are used for smaller lattice cell indices in x-, y-, and z-direction, respectively. Analogously, the terms "right", "top", and "front" are used for larger lattice cell indices in x-, y-, and z-direction, respectively. Our implementation works with 3D domains but illustrations in this part are drawn in 2D for clarity, thus, cuboids are visualized as rectangles.

The remainder of this chapter is structured as follows: First, the decomposition of the simulation domain and the assignment of lattice cells of subdomains to the computing devices is illustrated in section 14.1. Section 14.2 deals with the individual properties and optimizations of the LBM kernels for GPU and CPU, thus, with the lower levels of parallelism. Finally, a communication scheme is presented in section 14.3 incorporating communication between GPUs and CPUs, communication between different MPI processes minimizing the number of communication partners per process, and overlapping of communication with computations. These are aspects of the highest level of parallelism. Measurements to validate the scalability of the presented techniques are given in chapter 16.

14.1. Domain decomposition

We parallelize the LBM according to a data-parallel approach: Different chunks of data, i.e. sets of lattice cells with their probability densities, are assigned to different processing elements. The entire domain in subdivided in equally-sized connected *subdomains* of cuboid shape as depicted by figure 14.1 for a 2D example with 40×24 lattice cells. There is no overlap of subdomains. Hence, the size of the whole domain is divisible without remainder by the size of a subdomain in each dimension. This forms the highest level of parallelism. Every subdomain is further subdivided by a plane in xz-direction. The resulting upper cuboid is assigned to one GPU, the lower cuboid to at least one CPU core colored in green and blue in figure 14.1. Accordingly, they are called *GPU-* and *CPU-part of the subdomain*. The computation of multiple lattice cells on particular computing devices introduces additional levels of parallelism. All GPU-parts of the subdomains are of the same size, hence, also all CPU-parts of the subdomains are of the same size. The ratio between GPU- and CPU-part of a subdomain can be arbitrarily chosen at the beginning of the simulation but is then fixed during runtime. This makes it possible to adapt the simulation to computing devices with any ratio of computational performance. Our implementation supports GPU- or CPU-parts of subdomains of size 0 in y-direction. If the GPU-part of the subdomain is of size 0 in y-direction, a CPU-only scenario results and vice versa.

Every subdomain consisting of a GPU- and a CPU-part is assigned to one MPI process. Hence, the number of subdomains, the number of utilized GPUs, and the number of MPI

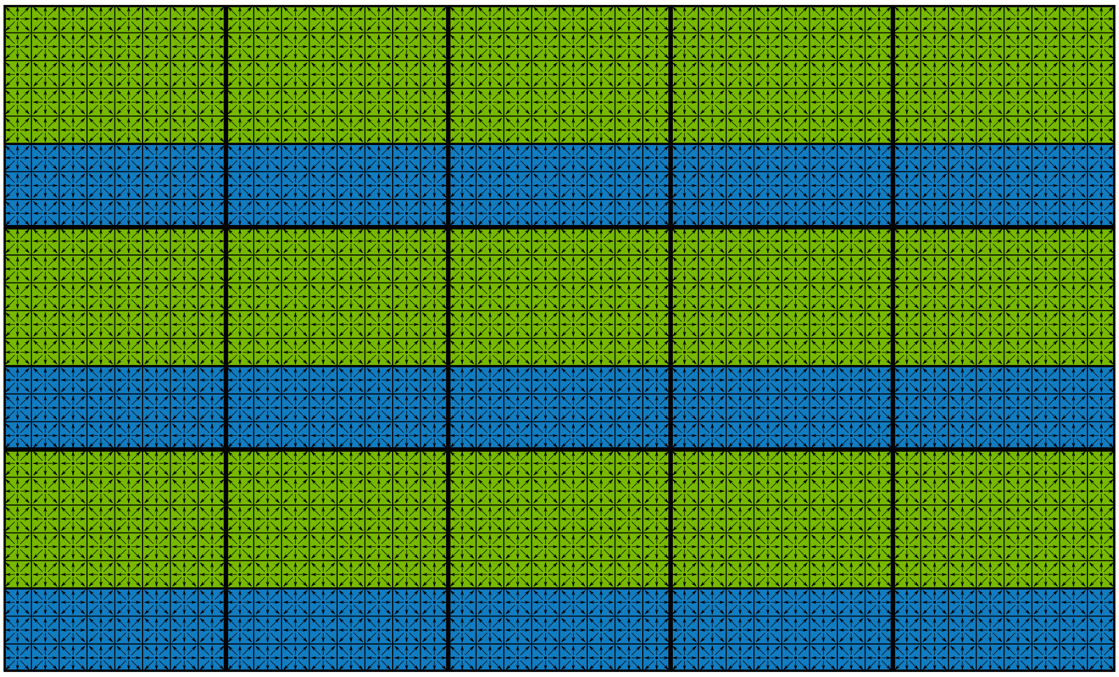


Figure 14.1.: Decomposition of a 2D domain consisting of 40×24 lattice cells in 5×3 subdomains. Each 8×8 subdomain is subdevided in a 8×5 upper part assigned to a GPU (green) and a 8×3 lower part assigned to a CPU (blue).

processes are equal. For every node of the GPU cluster, the number of executed MPI processes corresponds to the number of GPUs per node. Every MPI process uses the same number of CPU threads to process its CPU-part of the subdomain. If the number of CPU cores per node is a multiple of the number of GPUs per node, then one CPU thread can be pinned on one dedicated core. In such a case, there are no dedicated CPU cores for program control and communication and no simultaneous multithreading (SMT) such as hyperthreading is applied. Otherwise, either less threads are created so every thread can run on a separate core or oversubscription is applied leading to bad cache efficiency. No special treatment is used if there is more than one CPU per node, so non-uniform memory access (NUMA) effects are neglected. For example, on TSUBAME2.5 (see table 2.5 for specification), we launch three MPI processes on a node, each utilizing four CPU threads. Assignment of lattice cells to computing devices means that the probability densities of the cells exclusively reside in the corresponding computing device's memory. No copies of this data (except ghost cells explained in section 14.3) are managed by other computing devices.

This assignment of data to computing devices and parallel programming models such as MPI processes and CPU threads is special in terms of rigid rules but offers best performance. Alternatively, one of the following three assignment policies could be applied:

(a) One MPI process for each GPU and one MPI process for each CPU (or all CPU cores of one node)

(b) One MPI process for each GPU and one MPI process for each CPU core

(c) One MPI process for every node

The major drawback of option (a) is that GPUs and CPUs cannot communicate directly with each other but have to use MPI messages, even if they are located in the same node and could perform direct memory copy operations. Furthermore, it is much harder to evenly distribute work between GPUs and CPUs depending on their computational performance. Option (b) comes with the same drawbacks as option (a). In addition, communication between different CPU cores cannot be done in a shared memory manner but requires MPI messages. However, options (a) and (b) keep the complexity of the implementation low because synchronization is implicitly achieved by MPI mechanisms. Finally, option (c) gives full control for node-level optimizations to the application developer at the cost of a high degree of code complexity and synchronization requirements. Hence, the one MPI process per GPU- and CPU-part of a subdomain is utilized for the LBM implementation.

During the simulation, neighboring subdomains have to exchange data of their *boundary cells*, the outermost layer of cells of a subdomain; the remaining cells of a subdomain are called *inner cells*. This communication requires memory copy operations between the computing devices as well as within the host memory and MPI message passing. These mechanisms are explained in section 14.3.

14.2. Computation of the GPU- & CPU-part of a subdomain

In this section, the parallelization of the LBM on computing device level and computing device-specific optimization features are shown. GPU and CPU kernels perform the same operations, the data layout is the same for both computing devices. Hence, no conversation is necessary if data is copied from the GPU to the CPU and vice versa. Since the A-A memory layout pattern is applied (cf. section 13.3), no additional mechanisms for synchronization are required because synchronization occurs implicitly by α- and β-steps. For both, α- and β-step, there is each a dedicated kernel for the GPU and the CPU also implementing the necessary functionality to handle boundary conditions in the correct way. They implement the BGK collision operator and just differ in the way probability densities are read and written (see figure 13.3). All kernels cannot only be applied to entire subdomains but also to arbitrary sub-cuboids of a subdomain, e.g. to boundary or inner lattice cells. So, there are no dedicated kernels for boundary and inner lattice cells and code duplication is avoided.

14.2.1. Lattice Boltzmann method kernels for the GPU

The two GPU kernels for the α- and β-step are taken from [209]. For both kernels, one GPU thread is assigned to one lattice cell leading to the desired high amount of threads during runtime to hide memory latencies. To achieve coalesced memory access, continuous threads are assigned to continuous lattice cells. So, according to the memory layout pattern, index `i` for the x-direction is the fastest, index `j` for the y-direction is the second fastest, and index `k` for the z-direction is the slowest running index. For every probability density of a lattice cell, a dedicated register is used. Hence, it is not necessary to utilize shared memory to increase performance because data is once read from global memory to the registers at the start of the kernel and once written back to global memory at the end of the kernel. In the meantime, all computational operations can be performed directly using registers. On the Fermi and Kepler GPU architecture (cf. table 2.1), the unused shared memory can be exploited to increase the size of the L2 cache. However, this approach leads to a high per thread register consumption. Since the treatment of different boundary conditions is handled by the same kernels of the α- and β-step, the register pressure becomes even higher due to higher complexity of the kernels. This leads to a duality when reaching for high occupancy. On the one hand, small thread block sizes can limit occupancy because multiprocessors have a maximum number of blocks that can be active at a time. If the thread block size is small, the product of threads per block and maximum number of blocks that can be active at once per multiprocessor is too small to reach full occupancy. On the other hand, large thread block sizes lead to high register usage per block also limiting occupancy. A detailed discussion on this issue can be found in [176]. The best performing thread block size in such a scenario has to be determined experimentally as done in subsection 16.1.1 for the Tesla P100 utilized in Piz Daint. The performance of the GPU kernels when dealing with boundary cells highly depends on the face direction of the outermost lattice cell layer of the GPU domain. If the face points in y- (bottom and top face, respectively) or z-direction (back and front face, respectively), the performance is much higher than in x-direction (left and right face, respectively). This difference in performance originates from the memory layout pattern which uses `i` in x-direction as fastest running index. It only allows coalesced memory access if data to be loaded or stored is aligned continuously in x-direction being the case for layers in the xz- (bottom and top face) and xy-plane (back and front face) but not for layers in the yz-plane (left and right face). Since the LBM with the D3Q19 discretization scheme is memory-bound, its performance highly depends on the possibility of coalesced memory access.

14.2.2. Lattice Boltzmann method kernels for the CPU

The two kernels for the CPU originate from the α- and β-step kernel for the GPU. Instead of assigning lattice cells to threads, three nested loops iterate over the cells. The innermost loop iterates in x-direction, the outermost loop iterates in z-direction to fit the memory layout pattern. *Blocking* [127] is applied to increase the cache efficiency of the code for the CPU. Since cache lines are a contiguous chunk of linear data and the fastest running index is in x-direction, the best performing block sizes are big in x-direction and small in z-direction. Benchmark results for the Xeon E5-2690v3 utilized in Piz Daint using different blocking sizes are given in subsection 16.1.2.

```
1  ...
2  #pragma omp parallel
3  #pragma omp single
4  {
5      for (int k = 0; k < O; k++)
6      {
7  #pragma omp task
8          {
9              for (int j = 0; j < N; j++)
10             {
11                 for (int i = 0; i < M; i++)
12                     // do cell-wise work
13     }   }   }
14     // do work while OpenMP tasks are still running
15 }
16 // OpenMP task work is finished
17 ...
```

Listing 14.1: Code example for CPU kernels to use OpenMP tasks for parallelization. Clauses to configure the visibility of variables are neglected.

The parallelization of the CPU kernels is realized with OpenMP as demonstrated by an example given in listing 14.1. OpenMP tasks [15] are utilized, originally introduced in OpenMP 3.0 to deal with asymmetric work in OpenMP programs but also applicable to our LBM implementation. The most common way to parallelize loops with OpenMP is using `#pragma omp parallel for` but this construct leads to a loss of program control until the parallel section is finished by all threads. Even with the `nowait` clause, program control is returned not before the first thread finished the parallel section. In contrast, OpenMP tasks allow to spawn non-blocking tasks forming a pool of work being processed by OpenMP threads within a parallel section (lines 2–15). As soon as all OpenMP tasks are issued, work is continued within the parallel region (line 14). During this time, memory copy operations and MPI communication can be issued while the OpenMP tasks are still executed in parallel. Hence, OpenMP tasks are preferred to `#pragma omp parallel for` not because they provide superior performance but offer more flexibility to the application developer. An implicit synchronization assuring all OpenMP tasks are finished occurs after the parallel region (line 17). A new OpenMP task is issued with `#pragma omp task` (line 7). In the example of listing 14.1, for every iteration of the outermost loop (line 5), a new OpenMP task is created. Every OpenMP task deals with full iterations over the inner two loops (lines 9 and 11) and, thus, processes one plane in xy-direction of the domain. As for multi-threaded programming in general, the number of OpenMP tasks should be as small as possible (and the workload per OpenMP task should be as big as possible) to keep overhead small but enough OpenMP tasks should be created to utilize all available CPU cores. Hence, creating new OpenMP tasks for iterations of outer loops is more reasonable than issuing new OpenMP tasks in inner loops. Instead of creating new OpenMP tasks for iterations of loops, we initiate one OpenMP task for every block used to increase the cache efficiency. Using OpenMP tasks is the only technique utilized to parallelize the CPU kernels, no vectorization is applied.

Similar to the GPU kernels, the performance of the CPU kernels when handling boundary cells depends on the orientation of the outermost layer of lattice cells of a subdomain. If they point in y- or z-direction, the performance is much higher than in x-direction. The reason for this performance gap is again the memory layout pattern. Analogously to the possibility of coalesced memory access on the GPU, it enables efficient caching for data of a xz- and xy-plane but not of a yz-plane.

14.3. Communication scheme

In this section, the parallelization of the LBM on process level and the optimization of communication are shown. Subdomains have to exchange probability densities of boundary cells because the LBM requires data of neighboring cells to perform its operations. So a single layer of ghost cells is added around each GPU- and CPU-part of a subdomain always containing the probability densities stemming from adjacent subdomains essential to update the boundary cells. This leads to two phases during each step, holding for the classical collision and propagation step but also for the α- and β-step: First, computations are performed only for the boundary cells of each subdomain. Second, data is copied from the boundary cells to the ghost cells of the corresponding proximate subdomains. In the meantime, the computation of the inner cells can be carried out to hide the communication times with computation times. Specialties of the computations are already discussed in previous section 14.2, features of the communication are presented in the following.

There are two types of communication which have to be realized by an MPI process:

(a) Exchange of data inside MPI processes. This includes copy operations between the GPU- and CPU-parts of a subdomain as well as copy operations to and from communication buffers required for the second type of communication.

(b) Exchange of data between MPI processes.

For communication of type (a), the CUDA function `cudaMemcpy3DAsync()` is applied. It enables coping data of any cuboid-shaped region of a subdomain. Source and target can either be the device or the host. In our case, these cuboidic-shaped regions correspond to the data of lattice cells to be transfered and always have a thickness of one cell. Internally, `cudaMemcpy3DAsync()` delegates copy operations to successive calls of `cudaMemcpy()` or `memcpy()`. For every chunk of data contiguously stored in memory following the fastest running index, one execution of `cudaMemcpy()` or `memcpy()`, respectively, is issued. As seen for GPU and CPU kernels, this has a significant impact on performance depending on the orientation of the plane of data to be copied. Assuming a GPU- or CPU-part of a subdomain of size $M \times N \times O \in \mathbb{N}^3$, a copy operation in x-direction leads to $N \cdot O$ function calls, each transferring one element. This is much more inefficient than copy operation in y- or z-direction where O and N, respectively, calls are issued each transferring M elements at a time. If data has to be exchanged

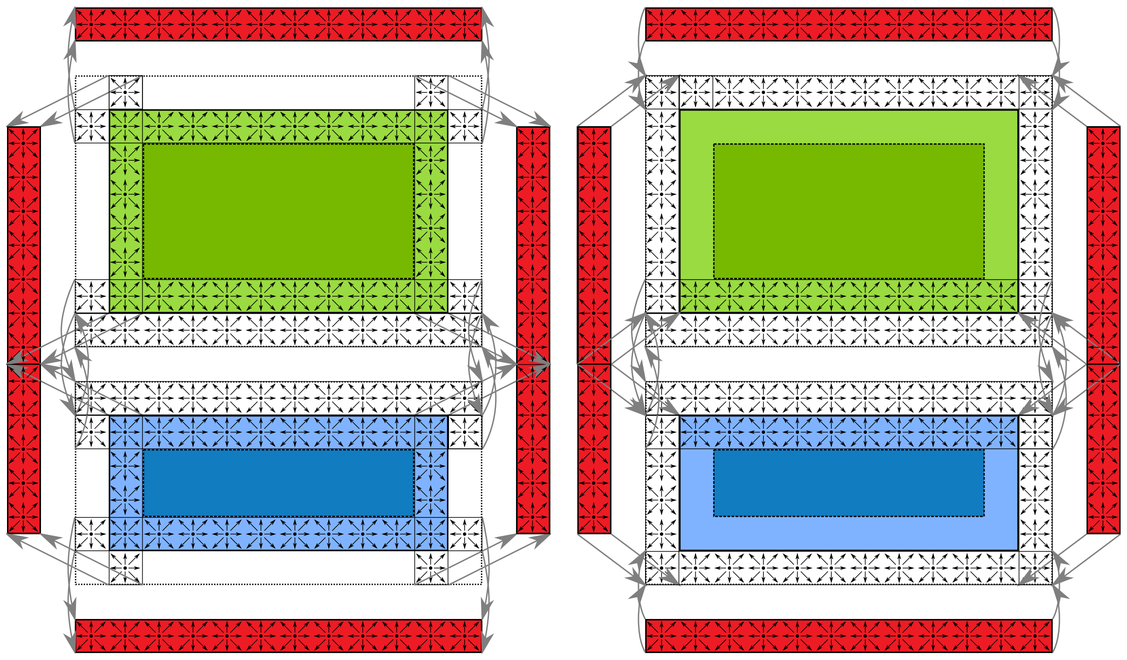
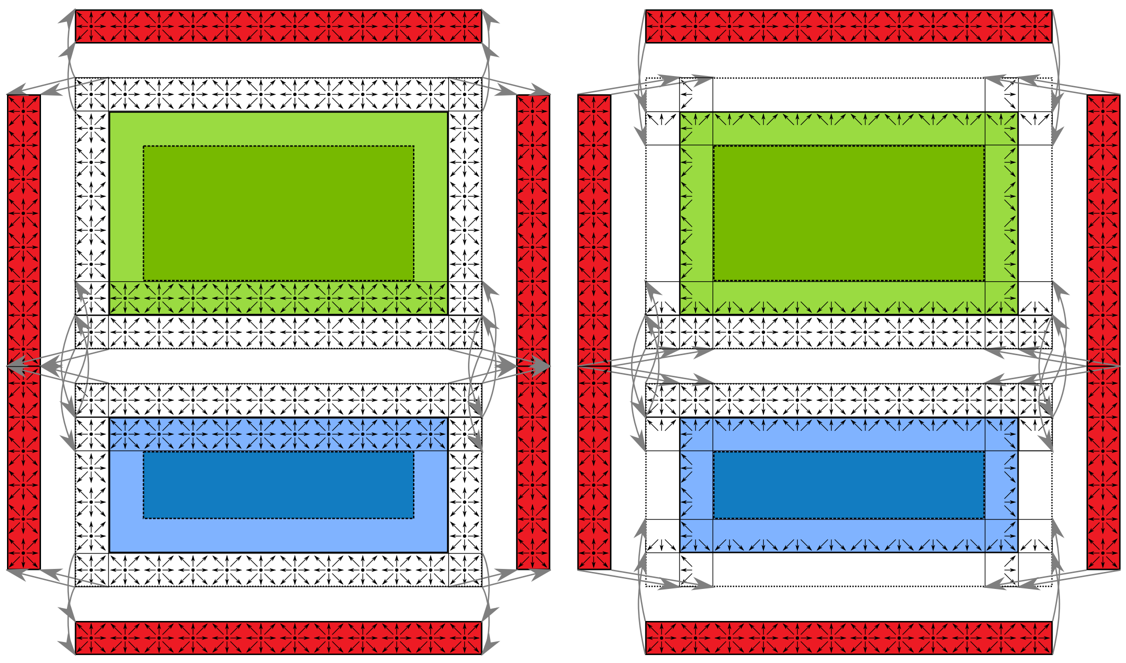

(a) Collecting data for communication buffers. (b) Copying data from buffers to ghost cells.

Figure 14.2.: Collection (subfigure 14.2(a)) and distribution (subfigure 14.2(b)) of data during an α-step for one MPI process. GPU- and CPU-part of the subdomain are colored in green and blue, respectively. Boundary cells are colored in light-green and light-blue, respectively, ghost cells are colored in white. Memory for the communication buffers is colored in red. Gray arrows indicate copy direction and location. Only those probability densities are depicted by black arrows which are actually copied.

between the GPU- and the CPU-part of a subdomain, it can be directly copied via `cudaMemcpy3DAsync()`. To send data to adjacent subregions, it has to be collected from the GPU- and the CPU-part of the subdomain in dedicated communication buffers, allocated and managed by the application developer, before it can be transferred via MPI. This collection step (and the distribution of data once it is received) is also achieved with `cudaMemcpy3DAsync()`. It performs memory copy operations asynchronously meaning that program control is immediately returned once copying is issued. Hence, computations can be performed while data exchange inside MPI processes are simultaneously performed in the background.

With the boundary data copied to the communication buffers, communication of type (b) can be performed. To overlap communication with computation, the non-blocking MPI functions `MPI_Isend()` and `MPI_Irecv()` are utilized. Dedicated communication buffers for send and receive operations avoid race conditions. All communication between MPI processes can be maintained with point-to-point communication because



(a) Collecting data for communication buffers. (b) Copying data from buffers to subdomains.

Figure 14.3.: Collection (subfigure 14.3(a)) and distribution (subfigure 14.3(b)) of data during an β-step for one MPI process. Color coding and meaning of data transfer arrows are equal to figure 14.2. Only those probability densities are depicted by black arrows which are actually copied.

only neighboring subdomains have to interact with each other. No global information has to be exchanged, thus, no collective operations are necessary.

Figures 14.2 and 14.3 depict a 2D representation of the per MPI process communication of the α- and β-step, separated in sending to (left subfigures) and receiving from (right subfigures) neighboring subdomains. GPU and CPU domain are colored in green and blue, inner cells are colored darker, boundary cells are highlighted. Ghost cells are colored in white and added around the actual computational subdomain. For every communication direction (left, right, bottom, top, back, and front), there are communication buffers for sending and receiving colored in red. While an α-step is performed, data is eventually transferred from boundary to ghost cells required not before the β-step because the collision operator of the α-step only deals with cell-local probability densities. During β-steps, data is transferred from ghost to boundary cells to implement the propagations as depicted by subfigure 13.3(b). Only probability densities whose lattice velocity points outward of the target subdomain have to be copied implicitly avoiding some memory transfers discussed in more detail in [20].

Actually, every subdomain has to exchange information with all neighbors in the directions of the lattice velocities. For the D3Q19 discretization scheme, these are 18 communication partners: Six adjacent to the faces of the subdomain and twelve neighboring to the edges of the cuboid. The *face neighbors* need information from an entire face, the *edge neighbors* only from a single line of lattice cells. By ordering the communication between MPI processes, communication can be reduced to the six face neighbors. First, data is exchanged between connected neighbors in x-direction (left and right), afterwards in y-direction (bottom and top), and finally in z-direction (back and front). Thereby, data designated to the, for example, right front edge neighbor is first transferred to the right neighbor in x-direction and from there to the front neighbor in z-direction. Analogously, communication with all other edge neighbors is possible. On the one hand, this idea serializes inter-process communication in three distinct phases, on the other hand, it significantly reduces the number of communication partners.

All computation and communication operations during one α- or β-step of an MPI process are summarized in figure 14.4. Vertically, figure 14.4 shows three dedicated blocks for operations on the GPU, the CPU, and for MPI. Horizontally, it shows temporal progress. The blocks for GPU and CPU are further broken down in operations for boundary cells of the six cuboid faces and inner cells. The block for MPI operations is itemized in communication directions denoted by "from . . . neighbor $\rightarrow$ . . . boundary layer". Computations on the GPU and CPU are colored in green and blue, respectively. MPI communication is colored in red. The orange bars before `MPI_Isend()` and the gray bars after `MPI_Irecv()` signal copy operations between the communication buffers managed by the application developer and the MPI buffers maintained by MPI. Different lengths of the bars indicate varying length in runtime depending on the size of the cuboid to process or effects such as coalesced memory access or caching. Bar lengths do not correspond to real-world measurements but just give a rough estimation of the actual runtimes. Black arrows depict data dependencies meaning probability densities cannot be send via MPI before they are copied from the computing devices to the communication buffers and cannot be copied to the computing devices before received via MPI. Purple arrows depict dependencies due to shared resources, for example either data from the GPU- or the CPU-part of the subdomain is written to the communication buffers at a time. During the phase for communication in y-direction, only data of the top face of the GPU-part and from the bottom face of the CPU-part of the subdomain are transferred to the communication buffers because the remaining faces in y-direction directly exchange data by utilizing `cudaMemcpy3DAsync()`, denoted by "CPU2GPU" and "GPU2CPU" in figure 14.4.. Dotted lines connect associated operations, e.g. direct copy operations between GPU and CPU or sending to and receiving from the same MPI communication partner. The latter example explicitly invokes send and receive operations instead of using `MPI_Isendrecv()`. Communication (framed by yellow areas) is carried out while the inner cells are processed. Depending on which procedure takes longer, the current step ends as soon as all communication or the treatment of the inner cells is finished.

After presenting the direction-dependent properties of the kernels for the computing devices in section 14.2 and introducing a communication scheme for the LBM on large-scale heterogeneous systems in this section, a performance model can be setup in the next chapter predicting the performance when combining these techniques.

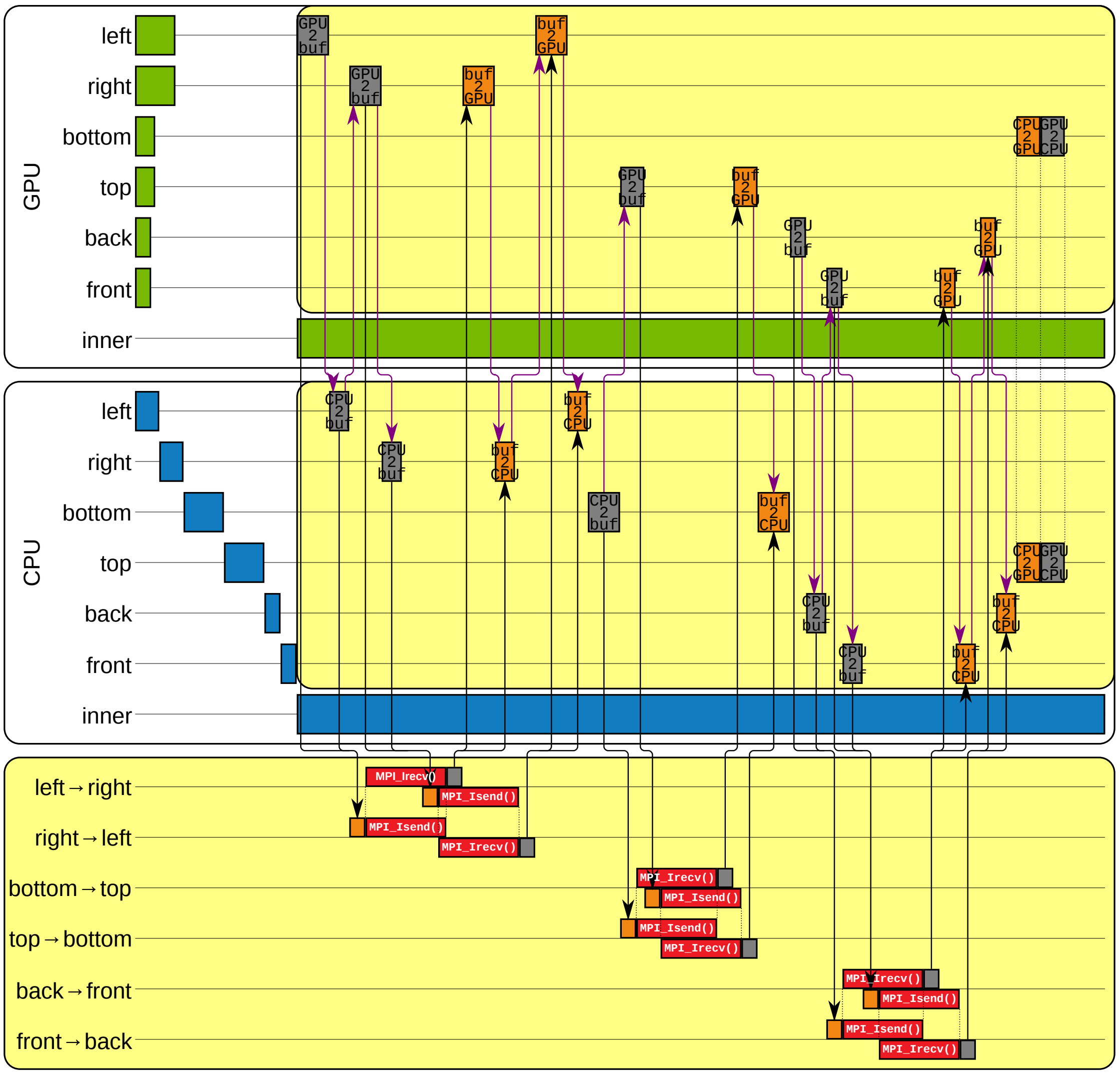


Figure 14.4.: Operations within one MPI process broken down in device (top), host (middle), and MPI (bottom) operations. Yellow background marks communication operations, white background shows computations. Green and blue bars indicate computations on the GPU and CPU, respectively. Gray bars represent copy operations to the communication buffers of the host, orange bars represent read operations from these buffers. Red bars symbolize MPI communication. Bars marked with "GPU2buf" perform a copy operation from the GPU to the communication buffers of the host and with "CPU2buf" from the CPU to the communication buffers. The vice versa copy operations from the communication buffers to the computing devices are marked with "buf2GPU" and "buf2CPU". "CPU2GPU" is assigned to copy operations from the CPU domain directly to the GPU domain and "GPU2CPU" is assigned to copy operations in the opposite direction. Black arrows indicate data dependencies, purple bars illustrate dependencies due to shared resources.

15. Performance modeling of the lattice Boltzmann method on heterogeneous systems

In this chapter, we introduce a performance model for the LBM considering for the first time the distinct properties of the different memories and communication channels in a heterogeneous system. By considering such components of heterogeneous systems, performance predictions of very high accuracy are feasible.

An often occurring question in computer science is how well a given program performs on a particular hardware architecture. Performance values of interest are for example runtime, achieved peak performance, or, especially relevant in the area of HPC, scalability. So, in the field of *performance modeling* [243], models describing features and properties of target hardware architectures are setup to predict the performance of programs executed on these particular architectures. Such performance models help to identify bottlenecks and ease the optimization for specific hardware or the adaptation of hardware to certain applications. In this chapter, a performance model for our LBM implementation with the parallel features presented in chapter 14 for GPU clusters exploiting all available computing devices and modeling all necessary communication is derived. Hence, we do not develop a performance model for hybrid parallel software on heterogeneous systems and clusters in general but present an application-tailored model.

Classical performance modeling [36, 37] utilizes tools such as queuing networks and Markov chains [151] to describe various types of hardware such as processing elements, memory buses, I/O devices, and networks. When it comes to HPC, properties such as compute- or memory-bound are of interest which can be determined, e.g., by the balance model [95]. For a complex application such as our LBM implementation, the performance of various components can be modeled: On a lower level, the particular parallel computing devices such as GPU and CPU are object of investigation. The roofline model [247, 246] incorporating the FLOPS per byte ratio for a particular operation is a very popular method for such tasks in general and in [93], a performance model tailored to LBM is presented. On a higher level, insight on the interplay of the computing devices with all aspects of copying and communicating data is desired. Within a single heterogeneous node, a model by Malony et al. [148] can be used while for a larger scale, the model by Lu et al. [142] handles a hybrid scenario similar to ours: Using CUDA and OpenMP to parallelize the GPU and CPU part, respectively, and the inter-node communication is achieved by MPI.

The main focus of part IV does not lie on the optimization of particular kernels for specific computing devices but on the scalable implementation of the LBM on large-scale heterogeneous systems by hybrid programming. Hence, the performance model in this chapter targets the above mentioned higher level, especially for strong scaling scenarios. It bases on work by Feichtinger [77, 78] but differs in two important aspects: First, the original model is adapted to match the domain decomposition, work assignment, and communication approach from sections 14.1 and 14.3. On the one hand, this modification restricts the general applicability of Feichtinger's model but on the other hand, more individual features are captured making our model more precise. Second, a generalization is realized by including the property of anisotropic behavior when it comes to the treatment of boundary cells. Depending on the orientation of the outermost layer of a subdomain, i.e. the boundary cells, the performance of updating cells and copying data from or to device or host memory varies much. This anisotropic behavior stems from the possibility of coalesced memory access and caching being feasible for boundary layers in y- and z-direction, but not in x-direction due to the utilized memory layout pattern with `i` as fastest running index in x-direction.

Our performance model estimates an upper bound for the runtime to execute an entire α- or β-step. Since the number of computations, memory accesses, and the amount of communicated data is almost equal for both types of steps, it is not necessary to do a separate modeling for the particular steps. The runtime of a single MPI process is modeled and it is assumed that the runtime of all MPI processes is equal independent from the number of running MPI processes. Instead, the size of the subdomain handled by the MPI process and, of course, the hardware properties of the GPU cluster, determine the total runtime. Such an assumption is only valid in a point-to-point communication scenario as it holds for the parallelization of the LBM. If collectives are used, the communication time does not only depend on the amount of transferred data but also on the number of communication participants. Performance values such as LUPS or FLOPS rate or memory bandwidth are not taken into account by our model. Instead, actual runtimes are incorporated which can either be derived from the even mentioned performance values or by micro-benchmarks as done in section 16.3 in the next chapter. A worst case scenario is assumed for the event that an MPI rank interacts with all six logical neighbors neglecting MPI ranks processing the boundaries of the domain and, hence, dealing with less neighbors.

With T denoting the total runtime to perform one α- or β-step, our performance model looks as follows:

$$T = \underbrace{\max(\tau^{GPU}_{boundary}, \tau^{CPU}_{boundary})}_{\text{time to compute boundary cells}} + \max(\underbrace{\max(\tau^{GPU}_{inner}, \tau^{CPU}_{inner})}_{\text{time to compute inner cells}}, t_{comm}) \tag{15.1}$$

While the first summand models the computation times of the boundary cells, the second summand describes accumulated communication times and computation times of the inner cells. It considers the hiding of communication times with computation times by the outer max() function.

All $\tau^{\text{computing device}}_{\text{region}} : \mathbb{N}^3 \rightarrow \mathbb{R}^+$ are functions indicating the runtime until a three-dimensional cuboid of lattice cells is updated, thus, specifies computation time. The runtimes refer to a "GPU" or a "CPU" as "computing device" and it is distinguished between “boundary” and “inner” cells to be updated as “region”. Since the time to update the boundary cells depends much on the face direction of the boundary layer, $\tau_{boundary}^{GPU}$ and $\tau_{boundary}^{CPU}$ are further subdivided in

$$\tau_{boundary}^{GPU} = \max(\tau_{boundary}^{x,GPU}, \tau_{boundary}^{y,GPU}, \tau_{boundary}^{z,GPU}) \tag{15.2a}$$

and

$$\tau_{boundary}^{CPU} = 2(\tau_{boundary}^{x,CPU} + \tau_{boundary}^{y,CPU} + \tau_{boundary}^{z,CPU}). \tag{15.2b}$$

The superscripts x, y, and z indicate the face direction of the corresponding boundary layer; the factor of 2 in equation (15.2b) originates from the negative and positive communication directions, e.g. left and right for x-direction. On the GPU, all kernel executions to update the boundaries are performed simultaneously to increase the degree of parallelism, so the max() function is used in equation (15.2a). On the CPU, all faces of boundary cells are processed after each other because there is already a sufficient degree of parallelism within a single kernel execution, so the particular execution times sum up in equation (15.2b).

The term t_{comm} in performance model (15.1) represents the total communication times of an MPI process including the communication and copy times in the MPI process itself and the communication times to other MPI processes. In weak scaling scenarios, t_{comm} is constant because the subdomain size does not depend on the degree of parallelism leading to constant, per MPI rank execution times and linear scalability. The communication time is further broken down in

$$t_{comm} = t_{comm}^{x} + t_{comm}^{y} + t_{comm}^{z} \tag{15.3}$$

to distinguish the communication times in specific directions. Due to the subdivision of every subdomain as described in section 14.1, the particular, direction-dependent communication times $t_{comm}^{\text{direction}}$ are assembled by

$$t_{comm}^{x} = 2(\underbrace{t_{GPU2buf}^{x} + t_{CPU2buf}^{x}}_{\text{gather boundary data}} + t_{MPI} + \underbrace{t_{buf2GPU}^{x} + t_{buf2CPU}^{x}}_{\text{update boundaries}}) \tag{15.4a}$$

$$t_{comm}^{y} = t_{GPU2buf}^{y} + t_{CPU2buf}^{y} + 2t_{MPI} + t_{buf2GPU}^{y} + t_{buf2CPU}^{y} + t_{GPU2CPU}^{y} + t_{CPU2GPU}^{y} \tag{15.4b}$$

$$t_{comm}^{z} = 2(\underbrace{t_{GPU2buf}^{z} + t_{CPU2buf}^{z}}_{\text{gather boundary data}} + t_{MPI} + \underbrace{t_{buf2GPU}^{z} + t_{buf2CPU}^{z}}_{\text{update boundaries}}) \tag{15.4c}$$

with functions $t_{\text{link}}^{\text{direction}} : \mathbb{N}^3 \to \mathbb{R}^+$ denoting times to transfer a three-dimensional cuboid of data via the connection “link” in a specific “direction”. Copy operations are performed from GPU and CPU memory, respectively, to the buffer used by MPI to exchange probability densities of boundary cells and vice versa. Accordingly, “link” is “GPU2buf”, “buf2GPU”, “CPU2buf”, and “buf2CPU” corresponding to the same identifiers in figure 14.4. The term $t_{\text{GPU2CPU}}^{y} + t_{\text{CPU2GPU}}^{y}$ in equation (15.4b) expresses the time to directly exchange boundary data between the GPU- and the CPU-part of the subdomain as in figure 14.4. Communication between dedicated MPI processes is indicated by the subscript "MPI". Analogously to equations (15.2a) and (15.2b), the "direction" is one of the three spatial dimensions x, y, and z. Equations (15.4a)–(15.4c) base on the assumption that communication times $t_{\text{link}}^{\text{direction}}$ cannot be overlapped in any way. Depending on two MPI processes that are running on the same or different cluster nodes, t_{MPI} can vary for the same three-dimensional cuboid of data because data can either be copied inside the node or transferred via the network. This effect occurs on GPU clusters with more than one GPU per node such as JuDGE, Hydra, and TSUBAME2.5. The proposed performance model does not consider varying t_{MPI} for the same cuboid size.

For now, the performance model is very fine-grained. Yet it can be simplified because some terms behave similarly. For example,

$$\tau_{inner}^{\text{computing device}} \approx \tau_{boundary}^{y,\text{computing device}} \approx \tau_{boundary}^{z,\text{computing device}} \tag{15.5}$$

because as long as computations can benefit from coalesced memory access and caching, respectively, the performance is basically the same for boundary and inner cells if the same number of lattice cells is handled. This does not hold for left and right boundary cells of a subdomain, i.e. the faces in x-direction, hence, showing significantly worse performance and justifying a direction-dependent modeling. When it comes to communication, it sounds reasonable that $t_{GPU2CPU}^{\text{direction}} \approx t_{CPU2GPU}^{\text{direction}} \approx t_{GPU2buf}^{\text{direction}} \approx t_{buf2GPU}^{\text{direction}}$ and $t_{CPU2buf}^{\text{direction}} \approx t_{buf2CPU}^{\text{direction}}$. So, there is no difference in runtime depending on the communication direction (from the computing device to the buffer used for MPI communication or vice versa) and the purpose of memory (buffer or actual domain). However, experiments in section 16.3 show that this assumption is wrong, at least on the tested hardware.

The performance model presented so far models the execution time of an MPI process by incorporating the runtimes of particular operations such as computations and various communication. This approach is software-driven since it follows the order of operations of the actual implementation as depicted by figure 14.4. An alternative approach is hardware-driven by modeling the particular hardware components of the utilized cluster such as computing devices, buses, and communication links. Such an approach orientates for example on figure 2.4 and considers the simultaneous utilization of shared resources by dedicated operations. An example of a shared resource is the memory of the host being a bottleneck for a memory-bound application such as the LBM using the D3Q19 discretization scheme. The host memory stores and loads data stemming from the GPU and MPI communication while the CPU performs α- and β-steps of the inner cells which themselves execute host memory operations. Our performance model assumes that such operations are performed simultaneously without any conflicts. This does not hold in reality but this issue has only a minor effect and, thus, does not limit the validity of the performance model.

The following chapter 16 lists numerous benchmark results of the discussed LBM implementation. Its concluding section 16.3 provides real-world values for the functions $\tau_{\text{region}}^{\text{direction, computing device}}$ and $t_{\text{link}}^{\text{direction}}$ enabling a performance estimation of large-scale runs. These estimations are compared to actually measured results to validate the performance model.

16. Results

Results and corresponding discussions of profilings and benchmarks are given in this chapter to validate the scalability of the combined optimization and parallelization techniques presented in chapter 14 on large-scale heterogeneous systems. This includes performance measurements of kernels on particular computing devices, analysis of the heterogeneous computation of a single subdomain, and scalability tests on three GPU clusters. Furthermore, the validity of the performance model introduced in the previous chapter 15 is testified experimentally.

Performance is measured in giga (10^9) lattice updates per second (GLUPS). For all test runs, the lid-driven cavity benchmark scenario [39, 86] is applied, cf. figure 16.1. The lid is attached on the top of the domain and drives from left to right with velocity 1m/s. Domain size in x-direction is always set to 1m. If the test domain is not a cube, its length in y- and z-direction is adapted accordingly to guarantee cubic lattice cells. To ensure a stable simulation, the timestep size dt is chosen $10\times$ smaller than required to satisfy the Courant-Friedrichs-Lewy (CFL) condition [63]. Hence, since dimensionless units are used within the simulation, the velocity is 0.1. Relaxation time $\tau = 0.6152 \in (0.5, 2)$, thus, the simulation is stable. All test runs take 1024 timesteps, thus, each 512 α- and

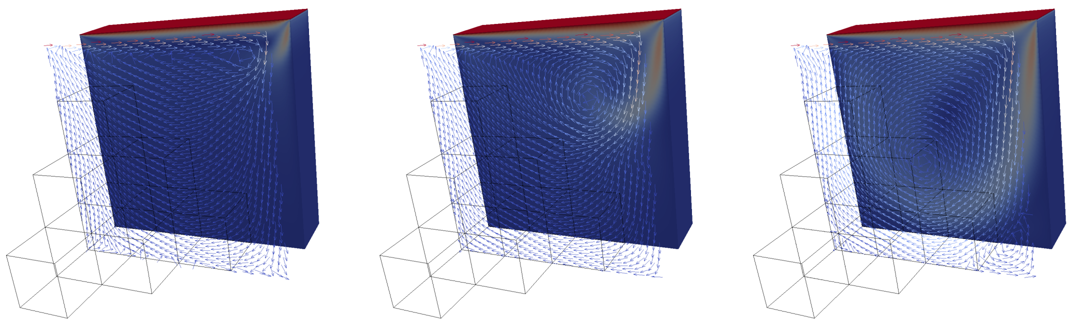

(a) 3840 timesteps $\widehat{=}$ 1s (b) 11,520 timesteps $\widehat{=}$ 3s (c) 46,080 timesteps $\widehat{=}$ 12s

Figure 16.1.: Visualization of a lid-driven cavity scenario with Reynolds number 1000 after 3840, 11,520, and 46,080 timesteps of length $dt = \frac{1}{3840}\text{s} \approx 0.260{,}417 \cdot 10^{-3}\text{s}$. The size of the cubic domain is 384^3 lattice cells, its volume is 1m^3. The domain is partitioned in 4^3 equally-sized subdomains and assigned to the same number of MPI processes. The upper 90% of every subdomain are processed by the GPU, the lower 10% by the CPU. Colors indicate velocity magnitude (red: high velocity, blue: low velocity) with lid velocity $1\frac{m}{s}$. Arrows point in velocity direction.

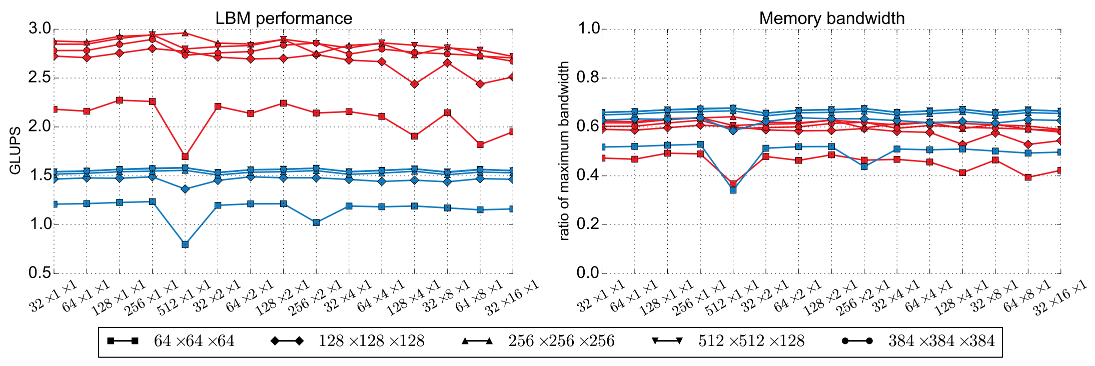


Figure 16.2.: Performance in GLUPS (left) and share of utilized memory bandwidth in peak memory bandwidth (right) of a Tesla P100. Values are determined in dependence of parallel setup for different domain sizes depicted by different lines. Single precision performance is colored in red, double precision performance in blue.

β-steps are executed. Single subdomain tests are carried out on one node of Piz Daint, multiple subdomains tests on the GPU clusters Hydra, TSUBAME2.5, and Piz Daint. Properties of the clusters and utilized software are listed in table 2.5. For these GPU clusters, the number of CPU cores per node is a multiple of the number of GPUs per core. Hence, each thread can be pinned on a dedicated CPU core and no cores idle. To improve the readability, the term “GPU/CPU ratio” is used instead of “ratio of the GPU-part of the domain to the CPU-part of the domain”.

The remainder of this chapter is structured as follows: First, the lower levels of parallelism are evaluated in section 16.1 profiling and benchmarking the GPU and CPU kernels on their corresponding computing devices, thus, on homogeneous architectures. Following, the performance on heterogeneous systems consisting of GPUs and CPUs is tested in section 16.2, starting with single subdomain scenarios before coming to large-scale setups consisting of multiple subdomains. Finally, the validity of our performance model is demonstrated for two weak and two strong scaling setups in section 16.3.

16.1. Characteristics of kernels

To evaluate the performance of the LBM kernels on the particular computing devices, four cubic domains of different size ranging from 64^3 to 384^3 and one cuboid domain consisting of $512 \times 512 \times 128$ lattice cells are benchmarked. Experiments are carried out in single and double precision. The largest 384^3 domain requires approximately 4.22GByte and 8.23GByte of memory when using `float` and `double`, respectively, the latter mentioned domain 2.50GByte and 4.86GByte, respectively. Kernels of the α- and β-step perform the same amount of computations and (coalesced or cached) memory accesses, thus, their performance is similar in terms of GLUPS. So it is not distinguished between

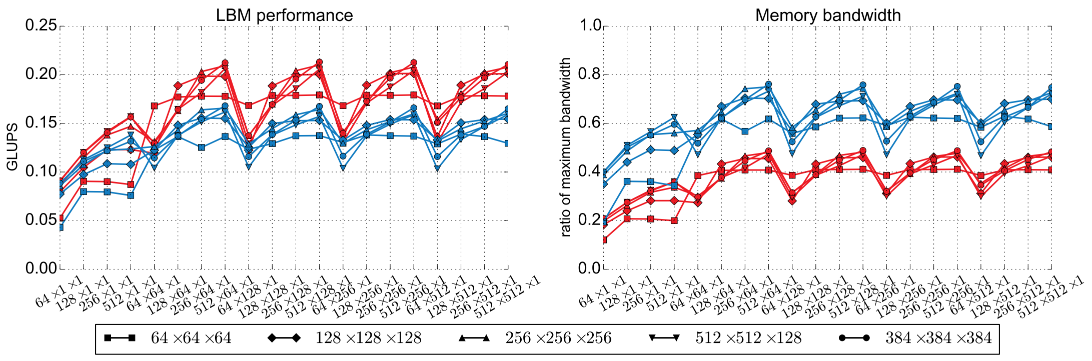


Figure 16.3.: Performance in GLUPS (left) and share of utilized memory bandwidth in peak memory bandwidth (right) of a Xeon E5-2690v3. Values are determined in dependence of block size used to increase cache efficiency. Color coding, line captions, and markers are equal to figure 16.2.

the performance of the two kernels. Since the LBM using the D3Q19 discretization scheme is memory-bound, plots depicting the percentages of achieved memory bandwidth performance are provided. Only computations are carried out, no copy operations or communication occurs. Measurements are just performed on one GPU and CPU architecture because the more relevant result is the behavior on heterogeneous systems presented in section 16.2. Further results of the kernels for different GPUs are given in [209, 20].

16.1.1. Results of the GPU kernels

Results of the GPU kernels carried out on a Tesla P100 are plotted in figure 16.2. Performance values depend on the parallel setup denoted by "threads per block in x-direction" × "threads per block in y-direction" × "threads per block in z-direction" assigned to the x-axes. Problem size is depicted by different lines. Single precision results are colored in red, double precision results in blue. The peak memory bandwidth of the Tesla P100 is 719.87GByte/s (cf. table 2.1).

Comparing the same problem size, the single precision version shows between 57.8% (domain size 64^3, parallel setup $64 \times 8 \times 1$) and 112.6% (domain size 64^3, parallel setup $512 \times 1 \times 1$) higher performance than the double precision version while the latter one has a slightly higher memory bandwidth utilization. In general, larger domains lead to higher GLUPS numbers but performance only slightly grows for domains bigger than 128^3. For such domains, there are enough lattice cells to invoke enough threads for optimal latency hiding. Performance is only slightly influenced by the parallel setup, the only exception occurs when using $512 \times 1 \times 1$ threads per block for the smallest domain. Since every lattice cell is assigned to a dedicated thread, the total number of warps is independant from the parallel setup explaining its low impact. Highest achieved single precision performance is 2.96GLUPS for the 256^3 domain when applying a parallel setup of $512 \times 1 \times 1$ threads per block reaching 64.2% of the peak memory bandwidth. With the same parallel setup, 1.58GLUPS are accomplished for the non-cubic domain with double precision leading to a memory bandwidth utilization of 67.7%. Due to the complexity of the kernels for α- and β-step and integer overheads for index computations, especially to check the boundary conditions, such values of memory bandwidth utilization are expectable and satisfying. Since we mainly focus on scalability on large-scale systems, no further effort is spent to improve the single-GPU performance.

16.1.2. Results of the CPU kernels

Results of the CPU kernels carried out on a Xeon E5-2690v3 (all 12 cores utilized, each executing one thread) are plotted in figure 16.3. Performance values depend on the block size used to increase cache efficiency denoted by "number of cells in x-direction" $\times$ "number of cells in y-direction" $\times$ "number of cells in z-direction" and assigned to the x-axes. Different problem sizes are depicted by different lines. Single precision results are colored in red, double precision results in blue. The peak memory bandwidth of the Xeon E5-2690v3 is 68GByte/s[1].

The difference between single and double precision performance ranges from 0.4% (domain size 256^3, block size $64 \times 1 \times 1$) to 42.1% (domain size 64^3, block size $64 \times 128 \times 1$) and, thus, is much smaller than for the GPU kernels. Problem size influences the performance but a smaller or larger problem does not necessarily lead to lower or higher performance. However, the applied block size has a major impact on performance: The larger the size of the block in x-direction, the higher the GLUPS number, pointed out by the drops in performance if number of cells in x-direction of the block size is only 64. Since probability densities are only used once per α- or β-step, the only performance improvement due to caching stems from data already moved to the cache as part of a cache line previously loaded. Due to the alignment of the probability densities in memory with fastest running index `i` in x-direction, large block sizes in x-direction are beneficial. This effect becomes clearer for larger problem sizes with the smallest impact when using the 64^3 lattice cells domain. The single precision version has a much smaller memory bandwidth utilization than the double precision version reaching at maximum 48.9% (domain size 384^3, block size $512 \times 128 \times 1$) of the peak bandwidth resulting in 0.21GLUPS. In comparison, the best double precision configuration utilizes 74.1% (domain size 384^3, block size $512 \times 64 \times 1$) of the peak bandwidth leading to 0.17GLUPS being a solid performance value. Further improvement could be achieved by considering the assignment of CPU cores to memory channels but this optimization approach is omitted due to the main focus on scalability on large-scale systems.

The best performing parallel setup for the GPU kernels when running a $512 \times 512 \times 128$ lattice cells domain is $512 \times 1 \times 1$ threads per block. Accordingly, the best performing block size for the CPU kernels for the same domain size is $512 \times 64 \times 1$. Both configurations hold for double precision. These values are used throughout the remainder of this chapter.

[1]`https://ark.intel.com/products/81713/`

16.2. Benchmark results for heterogeneous systems

After evaluating the LBM kernels on the particular computing devices, performance is now benchmarked on heterogeneous systems consisting of GPUs and CPUs. First, the interplay between one GPU and one CPU is analyzed in subsection 16.2.1. Subsequently, after providing more background on the setup of the large-scale tests on heterogeneous systems in 16.2.2, weak and strong scaling behavior are discussed in subsections 16.2.3 and 16.2.4. Those parallel setups for the GPU (see figure 16.2) and block sizes for the CPU (see figure 16.3) are used which lead to best performance.

16.2.1. Single subdomain results

To determine potential performance improvement by simultaneously using a GPU and a CPU instead of a computing device of only one type, measurements with a varying GPU/CPU ratio are conducted. Once again, different domain sizes ranging from 64^3 to 384^3 and the cuboid domain with $512 \times 512 \times 128$ lattice cells are used. For all tests, one GPU and one entire CPU handle the domain, thus, all CPU cores are utilized. Since only one domain with one GPU- and one CPU-part is handled, no MPI communication and, hence, no copy operations to the communication buffers is necessary.

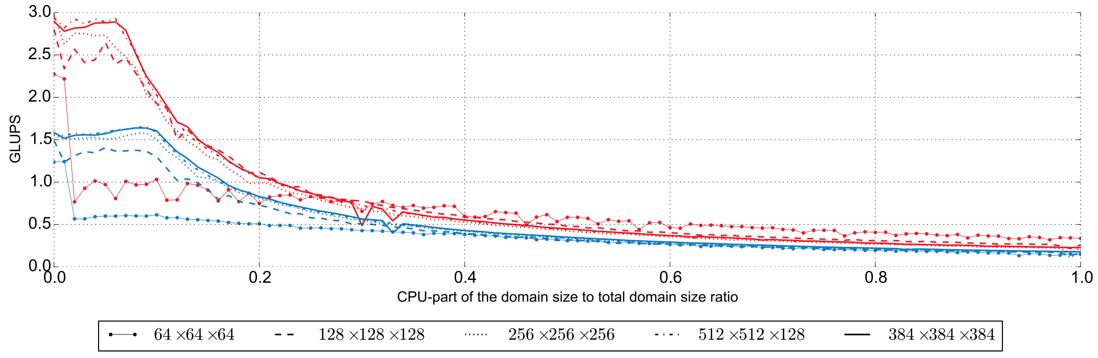


Figure 16.4.: Performance in GLUPS of a heterogeneous system in dependence of the size of the CPU-part of the domain. Size is given as share of the CPU-part in the total domain size ranging from 0 (100%/0% GPU/CPU ratio) to 1 (0%/100% ratio). The GPU-part is processed by a Tesla P100, the CPU-part by a Xeon E5-2690v3. Different lines depict different domain sizes. Single and double precision are colored in red and blue, respectively.

Figure 16.4 depicts benchmark results for all five domain sizes on one node of Piz Daint carried out in single (red lines) and double precision (blue lines). For single precision, it is hard to achieve any improvements with a heterogeneous setup. Actually, only for the large domains with $512 \times 512 \times 128$ and 384^3 lattice cells, similar results like for the GPU-only homogeneous version are accomplishable. Applying a GPU/CPU ratio of 94%/6% leads to 99.52% ($512 \times 512 \times 128$ lattice cells) and 99.83% (384^3 lattice cells) of the homogeneous performance. Here, overheads such as copy operations between the GPU- and CPU-part of the domain equalize the additional computational potential of the CPU. Single precision operations are carried out faster than double precision operations, thus, constant communication overheads such as copy invocation times have a bigger influence in single precision scenarios. For double precision, an improvement of 1.59%, 4.74%, and 3.49% can be realized for the 256^3, $512 \times 512 \times 128$, and 384^3 domain, respectively, when applying the optimal GPU/CPU ratio of 91%/9%. This is still worse than the theoretical improvement of 9.45% obtained by the ratio of the accumulated peak bandwidth of GPU and CPU to the sole GPU's peak bandwidth. Once again, communication overheads consume the theoretical maximum improvement but an acceleration is clearly observable. Small domains do not benefit from a heterogeneous setup at all: As soon as a CPU-part of the domain is involved, performance drops on the level of the CPU-only homogeneous version. Smaller domains require less runtime to be processed, thus, constant communication overheads become even more dominant.

cluster	GPU	CPU	ratio	imp. GPU	theo.	imp. CPU
Hydra	K20x	E5-2680v2	80%/20%	16.22%	23.91%	483.74%
TSUBAME2.5		X5670	95%/5%	1.17%	12.82%	1437.62%
Piz Daint	P100	E5-2690v3	91%/9%	4.74%	9.45%	962.85%

Table 16.1.: Performance improvement by using a heterogeneous system instead of a homogeneous one. Experiments are carried out for a $512 \times 512 \times 128$ domain using double precision. Column "ratio" lists the best performing GPU/CPU ratio. All GPUs are NVIDIA Tesla GPUs, all CPUs are Intel Xeon CPUs. Imp. stands for improvement, theo. for theoretical. Values in the columns "imp. GPU" and "imp. CPU" list the measured relative performance improvement of the heterogeneous setup in comparison to the particular computing devices. The theoretical improvement in comparison to the GPU-only configuration is given in column "theo.".

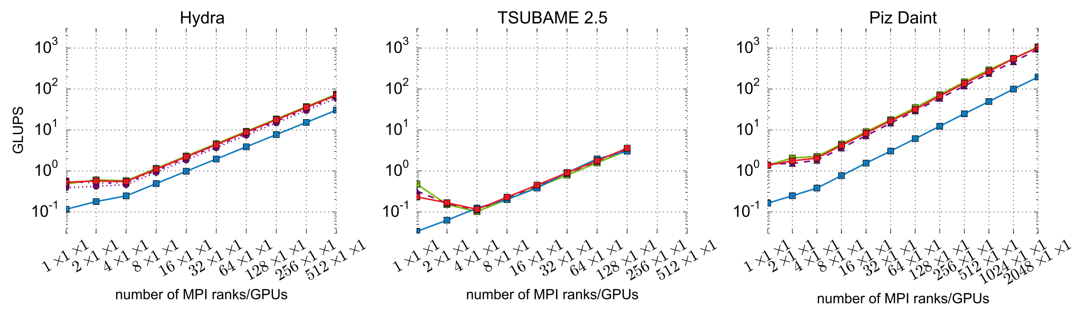
Improvements measured on nodes of all GPU clusters utilized for the large-scale benchmarks are summed up in table 16.1. These results are limited to only one domain of size $512 \times 512 \times 128$ and double precision, also being the setup applied for the large-scale runs in subsections 16.2.3 and 16.2.4. Such a domain fits entirely in the memory of every utilized GPU and every cluster node is equipped with enough host memory to store all subdomains assigned to the MPI processes running on it. A non-cubic test domain is chosen because its face in z-direction is of different size than in x- or y-direction leading to different communication times in all directions. Similar to the more detailed evaluation on Piz Daint illustrated in figure 16.4, one GPU and one entire CPU handle the domain, thus, all CPU cores are utilized. Table 16.1 contains the best performing GPU/CPU ratio, the theoretical improvement basing on the accumulated peak bandwidth of GPU and CPU, and the achieved improvement relative to a GPU- and CPU-only configuration. While on TSUBAME2.5 the achieved performance gain of the heterogeneous version in comparison to the GPU-only version is negligible, the performance on Hydra is not only significantly improved but also a relevant portion of the theoretical improvement can be reached. The behavior on a TSUBAME2.5 node can be explained by its architecture containing three GPUs and two CPUs. This leads to high GPU performance per node and restricts the heterogeneous performance due to a complex PCIexpress bus [89].

16.2.2. Preparations for multiple subdomains results

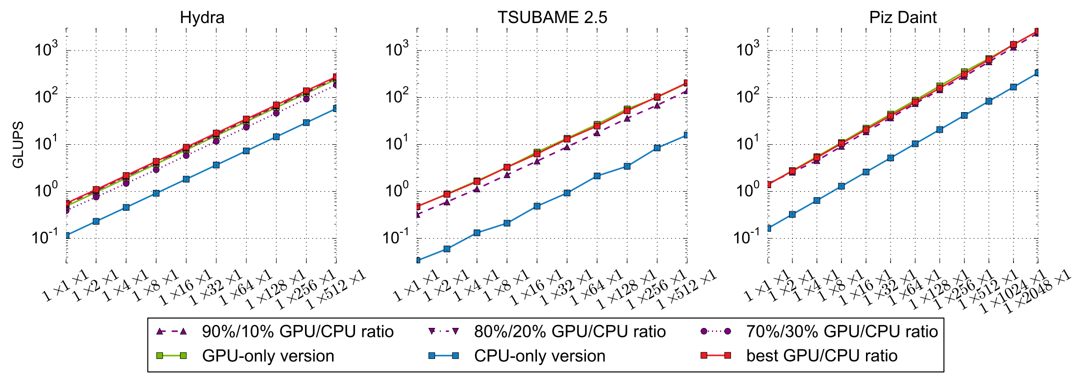
The large-scale benchmarks are carried out in double precision. Values of interest are performance in GLUPS, and from this quantity derived, parallel efficiency, both assigned to the ordinates of figures 16.5–16.12. Since we are interested how performance and parallel efficiency depend on the number of subdomains, these values are written to the abscissas of the figures. The number of subdomains is denoted by "number of subdomains in x-direction" $\times$ "number of subdomains in y-direction" $\times$ "number of subdomains in z-direction". Weak (see subsection 16.2.3 and figures 16.5–16.8) and strong scaling (see subsection 16.2.4 and figures 16.9–16.12) measurements are conducted. Adding subdomains in x-direction in such a way that communication is performed between left and right faces is parallelization in x-direction. Analogously, there is parallelization in y- and z-direction. If subdomains have neighbors in all spatial directions, it is called parallelization in $x/y/z$-direction. In all figures of the large-scale tests, different lines depict different homogeneous and heterogeneous versions: The homogeneous GPU- and CPU-only versions are colored in green and blue, respectively. For every GPU cluster, there are heterogeneous results for a GPU/CPU ratio of 90%/10% and for Hydra, there are also results for ratios 80%/20% and 70%/30%. The heterogeneous results are colored in purple. In addition, the heterogeneous version showing best performance in most cases

system	scenario	parallelization in ...-direction			
		x	y	z	xyz
Hydra	GPU-only	30.2%	99.6%	76.2%	16.9%
	CPU-only	51.5% (2^9)	98.3% (2^9)	89.7% (2^8)	42.9% (2^9)
	hetero.	26.3%	97.3%	60.2%	15.4%
TSUBAME2.5	GPU-only	5.1%	83.0%	64.8%	4.8%
	CPU-only	71.0% (2^7)	92.5% (2^9)	98.8% (2^9)	41.9% (2^9)
	hetero.	12.1%	84.4%	62.9%	4.8%
Piz Daint	GPU-only	37.6%	91.4%	54.8%	17.7%
	CPU-only	58.7% (2^{11})	99.9% (2^{11})	90.8% (2^{11})	49.5% (2^{11})
	hetero.	36.6%	90.6%	53.5%	17.6%

Table 16.2.: Weak scaling parallel efficiencies on three different GPU clusters for the largest performed runs. Number of utilized processes is given in parenthesis. Hetero. stands for heterogeneous. The best performing GPU/CPU ratio is taken for the heterogeneous scenario. Results for parallelization in $x/y/z$-direction as well as in all directions are listed. Detailed results are plotted in figures 16.7 and 16.8.



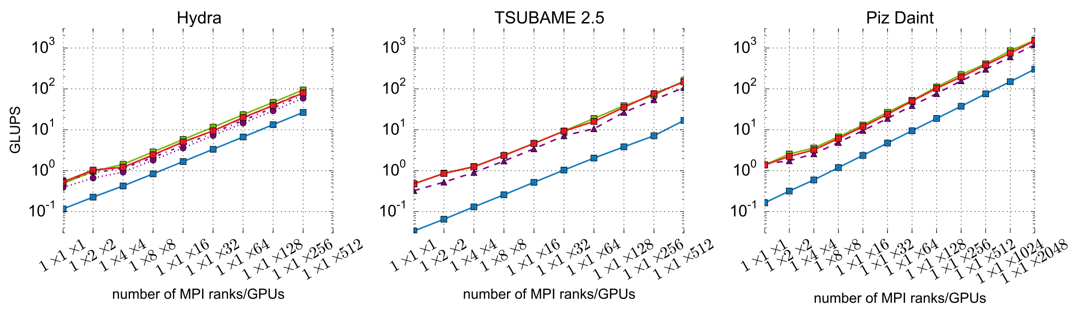
(a) Parallelization in x-direction



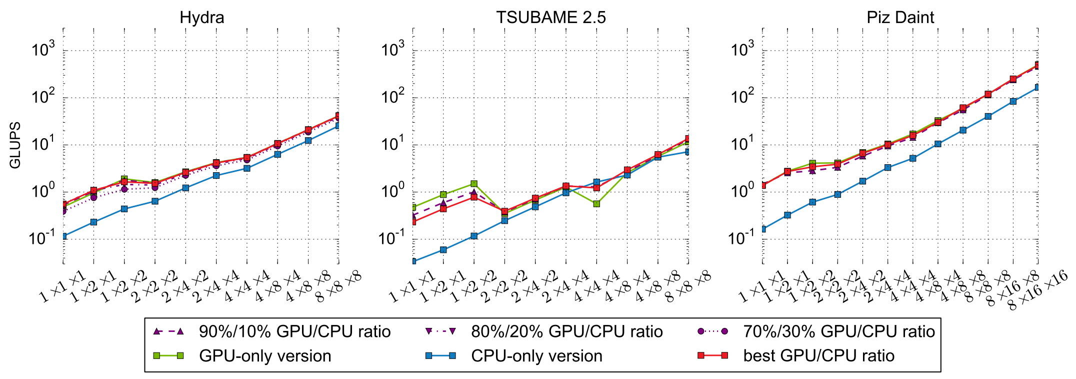
(b) Parallelization in y-direction

Figure 16.5.: Weak scaling performance in GLUPS on three different GPU clusters in dependence of number of subdomains. The number of subdomains is denoted by "number of subdomains in x-direction" $\times$ "number of subdomains in y-direction" $\times$ "number of subdomains in z-direction". One subdomain consists of $512 \times 512 \times 128$ lattice cells. Different lines depict different GPU/CPU ratios: The GPU-only homogeneous version is colored in green, the CPU-only homogeneous version in blue and different heterogeneous versions in purple. The best performing heterogeneous version is colored in red. Subfigure 16.5(a) plots results for parallelization in x-, subfigure 16.5(b) in y-direction. Axes are logarithmically scaled.

is colored in red. The GPU/CPU ratio of the best performing heterogeneous version varies for different parallelization strategies and on different GPU clusters. Although TSUBAME2.5 is equipped with three GPUs per node, only two of them are utilized while the remaining GPU idles. Hence, only two MPI processes are assigned to each TSUBAME2.5 node. Our experiments show that running three MPI processes per node leads to worse performance than running just two of them. The complex PCIexpress bus of a TSUBAME2.5 node [89] restricts performance when using all three GPUs and the CPU memory bandwidth has to be shared among three MPI processes.



(a) Parallelization in z-direction

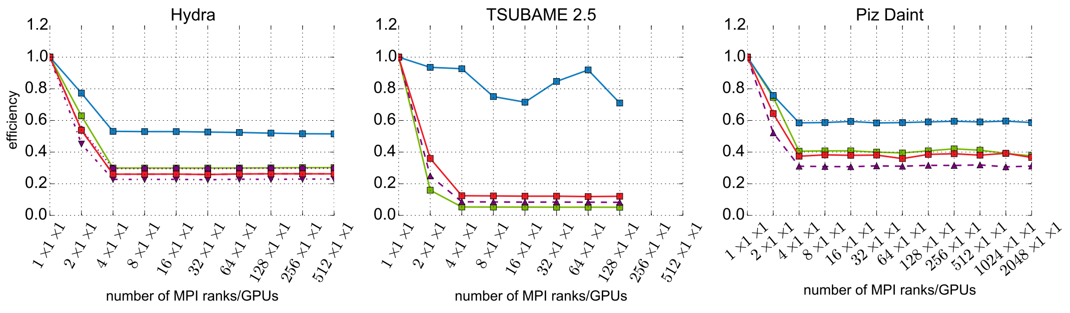


(b) Parallelization in $x/y/z$-direction

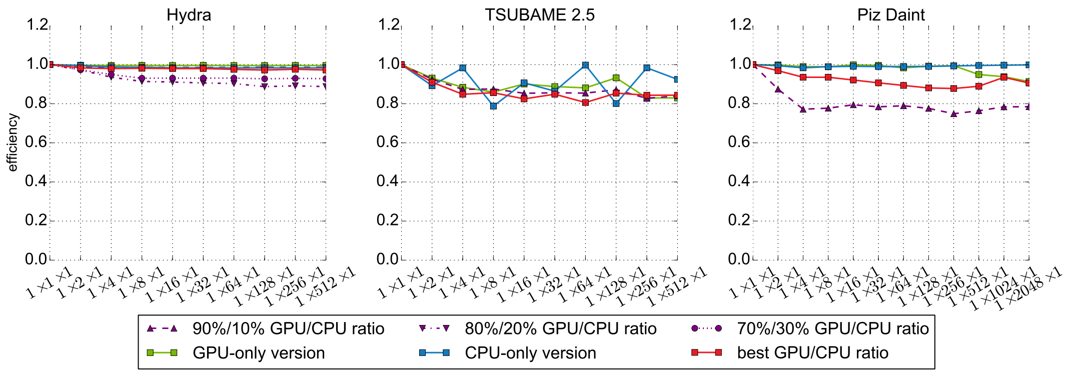
Figure 16.6.: Weak scaling performance in GLUPS on three different GPU clusters in dependence of number of subdomains. Subfigure 16.6(a) plots results for parallelization in z-direction, subfigure 16.6(b) in $x/y/z$-direction. Axes assignment and scaling, color coding, line captions, and subdomain size are equal to figure 16.5.

16.2.3. Weak scaling results of multiple subdomains

For the weak scaling benchmarks, a subdomain size of $512 \times 512 \times 128$ lattice cells is used because such a subdomain has different communication times in all directions. Figures 16.5 and 16.6 depict results for different parallelization strategies. The corresponding parallel efficiencies using the one subdomain setup as reference are plotted in figures 16.7 and 16.8, and are summed up in table 16.2 for clarity. On all three clusters, parallelization in x-direction does not lead to increased performance for small MPI process numbers. As mentioned in section 14.3, communication performance in x-direction is much worse than in other directions limiting performance when parallelizing in x-direction. This effect can also be observed for combined parallelization in $x/y/z$-direction in subfigures 16.6(b) and 16.8(b): As soon as additional subdomains are added in x-direction, increase in performance is worse than adding subdomains in y-



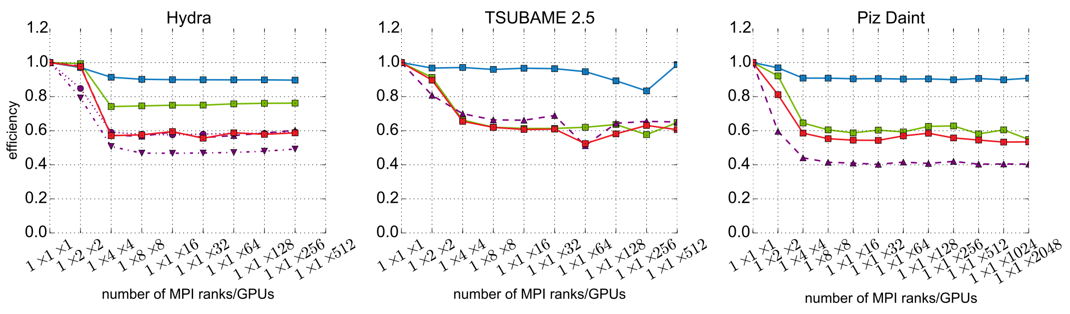
(a) Parallelization in x-direction



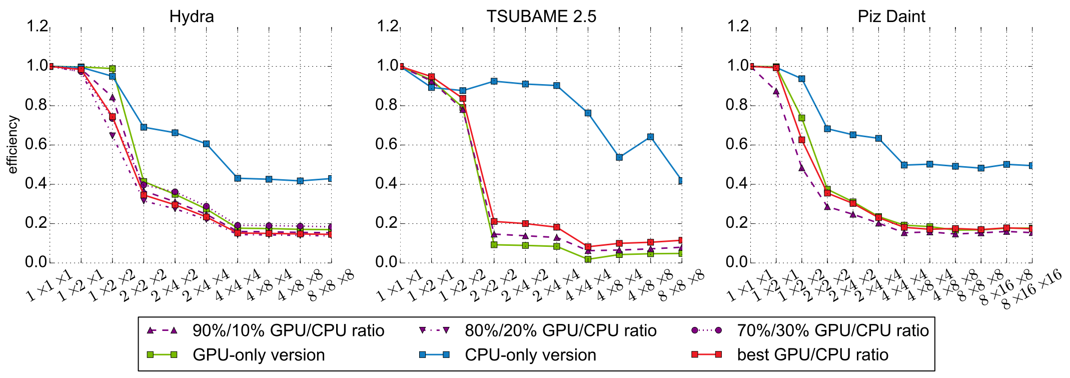
(b) Parallelization in y-direction

Figure 16.7.: Weak scaling parallel efficiency on three different GPU clusters in dependence of number of subdomains basing on values of figure 16.5. Parallel efficiency with one MPI process as baseline is plotted along ordinates. Subfigure 16.7(a) plots results for parallelization in x-, subfigure 16.7(b) in y-direction. The abscissa assignment, axes scaling, color coding, line captions, and subdomain size are equal to figure 16.5.

or z-direction. Pure parallelization in y-direction shows very nice scalability achieving parallel efficiencies of 97.27% on Hydra (512 MPI processes, 85%/15% GPU/CPU ratio), 84.40% on TSUBAME2.5 (512 MPI processes, 96%/4% GPU/CPU ratio), and 90.60% on Piz Daint (2048 MPI processes, 95%/5% GPU/CPU ratio). This results in highest measured performance of 2604.72GLUPS on Piz Daint utilizing 2048 GPUs and 24,576 CPU cores. Since the communication effort in y-direction (face size $512 \times 1 \times 128 = 2^{16}$ lattice cells) is smaller than in z-direction (face size $512 \times 512 \times 1 = 2^{18}$ lattice cells), parallelization in y-direction performs better than in z-direction. Such nice scaling rates are possible because communication can be entirely hidden by computations.



(a) Parallelization in z-direction



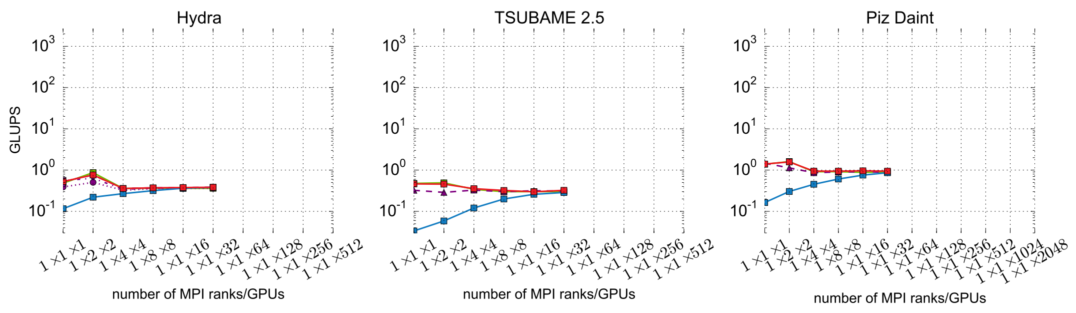
(b) Parallelization in $x/y/z$-direction

Figure 16.8.: Weak scaling parallel efficiency on three different GPU clusters in dependence of number of subdomains basing on values of figure 16.6. Parallel efficiency with one MPI process as baseline is plotted along ordinates. Subfigure 16.8(a) plots results for parallelization in z-direction, subfigure 16.8(b) in $x/y/z$-direction. The abscissa assignment, axes scaling, color coding, line captions, and subdomain size are equal to figure 16.5.

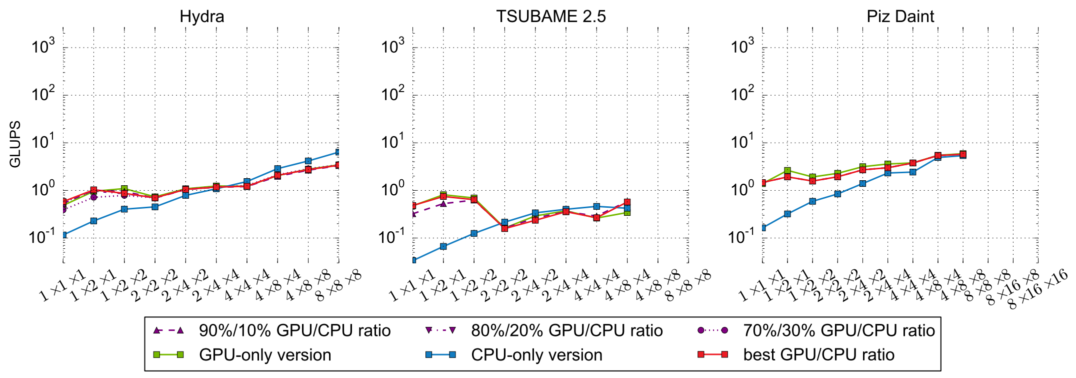
16.2.4. Strong scaling results of multiple subdomains

For the strong scaling benchmarks, two different domain sizes are applied: A smaller $512 \times 512 \times 128$ domain whose results are plotted in figures 16.9 and 16.10 and a larger domain consisting of 512^3 lattice cells whose results are given in figures 16.11 and 16.12. The larger domain is initially subdivided in $4 \times 1 \times 1$ subdomains and an alternative decomposition strategy is applied to increase performance and parallel efficiency.

Using more computational resources for the smaller domain does not lead to higher GLUPS numbers if parallelization occurs in z-direction (see subfigure 16.9(a)). Just the CPU-only version is accelerated by this kind of parallelization but parallel efficiency is still poor as illustrated in subfigure 16.10(a). Parallelizing in $x/y/z$-direction leads to slightly better scaling behavior (see subfigure 16.9(b)), although performance just slowly



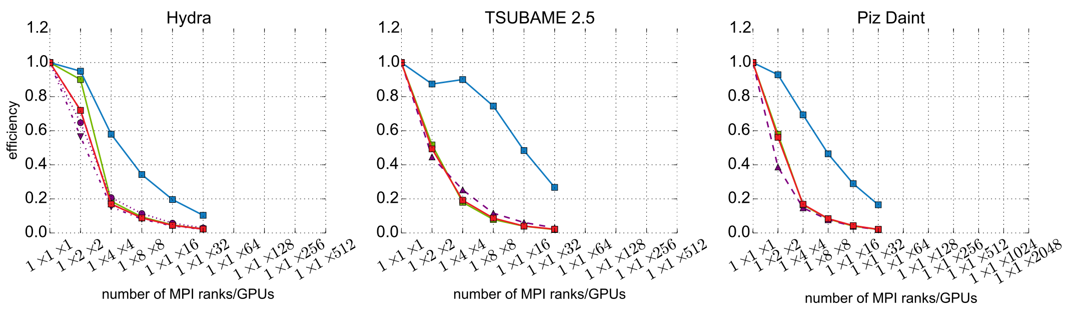
(a) Parallelization in z-direction



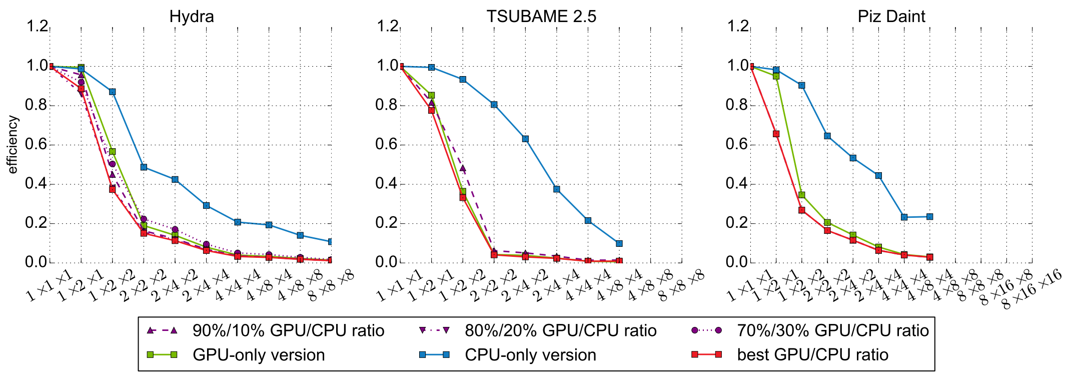
(b) Parallelization in $x/y/z$-direction

Figure 16.9.: Strong scaling performance in GLUPS on three different GPU clusters in dependence of number of subdomains. The number of subdomains is denoted by "number of subdomains in x-direction" × "number of subdomains in y-direction" × "number of subdomains in z-direction". Domain size is $512 \times 512 \times 128$ lattice cells. Different lines depict different GPU/CPU ratios: The GPU-only homogeneous version is colored in green, the CPU-only homogeneous version in blue and different heterogeneous versions in purple. The best performing heterogeneous version is colored in red. Subfigure 16.9(a) plots results for parallelization in z-direction, subfigure 16.9(b) in $x/y/z$-direction. Axes are logarithmically scaled.

grows for higher process numbers. Again, just the CPU-only version shows a significant performance improvement, especially on Hydra. This strong scaling scenario is clearly dominated by communication times. For example, if parallelization in z-direction is applied, the size of the faces in z-direction stays constant while the actual subdomains shrink. Subdomains become very thin in z-direction, thus, executions with more than 32 subdomains are not possible. Parallelizing in all spatial directions reduces the amount of data to be transferred, but subdomain faces become so small that communication latencies dominate communication times. Since communication times from and to the



(a) Parallelization in z-direction

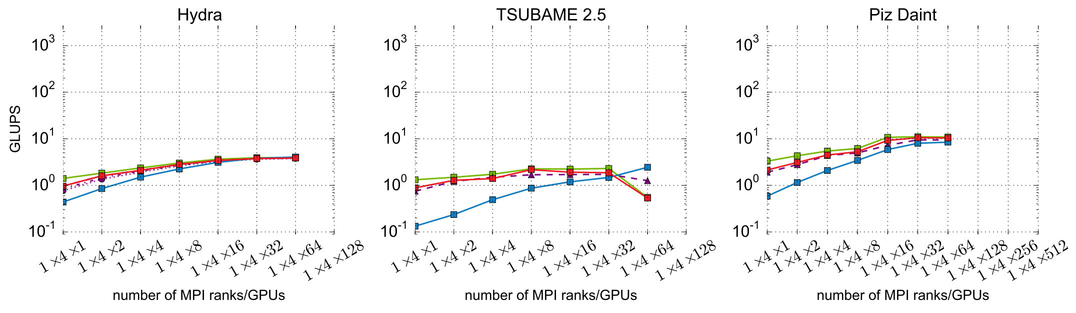


(b) Parallelization in $x/y/z$-direction

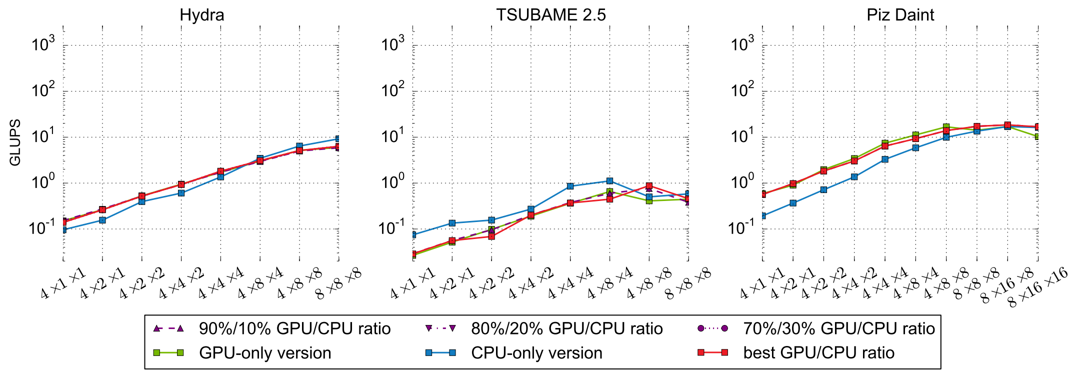
Figure 16.10.: Strong scaling parallel efficiency on three different GPU clusters in dependence of number of subdomains basing on values of figure 16.9. Parallel efficiency with one MPI process as baseline is plotted along ordinates. Subfigure 16.10(a) plots results for parallelization in z-direction, subfigure 16.10(b) in $x/y/z$-direction. The abscissa assignment, axes scaling, color coding, line captions, and domain size are equal to figure 16.9.

GPU are dropped for the CPU-only version, communication latencies are much shorter and the scaling behavior is better than for heterogeneous or GPU-only versions.

For the larger strong scaling scenario, basically the same performance behavior is observed as for the smaller domain if parallelization in z-direction is applied (see subfigures 16.11(a) and 16.12(a)): Except for the CPU-only version, performance stagnates or even decreases (using 256 processes on TSUBAME2.5) when more computational resources are utilized. However, the strong scaling behavior can be significantly enhanced by a smart decomposition strategy in all three spatial directions: First, subdividing the domain in x-direction should be avoided. Second, an alternating subdivision of the domain in y- and z-direction leads to subdomains having a more cube- than layer-shape appearance. Applying this strategy enables parallel efficiencies of 31.09% (GPU-only), 36.16% (75%/25% GPU/CPU ratio), and 75.16% (CPU-only) on Hydra with 512 subdomains.



(a) Parallelization in z-direction

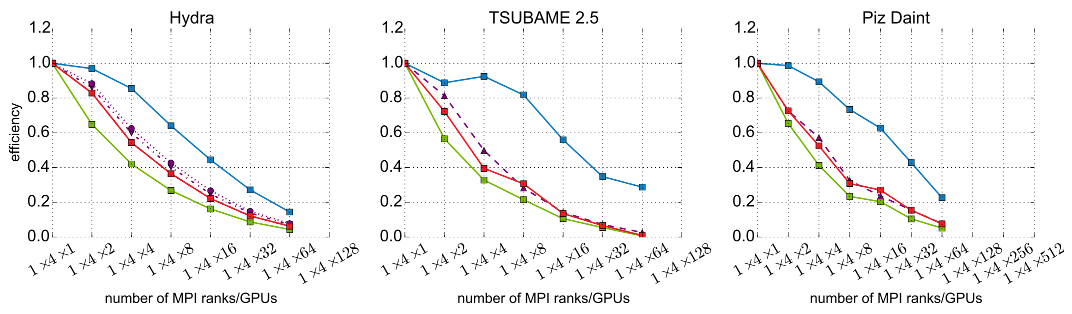


(b) Parallelization in $x/y/z$-direction

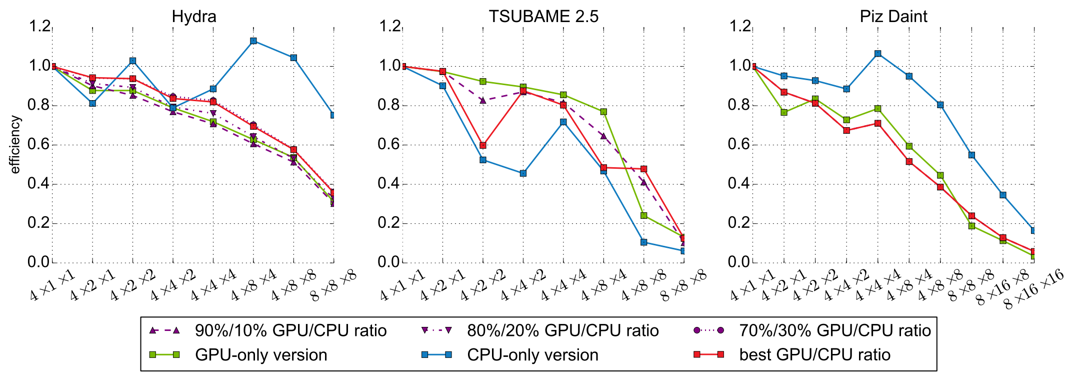
Figure 16.11.: Strong scaling performance in GLUPS on three different GPU clusters in dependence of number of subdomains. The number of subdomains is denoted by "number of subdomains in x-direction" $\times$ "number of subdomains in y-direction" $\times$ "number of subdomains in z-direction". Domain size is 512^3 lattice cells. Subfigure 16.11(a) plots results for parallelization in z-direction, subfigure 16.11(b) in $x/y/z$-direction. Axes scaling, color coding and line captions are equal to figure 16.9.

On a first look, these values look poor but considering the small subdomain sizes for high subdomain numbers and the eminent communication effort, such parallel efficiencies are notable in communication-dominated scenarios. Again, best scalability is accomplished for the CPU-only version because no communication times with the GPU have to be incorporated. The achievable parallel efficiencies on TSUBAME2.5 and Piz Daint are not as big as on Hydra but still much better than for the small subdomain scenario using the straight-forward parallelization strategy. Anyway, for very small subdomain sizes, even the smart decomposition strategy does not guarantee increasing performance as indicated on Piz Daint with 2048 processes (see subfigure 16.11(b)). In this case, a subdomain consists of just $64 \times 32 \times 32$ lattice cells.

Concluding this section, the application of heterogeneous computing leads to better



(a) Parallelization in z-direction



(b) Parallelization in $x/y/z$-direction

Figure 16.12.: Strong scaling parallel efficiency on three different GPU clusters in dependence of number of subdomains basing on values of figure 16.11. Parallel efficiency with four MPI processes as baseline is plotted along ordinates. Subfigure 16.12(a) plots results for parallelization in z-direction, subfigure 16.12(b) in $x/y/z$-direction. The abscissa assignment corresponds to the assignment in figure 16.11. Axes scaling, color coding, line captions, and domain size are equal to figure 16.9.

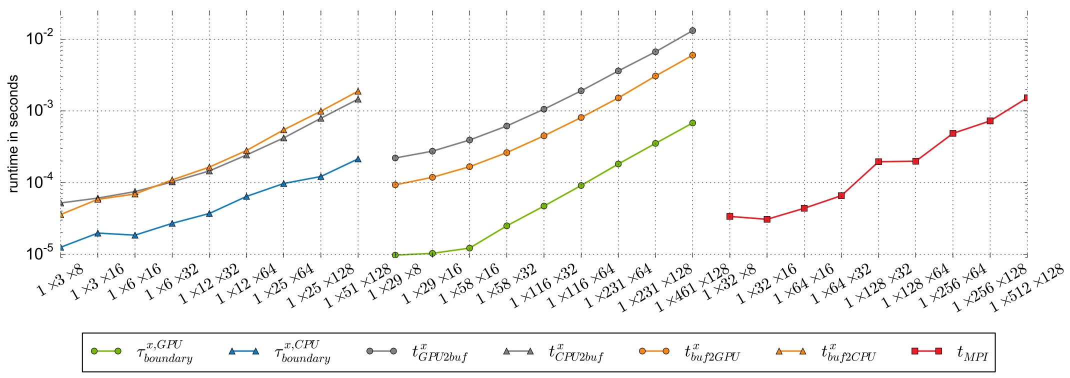
performance for the LBM than limiting operations to computing devices of only one type in most cases. Regarding the single subdomain results, the theoretical performance gain corresponds to the accumulated memory bandwidth of the computing devices in comparison to the memory bandwidth of the particular computing devices because the LBM is memory-bound. Depending on the node setup, the actually achieved performance gain varies: While on a Hydra node, 67.83% of the theoretical improvement is reached, this value is only 9.13% on a node of TSUBAME2.5. Regarding multiple subdomain results, the best performing heterogeneous version (85%/15% GPU/CPU ratio) achieves 12.85% higher GLUPS values than the GPU-only version when using 512 subdomains in weak scaling applying parallelization in y-direction on Hydra (cf. subfigure 16.5(b)). However, this does not hold for all combinations of scenario, parallelization strategy, and GPU cluster. For example on Hydra and TSUBAME2.5, the GPU-only version always achieves higher GLUPS values than the best performing heterogeneous version (90%/10% and 97%/3% GPU/CPU ratio, respectively) for weak scaling applying parallelization in z-direction (cf. subfigure 16.6(a)). There are also strong scaling examples such as for the smaller domain on Piz Daint when applying parallelization in $x/y/z$-direction (cf. subfigure 16.9(a)). In such cases, the overhead to incorporate the CPUs is bigger than the performance gain. Except for parallelization in x-direction, excellent weak scaling can be achieved on all three utilized GPU clusters (cf. table 16.2) because in most cases, communication times can be hidden with computation times of the inner cells. For strong scaling scenarios, communication times and especially communication latencies for large subdomain numbers become dominant. Hence, smart decomposition strategies have to applied to reach satisfying parallel efficiencies.

16.3. Validation of the performance model

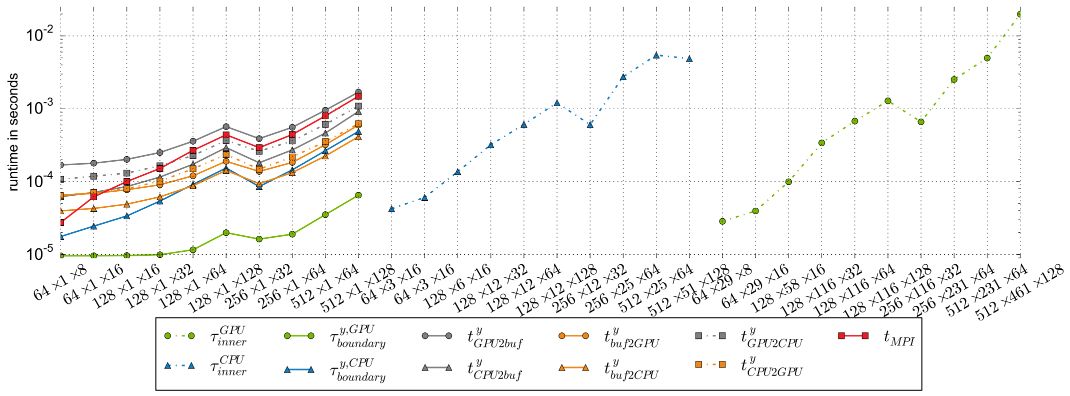
With the measured large-scale results from subsections 16.2.3 and 16.2.4 at hand, it is possible to experimentally validate the performance model from chapter 15. Two weak and two strong scaling scenarios are evaluated on Piz Daint: For the weak scaling scenarios, parallelization in y-direction and in $x/y/z$-direction is applied, thus, measured results are given in subfigures 16.5(b) and 16.6(b). For the strong scaling scenarios, parallelization in $x/y/z$-direction is applied for the cuboid domain consisting of $512 \times 512 \times 128$ lattice cells (cf. subfigure 16.9(b)) and the cubic domain with 512^3 lattice cells (cf. subfigure 16.11(b)). A heterogeneous version with a 90%/10% GPU/CPU ratio is used for all four test scenarios.

The performance model requires experimentally measured values of the functions $\tau_{\text{region}}^{\text{computing device}} : \mathbb{N}^3 \to \mathbb{R}^+$ modeling computation times and $t_{\text{link}}^{\text{direction}} : \mathbb{N}^3 \to \mathbb{R}^+$ modeling communication times for different spatial directions. Figure 16.13 gives all required runtimes for different cuboid sizes denoted by “number of lattice cells in x-direction” $\times$ “number of lattice cells in y-direction” $\times$ “number of lattice cells in z-direction”. Measured values are taken from scenarios using two subdomains giving a minimal parallel example in the corresponding spatial direction running for 1024 timesteps. For each spatial direction, there is a dedicated subfigure. Computing devices of a node of Piz Daint are utilized, hence, a Tesla P100 as GPU and a Xeon E5-2690v3 as CPU. In general, $\tau_{\text{region}}^{\text{computing device}}$ and $t_{\text{link}}^{\text{direction}}$ do not scale linearly because for small cuboid sizes, kernel invocation times and communication latencies become relevant. As mentioned in chapter 15, results in figure 16.13 testify that the assumptions $t_{GPU2CPU}^{\text{direction}} \approx t_{CPU2GPU}^{\text{direction}} \approx t_{GPU2buf}^{\text{direction}} \approx t_{buf2GPU}^{\text{direction}}$ and $t_{CPU2buf}^{\text{direction}} \approx t_{buf2CPU}^{\text{direction}}$ are wrong because the corresponding runtimes differ much, at least on Piz Daint.

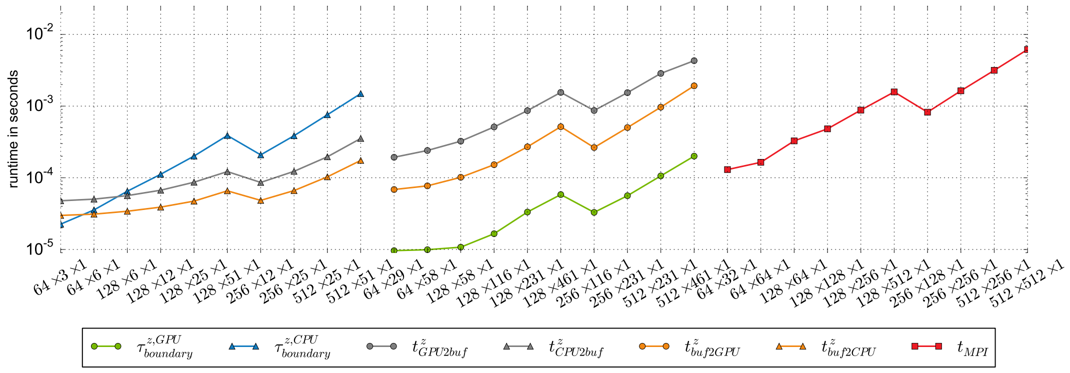
A comparison of measured and predicted performance values for the four evaluated scenarios is illustrated in figure 16.14. Here, for the smaller scenarios using parallelization in $x/y/z$-direction in which subdomains do not have a neighbor in all three spatial directions as assumed by the performance model, the corresponding runtimes are neglected to obtain more accurate predictions. Perfect linear scaling is predicted by the



(a) x-direction

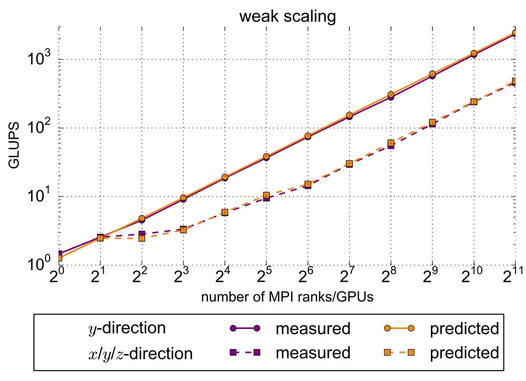


(b) y-direction

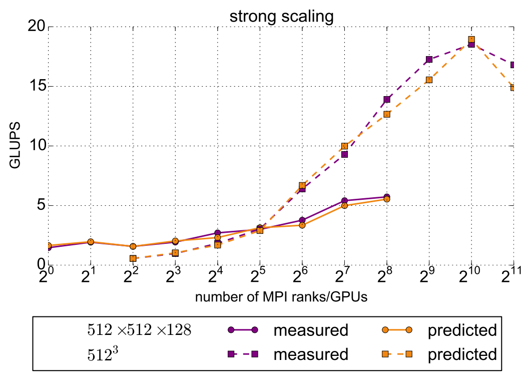


(c) z-direction

Figure 16.13.: Selected runtimes (assigned to ordinates) of $\tau^{\text{computing device}}_{\text{region}}$ and $t^{\text{direction}}_{\text{link}}$ (depicted by different lines) for different three-dimensional cuboids of data (assigned to abscissas). Ordinates are logarithmically scaled. Different subfigures refer to different spatial directions. Piz Daint's computing devices Tesla P100 and Xeon E5-2690v3 are utilized.



(a) weak scaling



(b) strong scaling

Figure 16.14.: Comparison of predicted (orange lines) and measured (purple lines) performance for two weak (subfigure 16.14(a)) and two strong (subfigure 16.14(b)) scaling scenarios. Performance is given in GLUPS (ordinates) and depends on number of subdomains (abscissas). The GPU/CPU ratio is 90%/10%. In subfigure 16.14(a), solid lines depict parallelization in y-direction and dashed lines in $x/y/z$-direction. In subfigure 16.14(b), solid lines depict results of the cuboid $512 \times 512 \times 128$ domain and dashed lines of the cubic 512^3 domain. All axes except the ordinate of subfigure 16.14(b) are scaled logarithmically.

performance model for the weak scaling scenario using parallelization in y-direction being actually realized by our LBM implementation. The same holds for parallelization in $x/y/z$-direction once parallelization in every spatial direction is realized (≥ 8 subdomains) because from then on, the runtime per subdomain stays constant. Largest deviations between measured and predicted performance are 9.10% (256 subdomains) and 15.44% (4 subdomains), respectively, for the two weak scaling scenarios plotted in subfigure 16.14(a). Regarding the two strong scaling scenarios, the difference between measured and predicted performance are 17.09% (16 subdomains) and 12.82% (2048 subdomains), respectively, as shown in subfigure 16.14(b). Besides quantitative predictions, the performance model is also a qualitative indicator as for the drop in performance of the large strong scaling scenario when using 2048 subdomains.

This comparison shows that the performance model reliably predicts the performance of our LBM implementation, for weak scaling as well as strong scaling scenarios. Limitations such as the simultaneous usage of shared resources only have a minor influence on its quality.

Concluding part IV

We briefly present the LBM with the D3Q19 discretization scheme and aspects for an efficient serial implementation in chapter 13. Various techniques for a parallelization of the LBM on large-scale heterogeneous systems are presented in chapter 14. As demonstrated in chapter 16, the combination of these techniques enables a scalable implementation of the LBM on GPU clusters. Furthermore, a performance model tailored to the applied techniques is introduced in chapter 15 to predict the performance on different heterogeneous systems.

Our main goal is to achieve good scalability on large-scale heterogeneous systems. Scalability is a measurement of future-proofness since prospective supercomputers mainly achieve higher performance by a larger degree of parallelism, maybe using different types of specialized computing devices. In this part, a collection of best practices is presented, basically applicable to every high performance application for heterogeneous architectures, enabling such scalability. Using the LBM, a holistic reference implementation of these best practices is provided and discussed. Beginning from the bottom, we use dedicated kernels for the particular computing devices basing on work by Schreiber [209, 210]. These kernels are optimized by using the A-A memory layout pattern for the GPU and the CPU enabling data exchange between computing devices without any conversions. Since the LBM is memory-bound, optimizations such as coalesced memory access on the GPU and caching on the CPU permit high-performance kernels. Due to the high theoretical memory bandwidth of GPUs (see row "PMBW (GByte/s)" in tables 2.1 and 2.2), they are an ideal computing device for memory-bound problems. All computing devices perform the same type of work, namely α- and β-steps applying CUDA for the GPU and OpenMP tasks for the CPU. From the top, communication between the computing devices within one node and communication among multiple nodes has to be realized. This task is achieved by mapping each GPU and some CPU cores handling a cuboid subdomain to one MPI process. To gain good scalability of the hybrid implementation, communication times have to be hidden by computation times. Hence, the boundary cells of the GPU- & CPU-parts are processed before the inner cells are handled. While the inner cells are computed, communication can be performed simultaneously. The challenge is the successful and efficient combination of the parallelization models for all levels of parallelism (computing devices, computing devices within one node, and multiple nodes) on the one hand and programming models such as CUDA, OpenMP, and MPI on the other hand. The GPU is one incorporated aspect, but only one besides many others.

For a GPU (Tesla P100), up to 67.7% of the peak memory bandwidth could be reached for a 256^3 lattice cells domain using double precision resulting in 1.58GLUPS. Accordingly, for a CPU (Xeon E5-2690v3), the highest achieved peak memory bandwidth utilization is 74.1% leading to 0.17GLUPS for a 384^3 lattice cells domain using double precision. Combining all computational resources of a node can increase performance by up to 16.22% (one GPU and one CPU of Hydra, GPU/CPU ration 80%/20%) in comparison to a GPU-only version achieving 67.84% of the maximum potential performance improvement. Going one step further and utilizing multiple GPUs and CPUs, up to 2604.72GLUPS are carried out in a weak scaling scenario on Piz Daint by using 2048 subdomains (95%/5% GPU/CPU ratio) parallelized in y-direction with a parallel efficiency of 90.60%. In this scenario, communication times can be completely hidden by computations of inner lattice cells. By using a smart decomposition scheme, a parallel efficiency of 36.2% can be obtained by the best performing heterogeneous version in the large strong scaling scenario on Hydra with 512 subdomains. The CPU-only version with the same setup accomplishes 75.2% parallel efficiency. Using a heterogeneous GPU cluster can accelerate the application by up to 12.85% in comparison to the GPU-only version (weak scaling, Hydra, 512 subdomains, 85%/15% GPU/CPU ratio).

The presented performance model estimates the runtime by determining the execution time to perform all computations and communication of one single subdomain and assumes this runtime to be equal for all subdomains. It considers different computation and communication times depending on the face orientation of the corresponding lattice cells but ignores shared resources such as joint memories and buses. The deviations between measured and predicted performance are always $< 20\%$ for the tested scenarios, in most cases even $< 5\%$ and, thus, quite accurate.

This final part IV does not only deal with single or multiple GPUs but sketches how to use them in a heterogeneous scenario by applying the LBM. The CPUs of the clusters are not degraded to communication tasks but actively contribute to the computations. Numerous communication channels are considered, optimizations on all levels of parallelism are applied, data transfers and computations are executed concurrently, and so, scalability is achieved with thousands of GPUs and ten-thousands of CPU cores.

Part V.

Conclusion

Although GPUs now have been used for more than a decade to enhance scientific computing applications in special and much experience was gained in the context of accelerators in general, GPUs still offer untapped potential usable by applying non-trivial approaches. Following two golden threads—various aspects of differential equations as the central object for modeling and a comprehensive view on GPUs and their ecosystem—new utilization areas and scenarios for GPUs by means of three applications are discussed in this thesis: First, the SBTH algorithm, one possible step of the solution of the Eigenvalue problem, e.g. to interpret phenomena described by differential equations, is implemented on single GPUs by delegating work to BLAS level 1 and level 2 routines in a pipelined way. This application is subject of part II. Afterwards, three levels of parallelism are exploited to generally solve RODEs, differential equations incorporating a stochastic process, by applying four successive algorithmical steps on multiple GPUs. Part III deals with this application. Finally, various optimization and parallelization techniques are applied in part IV to achieve scalability of the LBM, an alternative discretization of differential equations modeling fluid dynamics, on large-scale heterogeneous clusters. It is shown that GPUs are not just interesting for compute-bound but also memory- and latency-bound problems.

We show that one single GPU can provide equivalent performance or even outperform multi-node distributed memory parallel systems on the basis of the SBTH algorithm, depending on the utilized GPU and BLAS implementation. Our RODE solver consisting of four building blocks is the first high-performance implementation for this type of differential equations. Since numerical solvers for RODEs are computationally very expensive, such a high-performance implementation allows simulating scenarios of reasonable size for the first time. The first three building blocks generation of normal random numbers, realization of the OU process, and averaging are not limited to handle RODEs but are also generally applicable in other domains. A quasi by-product of parallelizing and optimizing the building blocks are very competitive normal PRNGs delivering up to 4.46 GPRN/s (Wallace method on the Tesla M2090) outperforming state-of-the-art normal PRNGs for the CPU (MKL) by up to 2.61× and state-of-the-art GPU libraries (cuRAND) by up to 4.53×. Another quasi by-product is the first successful parallelization of the OU process. The hybrid implementation of the LBM realized by us offers excellent scaling behavior achieving up to 2604.72GLUPS in a weak scaling scenario utilizing 2048 GPUs and 24,576 CPU cores of Piz Daint in a heterogeneous way with a parallel efficiency of 90.60%. Such good scalability promises high performance when executing bigger jobs on larger (future) clusters enabling simulations of yet unfeasible size, resolution, and numerical accuracy. By incorporating all computational resources of a heterogeneous cluster (in our case, also utilizing the CPUs besides the GPUs), a performance improvement of up to 12.85% is possible for the LBM (512 subdomains on Hydra with a GPU/CPU ratio of 85%/15%). This proves that characteristics of diverse computational devices such as peak memory bandwidth can be combined to increase performance. The implementation of the LBM is accompanied by a tailored performance model to predict performance on different architectures. It is limited to our LBM implementation but considers specific properties of it allowing a prediction accuracy with a deviation of $< 20\%$ between measured and predicted performance.

None of the presented algorithms, optimization and parallelization techniques, and best practices is limited to GPUs or GPU-equipped systems but can be generalized to numerous architectures. For example, the pipelined approach of the SBTH algorithm works on any kind of hardware architecture supporting concurrent kernel execution, the parallelization of the OU process works on parallel architectures in general, and the domain decomposition scheme and separated processing of boundary and inner cells enables scalability of Cartesian grid based methods on all kinds of heterogeneous architectures. In many cases, the wheel does not have to be reinvented to tackle challenges in the context of GPUs and heterogeneous computing: The SBTH algorithm can be completely expressed by BLAS level 1 and level 2 routines enabling the operation of existing BLAS implementations for GPUs. The parallelization of the OU process is obtained via a mapping to the prefix sum operation and the usage of existing parallelizations for this operation. The memory layout pattern of the LBM on GPUs also provides nice properties and good performance on CPUs. GPUs are not just ideal candidates for compute-intensive problems but also for tasks requiring many memory operations due to the GPU's extraordinary high memory bandwidth. Even if the FLOP/byte ratio of GPUs is much higher than for CPUs, memory-bound problems can be significantly accelerated as demonstrated for averaging and the LBM. Latency-bound problems such as the realization of the OU process benefit from the capability of GPUs to handle enormous amounts of threads to hide memory latencies. Various programming models are necessary to address the different levels of parallelism and they have to be combined to develop software for large-scale heterogeneous systems: The programming model for GPUs expresses parallelism by scalar kernels being executed in a SIMT way, (OpenMP) tasks are utilized by us to write parallel code for the CPU, and message passing is used for inter-process communication.

Even if it is not clear yet how the details of actual exaFLOPS machines look like, it is very likely that they will be some kind of heterogeneous system, e.g. augmented by GPUs. Thus, techniques on all levels of parallelism such as those ones presented in this thesis are indispensable to fully exploit such systems. This includes the programming of the multiple levels of hardware parallelism by hybrid programming as well as the introduction of new algorithmic levels of parallelism such as the pipelining of the SBTH algorithm or the Monte Carlo approach for RODEs. So this thesis discusses a small but comprehensive number of ideas. Since different scientific computing applications require different solutions for the multiple levels of parallelism issue, this topic will stay an active field of HPC research.

Bibliography

[1] Basic Linear Algebra Subprograms - A Quick Reference Guide. `http://www.netlib.org/blas/blasqr.pdf`, May 1997.

[2] clBLAS. `https://github.com/clMathLibraries/clBLAS`, 2016.

[3] hcBLAS. `https://bitbucket.org/multicoreware/hcblas`, 2016.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. `https://arxiv.org/abs/1603.04467`, March 2016.

[5] E. Agullo, J. Demmel, J. J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1), 2009.

[6] C. K. Aidun and J. R. Clausen. Lattice-Boltzmann Method for Complex Flows. *Annual Review of Fluid Mechanics*, 42(1):439–472, January 2010.

[7] AMD. AMD Accelerated Parallel Processing - OpenCL User Guide, December 2014.

[8] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, New York, New York, USA, 1967. ACM Press.

[9] E. Anderson, Z. Bai, C. H. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide.* SIAM, 3rd edition, 1999.

[10] S. Ansumali, I. V. Karlin, and H. C. Öttinger. Minimal entropic kinetic models for hydrodynamics. *Europhysics Letters (EPL)*, 63(6):798–804, September 2003.

[11] M. W. Attia, N. Maruyama, and T. Aoki. Daino: A High-Level Framework for Parallel and Efficient AMR on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '16*, pages 53:1–53:12, Salt Lake City, Utah, 2016.

[12] T. Auckenthaler. *Highly Scalable Eigensolvers for Petaflop Applications.* PhD Thesis, Technische Universität München, 2013.

[13] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. Willems. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Computing*, 37(12):783–794, December 2011.

[14] T. Auckenthaler, H.-J. Bungartz, T. Huckle, L. Krämer, B. Lang, and P. Willems. Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem. *Journal of Computational Science*, 2(3):272–278, August 2011.

[15] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A Proposal for Task Parallelism in OpenMP. In B. Chapman, W. Zheng, G. R. Gao, M. Sato, E. Ayguadé, and D. Wang, editors, *A Practical Programming Model for the Multi-Core Era*, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[16] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower, and S. Gottlieb. Scaling lattice QCD beyond 100 GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '11*, New York, New York, USA, 2011. ACM Press.

[17] M. Bach, V. Lindenstruth, O. Philipsen, and C. Pinke. Lattice QCD based on OpenCL. *Computer Physics Communications*, 184(9):2042–2052, September 2013.

[18] M. Bach, V. Lindenstruth, C. Pinke, and O. Philipsen. Twisted-mass lattice QCD using OpenCL. In *31st International Symposium on Lattice Field Theory - LATTICE 2013*, 2013.

[19] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, and M. O. Saar. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In *Proceedings of the International Conference on Parallel Processing*, pages 550–557, 2009.

[20] A. Bakhtiari. *MPI Parallelization of GPU-based Lattice Boltzmann Simulations.* Master's Thesis, Technische Universität München, 2013.

[21] G. Bal. Parallelization in time of (stochastic) ordinary differential equations. *Math. Meth. Anal. Num.*, 2003.

[22] G. M. Ballard. *Avoiding Communication in Dense Linear Algebra.* PhD Thesis, University of California, Berkeley, 2013.

[23] L. Y. Barash and L. N. Shchur. PRAND: GPU accelerated parallel random number generation library: Using most reliable algorithms and applying parallelism of modern GPUs and CPUs. *Computer Physics Communications*, 185(4):1343–1353, April 2014.

[24] E. H. Bareiss. Numerical solution of linear equations with Toeplitz and Vector Toeplitz matrices. *Numerische Mathematik*, 13(5):404–424, October 1969.

[25] P. Bastian, C. Engwer, J. Fahlke, M. Geveler, D. Göddeke, O. Iliev, O. Ippisch, R. Milk, J. Mohring, S. Müthing, M. Ohlberger, D. Ribbrock, and S. Turek. Hardware-Based Efficiency Advances in the EXA-DUNE Project. In H.-J. Bungartz, P. Neumann, and W. E. Nagel, editors, *Software for Exascale Computing - SPPEXA 2013-2015*, pages 3–23. Springer International Publishing, 2016.

[26] K.-J. Bathe and E. L. Wilson. *Numerical methods in finite element analysis.* Prentice-Hall Englewood Cliffs, NJ, 1976.

[27] J. D. Beasley and S. G. Springer. Algorithm AS 111: The Percentage Points of the Normal Distribution. *Applied Statistics*, 26(1), 1977.

[28] B. Bebee. Graph Database and Analytics in a GPU-Accelerated Cloud Offering. `http://on-demand.gputechconf.com/gtc/2016/presentation/s6395-brad-bebee-graph-database-analytics-gpu-accelerated-cloud-offering.pdf`, 2016. GPU Technology Conference 2016 (GTC'16).

[29] J. Bedorf, E. Gaburov, M. S. Fujii, K. Nitadori, T. Ishiyama, and S. P. Zwart. 24.77 Pflops on a Gravitational Tree-Code to Simulate the Milky Way Galaxy with 18600 GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '14*, pages 54–65. IEEE, November 2014.

[30] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems: Jade Edition*, pages 359–371. Elsevier, 2011.

[31] P. L. Bhatnagar, E. P. Gross, and M. Krook. A Model for Collision Processes in Gases. *Physical Review*, 94(3):511–525, May 1954.

[32] C. H. Bischof, B. Lang, and X. Sun. A framework for symmetric band reduction. *ACM Transactions on Mathematical Software*, 26(4):581–601, December 2000.

[33] C. H. Bischof, X. Sun, and B. Lang. Parallel tridiagonalization through two-step band reduction. In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 23–27. IEEE Comput. Soc. Press, 1994.

[34] G. E. Blelloch. Prefix Sums and Their Applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[35] V. Blum, R. Gehrke, F. Hanke, P. Havu, V. Havu, X. Ren, K. Reuter, and M. Scheffler. Ab initio molecular simulations with numeric atom-centered orbitals. *Computer Physics Communications*, 180(11):2175–2196, November 2009.

[36] G. Bolch. *Leistungsbewertung von Rechensystemen.* Vieweg+Teubner Verlag, Wiesbaden, 1989.

[37] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains.* John Wiley & Sons, Inc., Hoboken, NJ, USA, March 2006.

[38] G. E. P. Box and M. E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[39] J. D. Bozeman and C. Dalton. Numerical study of viscous flow in a cavity. *Journal of Computational Physics*, 12(3):348–363, July 1973.

[40] T. Bradley. GPU Performance Analysis and Optimisation. `https://people.maths.ox.ac.uk/gilesm/cuda/lecs/NV_Profiling_lowres.pdf`, 2012. GPU Technology Conference 2012 (GTC'12).

[41] P. Bratley, B. L. Fox, and L. E. Schrage. *A guide to simulation.* Springer-Verlag New York, 1983.

[42] R. P. Brent. A fast vectorised implementation of Wallace's normal random number generator. Technical report, Australian National University, April 1997.

[43] R. P. Brent. Random Number Generation and Simulation on Vector and Parallel Computers. In *Euro-Par'98 Parallel Processing*, pages 1–20. Springer, 1998.

[44] R. P. Brent. Some Comments on C. S. Wallace's Random Number Generators. *The Computer Journal*, 51(5):579–584, February 2008.

[45] R. P. Brent. Uniform and Normal Random Number Generators. `http://maths-people.anu.edu.au/~brent/random.html`, 2016.

[46] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. GPU-Accelerated Database Systems: Survey and Open Challenges. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV*, pages 1–35. Springer Berlin Heidelberg, 2014.

[47] A. L. Brophy. Approximation of the inverse normal distribution function. *Behavior Research Methods, Instruments, & Computers*, 17(3):415–417, May 1985.

[48] W. M. Brown, A. Kohlmeyer, S. J. Plimpton, and A. N. Tharrington. Implementing molecular dynamics on hybrid high performance computers – Particle–particle particle-mesh. *Computer Physics Communications*, 183(3):449–459, March 2012.

[49] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington. Implementing molecular dynamics on hybrid high performance computers – short range forces. *Computer Physics Communications*, 182(4):898–911, April 2011.

[50] J. Buchmann, D. Cabarcas, F. Göpfert, A. Hülsing, and P. Weiden. Discrete Ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8282 LNCS, pages 402–417, 2014.

[51] H. Bunke. *Gewöhnliche Differentialgleichungen mit zufälligen Parametern.* Akademie-Verlag, Berlin, 1972.

[52] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, U. Schramm, J. Schuchart, and R. Widera. Radiative signatures of the relativistic Kelvin-Helmholtz instability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '13*, pages 1–12, New York, New York, USA, 2013. ACM Press.

[53] E. Calore, D. Marchi, S. F. Schifano, and R. Tripiccione. Optimizing communications in multi-GPU Lattice Boltzmann simulations. In *2015 International Conference on High Performance Computing & Simulation (HPCS)*, pages 55–62. IEEE, July 2015.

[54] S. Chandrasekaran, M. Gu, X. Sun, J. Xia, and J. Zhu. A Superfast Algorithm for Toeplitz Systems of Linear Equations. *SIAM Journal on Matrix Analysis and Applications*, 29(4):1247–1266, January 2008.

[55] S. Chapman and T. G. Cowling. *The Mathematical Theory of Non-uniform Gases.* Cambridge University Press, 1970.

[56] S. Chen and G. D. Doolen. LATTICE BOLTZMANN METHOD FOR FLUID FLOWS. *Annual Review of Fluid Mechanics*, 30(1):329–364, January 1998.

[57] D. Cireşan, U. Meier, J. Masci, and J. Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, August 2012.

[58] K. Claessen and M. H. Pałka. Splittable pseudorandom number generators using cryptographic hashing. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell - Haskell '13*, pages 47–58, New York, New York, USA, 2013. ACM Press.

[59] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi. Solving lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications*, 181(9):1517–1528, September 2010.

[60] R. Collobert, Koray Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*, pages 1–6, 2011.

[61] S. Cook. *CUDA programming - A developer's guide to parallel computing with GPUs*. Morgan Kaufmann, Amsterdam, 2013.

[62] S. Corlay and G. Pagès. Functional quantization-based stratified sampling methods. *Monte Carlo Methods and Applications*, 21(1):1–32, January 2015.

[63] R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100(1):32–74, December 1928.

[64] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36(2):177–195, June 1980.

[65] S. Dalton, N. Bell, L. Olson, and M. Garland. CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. `https://cusplibrary.github.io/`, 2014.

[66] N. Darapaneni, P. Somawanshi, and M. Joshi. Stochastic Differential Equations simulation using GPU. In *Proceedings of International Simulation Conference of India*, 2012.

[67] C. De Schryver, D. Schmidt, N. Wehn, E. Korn, H. Marxen, and R. Korn. A new hardware efficient inversion based random number generator for non-uniform distributions. In *Proceedings - 2010 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2010*, pages 190–195, 2010.

[68] A. Debudaj-Grabysz and R. Rabenseifner. Nesting OpenMP in MPI to Implement a Hybrid Communication Method of Parallel Simulated Annealing on a Cluster of SMP Nodes. In B. Di Martino, D. Kranzlmüller, and J. J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 12th European PVM/MPI Users' Group Meeting Sorrento, Italy, September 18-21, 2005. Proceedings*, pages 18–27. Springer Berlin Heidelberg, 2005.

[69] J. Demouth. Shuffle: Tips and Tricks. `http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf`, 2013. GPU Technology Conference 2013 (GTC'13).

[70] L. Devroye. Nonuniform Random Variate Generation. In *Handbooks in Operations Research and Management Science*, volume 13, pages 83–121. Springer, 2006.

[71] D. D'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.-S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 360(1792):437–451, March 2002.

[72] J. A. Doornik. An improved Ziggurat method to generate normal random samples. Technical report, University of Oxford, 2005.

[73] F. Dubois. Equivalent partial differential equations of a lattice Boltzmann scheme. *Computers & Mathematics with Applications*, 55(7):1441–1449, April 2008.

[74] H. M. Edrees, B. Cheung, M. Sandora, D. Nummey, and S. Deian. Hardware-Optimized Ziggurat Algorithm for High-Speed Gaussian Random Number Generators. In *International Conference on Engineering of Reconfigurable Systems & Algorithms, ERSA*, pages 254–260, 2009.

[75] R. G. Edwards and B. Joó. The Chroma Software System for Lattice QCD. *Nuclear Physics B - Proceedings Supplements*, 140:832–834, March 2005.

[76] A. Fakhari and T. Lee. Finite-difference lattice Boltzmann method with a block-structured adaptive-mesh-refinement technique. *Physical Review E*, 89(3):033310, March 2014.

[77] C. Feichtinger. *Design and Performance Evaluation of a Software Framework for Multi-Physics Simulations on Heterogeneous Supercomputers.* PhD Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2012.

[78] C. Feichtinger, J. Habich, H. Köstler, U. Rüde, and T. Aoki. Performance Modeling and Analysis of Heterogeneous Lattice Boltzmann Simulations on CPU-GPU Clusters. *Parallel Computing*, 46:1–13, 2014.

[79] J. Foley. Lattice QCD using MILC and QUDA. `http://on-demand.gputechconf.com/gtc/2014/presentations/S4641-lattice-qcd-milc-quda.pdf`, 2014. GPU Technology Conference 2014 (GTC'14).

[80] A. Gaikwad and I. M. Toke. GPU based sparse grid technique for solving multi-dimensional options pricing PDEs. In *Proceedings of the 2nd Workshop on High Performance Computational Finance - WHPCF '09*, pages 1–9, New York, New York, USA, 2009. ACM Press.

[81] M. J. Gander. 50 Years of Time Parallel Time Integration. In T. Carraro, M. Geiger, S. Körkel, and R. Rannacher, editors, *Multiple Shooting and Time Domain Decomposition Methods*, pages 69–113. Springer International Publishing, 2015.

[82] P. C. Gao, Y. B. Tao, Z. H. Bai, and H. Lin. Mapping the SBR and TW-ILDCs to heterogeneous CPU-GPU architecture for fast computation of electromagnetic scattering. *Progress In Electromagnetics Research*, 122:137–154, 2012.

[83] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL.* Morgan Kaufmann, 2nd edition, 2012.

[84] M. Geier and M. Schönherr. Esoteric Twist: An Efficient in-Place Streaming Algorithmus for the Lattice Boltzmann Method on Massively Parallel Hardware. *Computation*, 5(2), March 2017.

[85] J. E. Gentle. *Random number generation and Monte Carlo methods.* Springer Science & Business Media, 1998.

[86] U. Ghia, K. N. Ghia, and C. T. Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 48(3):387–411, December 1982.

[87] D. Gillespie. Exact numerical simulation of the Ornstein-Uhlenbeck process and its integral. *Physical Review E*, 54(2):2084–2091, 1996.

[88] I. Ginzburg, F. Verhaeghe, and D. D'Humières. Two-relaxation-time Lattice Boltzmann scheme: About parametrization, velocity, pressure and mixed boundary conditions. *Communications in Computational Physics*, 3(2):427–478, 2008.

[89] Global Scientific Information and Computing Center. TSUBAME2.5 Hardware Software Specifications. Technical report, Tokyo Institute of Technology, Tokyo, 2013.

[90] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* The MIT Press, 2016.

[91] M. J. Grote and T. Huckle. Parallel Preconditioning with Sparse Approximate Inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, May 1997.

[92] L. Grune and P. E. Kloeden. Pathwise Approximation of Random Ordinary Differential Equations. *Bit Numerical Mathematics*, 41(4):711–721, 2001.

[93] J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein. Performance engineering for the lattice Boltzmann method on GPGPUs: Architectural requirements and performance results. *Computers & Fluids*, 80:276–282, July 2013.

[94] R. D. Hagan. *Multi-GPU Load Balancing for Simulation and Rendering.* Master's thesis, Virginia Polytechnic Institute and State University, 2011.

[95] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* Chapman & Hall/CRC Computational Science. CRC Press, July 2010.

[96] A. Haidar, R. Solcà, M. Gates, S. Tomov, T. C. Schulthess, and J. J. Dongarra. Leading Edge Hybrid Multi-GPU Algorithms for Generalized Eigenproblems in Electronic Structure Calculations. In J. M. Kunkel, T. Ludwig, and H. W. Meuer, editors, *Supercomputing: 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*, pages 67–80. Springer Berlin Heidelberg, 2013.

[97] M. J. Hallock, J. E. Stone, E. Roberts, C. Fry, and Z. Luthey-Schulten. Simulation of reaction diffusion processes over biologically relevant size and time scales using multi-GPU workstations. *Parallel Computing*, 40(5-6):86–99, 2014.

[98] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing – HiPC 2007*, pages 197–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[99] M. Harris and M. Garland. Optimizing Parallel Prefix Operations for the Fermi Architecture. In *GPU Computing Gems: Jade Edition*, pages 29–38. Elsevier, 2011.

[100] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, pages 1–24. Addison-Wesley Professional, 2007.

[101] M. J. Harvey and G. De Fabritiis. A survey of computational molecular science using graphics processing units. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 2(5):734–742, September 2012.

[102] X. He and L.-S. Luo. Lattice Boltzmann Model for the Incompressible Navier–Stokes Equation. *Journal of Statistical Physics*, 88(3/4):927–944, August 1997.

[103] X. He and L.-S. Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811–6817, December 1997.

[104] C. Hirsch. *Numerical computation of internal and external flows: The fundamentals of computational fluid dynamics.* Butterworth-Heinemann, 2007.

[105] W. G. Horner. A New Method of Solving Numerical Equations of All Orders, by Continuous Approximation. *Philosophical Transactions of the Royal Society of London*, 109:308–335, 1819.

[106] G. W. Housner and P. C. Jennings. Generation of Artificial Earthquakes. *Journal of the Engineering Mechanics Division*, 90(1):113–152, 1964.

[107] D. Hänel. *Molekulare Gasdynamik - Einführung in die kinetische Theorie der Gase und Lattice-Boltzmann-Methoden.* Springer-Verlag, Berlin/Heidelberg, 1 edition, 2004.

[108] K. Iglberger and U. Rüde. Massively parallel granular flow simulations with non-spherical particles. *Computer Science - Research and Development*, 25(1-2):105–113, May 2010.

[109] P. Imkeller and C. Lederer. The Cohomology of Stochastic and Random Differential Equations, and Local Lineraization of Stochastic Flows. *Stochastics and Dynamics*, 2(2):131–159, 2002.

[110] P. Imkeller and B. Schmalfuss. The Conjugacy of Stochastic and Random Differential Equations and the Existence of Global Attractors. *Journal of Dynamics and Differential Equations*, 13(2):215–249, 2001.

[111] Intel Corporation. Benchmarks for Intel Math Kernel Library. `https://software.intel.com/en-us/intel-mkl/benchmarks`, 2016.

[112] M. Januszewski and M. Kostur. Accelerating numerical solution of stochastic differential equations with CUDA. *Computer Physics Communications*, 181(1):183–188, 2010.

[113] E. T. Jaynes. Bayesian Methods: General Background. In J. H. Justice, editor, *Maximum Entropy and Bayesian Methods in Applied Statistics*, pages 1–25, Cambridge, 1986. Cambridge University Press.

[114] A. Jentzen and P. E. Kloeden. *Taylor Approximations for Stochastic Partial Differential Equations*. SIAM Press, 2011.

[115] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the ACM International Conference on Multimedia - MM '14*, pages 675–678, New York, New York, USA, 2014. ACM Press.

[116] K. Kanai. Semi-Empirical Formula for the Characteristics of the Ground. *Bulletin of the Earthquake Research Institute*, 35:307–325, 1957.

[117] K. Kanai. An Empirical Formula for the Spectrum of Strong Earthquake Motions. *Bulletin of the Earthquake Research Institute*, 39:85–95, 1961.

[118] I. Karatzas and S. Shreve. *Brownian Motion and Stochastic Calculus*. Springer, 2nd edition, 1991.

[119] L. Kaufman. Banded Eigenvalue Solvers on Vector Machines. *ACM Transactions on Mathematical Software*, 10(1):73–85, January 1984.

[120] L. Kaufman. Band reduction algorithms revisited. *ACM Transactions on Mathematical Software*, 26(4):551–567, December 2000.

[121] D. Kirk and W.-m. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufman, 2nd edition, 2012.

[122] P. E. Kloeden and A. Jentzen. Pathwise convergent higher order numerical schemes for random ordinary differential equations. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 463(2087):2929–2944, 2007.

[123] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997.

[124] C. Körner, M. Thies, T. Hofmann, N. Thürey, and U. Rüde. Lattice Boltzmann Model for Free Surface Flow for Modeling Foaming. *Journal of Statistical Physics*, 121(1-2):179–196, October 2005.

[125] H. Köstler, D. Ritter, and C. Feichtinger. A Geometric Multigrid Solver on GPU Clusters. In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 407–422. Springer Berlin Heidelberg, 2013.

[126] R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27(4):831–838, 1980.

[127] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems - ASPLOS-IV*, pages 63–74, New York, New York, USA, 1991. ACM Press.

[128] B. Lang. A Parallel Algorithm for Reducing Symmetric Banded Matrices to Tridiagonal Form. *SIAM Journal on Scientific Computing*, 14(6):1320–1338, 1993.

[129] B. Lang. *Effiziente Orthogonaltransformationen bei der Eigen- und Singulärwertzerlegung.* Habilitation Thesis, Bergische Universität Wuppertal, 1997.

[130] H. P. Langtangen. Numerical Solution of First Passage Problems in Random Vibrations. *SIAM Journal on Scientific Computing*, 15(4):977–996, July 1994.

[131] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, April 2009.

[132] P. L'Écuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, 1990.

[133] P. L'Écuyer. Uniform random number generation. *Annals of Operations Research*, 53(1):77–120, 1994.

[134] P. L'Écuyer, D. Munger, B. Oreshkin, and R. Simard. Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs. *Mathematics and Computers in Simulation*, 135:3–17, May 2016.

[135] P. L'Écuyer and R. Simard. A Software Library in ANSI C for Empirical Testing of Random Number Generators. Technical report, Département d'Informatique et de Recherche Opérationnelle Université de Montréal, 2002.

[136] P. L'Écuyer and R. Simard. TestU01. *ACM Transactions on Mathematical Software*, 33(4), 2007.

[137] D.-U. Lee, W. Luk, J. D. Villasenor, G. Zhang, and P. H. W. Leong. A hardware Gaussian noise generator using the Wallace method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(8):911–919, August 2005.

[138] P. H. W. Leong, G. Zhang, D.-U. Lee, W. Luk, and J. D. Villasenor. A Comment on the Implementation of the Ziggurat Method. *Journal of Statistical Software*, 12(7):1–4, 2005.

[139] D. Li and M. Becchi. Deploying Graph Algorithms on GPUs: An Adaptive Solution. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1013–1024. IEEE, May 2013.

[140] Y. K. Lin and G. Q. Cai. *Probabilistic Structural Dynamics.* McGraw-Hill, 2004.

[141] J. Linxweiler. *Ein integrierter Softwareansatz zur interaktiven Exploration und Steuerung von Strömungssimulationen auf Many-Core-Architekturen.* Phd thesis, Technische Universität Braunschweig, 2011.

[142] F. Lu, J. Song, F. Yin, and X. Zhu. Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters. *Computer Physics Communications*, 183(6):1172–1181, 2012.

[143] M. G. Luby. *Pseudorandomness and cryptographic applications*, volume 31. Princeton University Press, 1996.

[144] J. Luitjens. CUDA Pro Tip: Increase Performance with Vectorized Memory Access. `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/`, 2013.

[145] J. Luitjens. Faster Parallel Reductions on Kepler. `https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/`, 2014.

[146] D. Lukarski and N. Trost. PARALUTION User Manual. `http://www.paralution.com/downloads/paralution-um.pdf`, 2016.

[147] T. Luu. Efficient and Accurate Parallel Inversion of the Gamma Distribution. *SIAM Journal on Scientific Computing*, 37(1):C122–C141, January 2015.

[148] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *2011 International Conference on Parallel Processing*, pages 176–185. IEEE, September 2011.

[149] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer. The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter*, 26(21), May 2014.

[150] A. Marek and H. Lederer. Recent optimizations of ELPA eigensolvers. `http://www.mpcdf.mpg.de/about-mpcdf/publications/bits-n-bytes?BB-View=194&BB-Doc=173`, 2016.

[151] A. A. Markov. Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete*, 15:135–156, 1906.

[152] G. Marsaglia. Generating a variable from the tail of the normal distribution. *Technometrics*, 6(1):101–102, 1964.

[153] G. Marsaglia. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. `http://stat.fsu.edu/pub/diehard/`, 1995.

[154] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8:1–6, 2003.

[155] G. Marsaglia and T. A. Bray. A Convenient Method for Generating Normal Variables. *SIAM Review*, 6(3):260–264, 1964.

[156] G. Marsaglia and W. W. Tsang. A Fast, Easily Implemented Method for Sampling from Decreasing or Symmetric Unimodal Density Functions. *SIAM Journal on Scientific and Statistical Computing*, 5(2):349–359, 1984.

[157] G. Marsaglia and W. W. Tsang. The Ziggurat Method for Generating Random Variables. *Journal of Statistical Software*, 5(8), 2000.

[158] N. Maruyama and T. Aoki. Optimizing Stencil Computations for NVIDIA Kepler GPUs. In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*, pages 89–95, 2014.

[159] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[160] R. Mei, W. Shyy, D. Yu, and L.-S. Luo. Lattice Boltzmann Method for 3-D Flows with Curved Boundary. *Journal of Computational Physics*, 161(2):680–699, July 2000.

[161] D. Merrill, M. Garland, and A. Grimshaw. High-Performance and Scalable GPU Graph Traversal. *ACM Transactions on Parallel Computing*, 1(2):1–30, February 2015.

[162] P. Micikevicius. Local Memory and Register Spilling. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf`, 2011. GPU Technology Conference 2011 (GTC'11).

[163] V. Minden, B. Smith, and M. G. Knepley. Preliminary Implementation of PETSc Using GPUs. In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 131–140. Springer Berlin Heidelberg, 2013.

[164] S. Mittal and J. S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys*, 47(4):1–35, July 2015.

[165] J.-i. Muramatsu, T. Fukaya, S.-L. Zhang, K. Kimura, and Y. Yamamoto. Acceleration of Hessenberg Reduction for Nonsymmetric Eigenvalue Problems in a Hybrid CPU-GPU Computing Environment. *International Journal of Networking and Computing*, 1(2):132–143, 2011.

[166] A. F. Muraraşu. *Advanced Optimization Techniques for Sparse Grids on Modern Heterogeneous Systems.* PhD Thesis, Technische Universität München, 2013.

[167] A. F. Muraraşu, J. Weidendorfer, G. Buse, D. Butnaru, and D. Pflüger. Compact data structure and scalable algorithms for the sparse grid technique. *ACM SIGPLAN Notices*, 46(8), September 2011.

[168] K. Murata and K. Horikoshi. A New Method for the Tridiagonalization of the Symmetric Band Matrix. *Inf. Proc. Japan*, 15:108–112, 1975.

[169] Y. Nakamura and H. Stüben. BQCD - Berlin quantum chromodynamics program. Technical report, Konrad-Zuse-Zentrum für Informationstechnik Berlin, October 2010.

[170] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka. AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, January 2015.

[171] C.-L. Navier. Mémoire sur les lois du mouvement des fluides. *Mem. Acad. Sci. Inst. France*, 6:389–416, 1823.

[172] T. Neckel, A. Parra Hinojosa, and F. Rupp. Path-Wise Algorithms for Random & Stochastic ODEs with Applications to Ground-Motion-Induced Excitations of Multi-Storey Buildings. Technical Report TUM-I1758, Technische Universität München, 2017.

[173] T. Neckel and F. Rupp. *Random Differential Equations in Scientific Computing.* Versita, De Gruyter publishing group, Warsaw, 2013.

[174] P. Neumann. *Hybrid Multiscale Simulation Approaches For Micro- and Nanoflows.* PhD Thesis, Technische Universität München, 2013.

[175] P. Neumann and T. Neckel. A dynamic mesh refinement technique for Lattice Boltzmann simulations on octree-like grids. *Computational Mechanics*, 51(2):237–253, February 2013.

[176] NVIDIA Corporation. Achieved Occupancy. `https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm`, 2015.

[177] NVIDIA Corporation. Tuning CUDA applications for Kepler. `http://docs.nvidia.com/cuda/kepler-tuning-guide/`, 2015.

[178] NVIDIA Corporation. Tuning CUDA applications for Maxwell. `http://docs.nvidia.com/cuda/maxwell-tuning-guide/`, 2015.

[179] NVIDIA Corporation. *cuBLAS*, 8.0 edition, September 2016.

[180] NVIDIA Corporation. *CUDA C Programming Guide*, 8.0 edition, September 2016.

[181] NVIDIA Corporation. cuRAND Library Programming Guide. `http://docs.nvidia.com/cuda/curand/`, 2016.

[182] NVIDIA Corporation. *Profiler User's Guide*, 8.0 edition, September 2016.

[183] NVIDIA Corporation. cuRAND. `https://developer.nvidia.com/curand`, January 2017.

[184] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers & Mathematics with Applications*, 61(12):3628–3638, June 2011.

[185] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Multi-GPU implementation of the lattice Boltzmann method. *Computers & Mathematics with Applications*, 65(2):252–261, January 2013.

[186] S. Páll and B. Hess. A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications*, 184(12):2641–2650, December 2013.

[187] L. G. Paparizos. Some observations on the random response of hysteretic systems. Technical report, California Institute of Technology, 1986.

[188] A. Parra Hinojosa and T. Neckel. K-RODE-Taylor schemes of order 3 and 4 for the Kanai-Tajimi earthquake model. Technical Report TUM-I1524, Technische Universität München, 2015.

[189] M. Petschow and P. Bientinesi. MR3-SMP: A symmetric tridiagonal eigensolver for multi-core architectures. *Parallel Computing*, 37(12):795–805, December 2011.

[190] O. Philipsen, C. Pinke, A. Sciarra, and M. Bach. CL2QCD - Lattice QCD based on OpenCL. In *The 32nd International Symposium on Lattice Field Theory*, November 2014.

[191] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kale. NAMD: Biomolecular Simulation on Thousands of Processors. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 2002.

[192] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436. IEEE, 2009.

[193] A. Rahimian, I. Lashuk, S. K. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. S. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200K cores and

heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '10*, 2010.

[194] D. A. Reed, R. Bajcsy, M. A. Fernandez, J.-M. Griffiths, R. D. Mott, J. J. Dongarra, C. R. Johnson, A. S. Inouye, W. Miner, M. K. Matzke, and T. L. Ponick. Computational science: Ensuring America's competitiveness. Technical report, National Coordination Office for Information Technology Research and Development, 2005.

[195] S. Rennich. CUDA C/C++ Streams and Concurrency. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf`, 2011. GPU Technology Conference 2011 (GTC'11).

[196] W. A. Richards, R. Antoine, A. Sahai, and M. R. Acharya. An Efficient Polynomial Approximation to the Normal Distribution Function and Its Inverse Function. *Journal of Mathematics Research*, 2(4), October 2010.

[197] C. Riesinger and T. Neckel. A runtime/memory trade-off of the continuous Ziggurat method on GPUs. In *2015 International Conference on High Performance Computing & Simulation (HPCS)*, pages 27–34. IEEE, July 2015.

[198] C. Riesinger, T. Neckel, and F. Rupp. Non-standard Pseudo Random Number Generators revisited for GPUs. *Future Generation Computer Systems*, 2016.

[199] C. Riesinger, T. Neckel, and F. Rupp. Solving Random Ordinary Differential Equations on GPU Clusters using Multiple Levels of Parallelism. *SIAM Journal on Scientific Computing*, 38(4):C372–C402, July 2016.

[200] C. Riesinger, T. Neckel, F. Rupp, A. Parra Hinojosa, and H.-J. Bungartz. GPU Optimization of Pseudo Random Number Generators for Random Ordinary Differential Equations. In *2014 International Conference on Computational Science*, volume 29, pages 172–183. Elsevier, 2014.

[201] C. Rogers and D. Williams. *Diffusions, Markov processes and martingales - Volume 2: Itô calculus.* Cambridge University Press, 2nd edition, 2000.

[202] M. Rohde, D. Kandhai, J. J. Derksen, and H. E. A. van den Akker. A generic, mass conservative local grid refinement technique for lattice-Boltzmann schemes. *International Journal for Numerical Methods in Fluids*, 51(4):439–468, June 2006.

[203] D. Rohr. The L-CSC cluster: An AMD-GPU-based cost- and power-efficient multi-GPU system for Lattice-QCD calculations at GSI. `https://www.top500.org/files/green500/SC14-bof-lcsc.pdf`, 2014.

[204] D. Rossinelli, G. Karniadakis, M. Fatica, I. Pivkin, P. Koumoutsakos, Y.-H. Tang, K. Lykov, D. Alexeev, M. Bernaschi, P. Hadjidoukas, M. Bisson, W. Joubert, and

C. Conti. The in-silico lab-on-a-chip: petascale and high-throughput simulations of microfluidics at cell resolution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '15*, pages 1–12, New York, New York, USA, 2015. ACM Press.

[205] U. Rüde and N. Thürey. Free surface lattice-Boltzmann fluid simulations with and without level sets. In *Proceedings of Vision, Modeling and Visualization*, pages 199–208, 2004.

[206] K. Rupp. The High-Level Linear Algebra Library ViennaCL and Its Applications. `http://on-demand.gputechconf.com/gtc/2012/presentations/S0071-High-Level-Linear-Algebra-Library-ViennaCL-Apps.pdf`, 2012. GPU Technology Conference 2012 (GTC'12).

[207] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '11*, pages 1–12, 2011.

[208] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, January 2015.

[209] M. Schreiber. *GPU based simulation and visualization of fluids with free surfaces.* Diploma Thesis, Technische Universität München, 2010.

[210] M. Schreiber, P. Neumann, S. Zimmer, and H.-J. Bungartz. Free-Surface Lattice-Boltzmann Simulation on Many-Core Architectures. In *Procedia Computer Science*, volume 4, pages 984–993, 2011.

[211] Z. Schuss. *Theory and Applications of Stochastic Processes.* Springer-Verlag, 2010.

[212] H. R. Schwarz. Algorithm 183: reduction of a symmetric bandmatrix to triple diagonal form. *Communications of the ACM*, 6(6):315–316, June 1963.

[213] H. R. Schwarz. Tridiagonalization of a symetric band matrix. *Numerische Mathematik*, 12(4):231–241, November 1968.

[214] M. Schönherr, K. Kucher, M. Geier, M. Stiebler, S. Freudiger, and M. Krafczyk. Multi-thread implementations of the lattice Boltzmann method on non-uniform grids for CPUs and GPUs. *Computers & Mathematics with Applications*, 61(12):3730–3743, June 2011.

[215] V. Sellappan and B. Desam. Accelerating ANSYS Fluent simulations with NVIDIA GPUs. `http://resource.ansys.com/staticassets/ANSYS/staticassets/resourcelibrary/article/Accelerationg-ANSYS-Fluent-Simulations-with-NVIDIA-GPUs-AA-V9-I1.pdf`, 2015.

[216] S. Sengupta, M. Harris, M. Garland, and J. D. Owens. Efficient Parallel Scan Algorithms for Many-core GPUs. In *Scientific Computing with Multicore and Accelerators*, pages 413–442. Chapman & Hall/CRC Computational Science, 2011.

[217] X. Shan and H. Chen. Lattice Boltzmann model for simulating flows with multiple phases and components. *Physical Review E*, 47(3):1815–1819, March 1993.

[218] J. Shen, A. L. Varbanescu, H. Sips, M. Arntzen, and D. G. Simons. Glinda: A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms. In *Proceedings of the ACM International Conference on Computing Frontiers - CF '13*, New York, New York, USA, 2013. ACM Press.

[219] T. Shimokawabe, T. Aoki, T. Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, and S. Matsuoka. Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '11*, pages 1–11, 2011.

[220] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[221] R. Smith. Retrofit of Legacy MPI CFD System with GPU Acceleration. `http://on-demand.gputechconf.com/gtc/2013/presentations/S3181-MPI-CFD-Retrofit-with-GPU-Acceleration.pdf`, 2013. GPU Technology Conference 2013 (GTC'13).

[222] R. C. Smith. *Uncertainty Quantification: Theory, Implementation, and Applications.* SIAM, 2013.

[223] J. E. Stone, R. McGreevy, B. Isralewitz, and K. Schulten. GPU-accelerated analysis and visualization of large structures solved by molecular dynamics flexible fitting. *Faraday Discuss.*, 169:265–283, March 2014.

[224] S. Succi. *The Lattice Boltzmann equation: for fluid dynamics and beyond.* Oxford University Press, 2013.

[225] H. J. Sussmann. On the Gap Between Deterministic and Stochastic Ordinary Differential Equations. *The Annals of Probability*, 6(1):19–41, February 1978.

[226] M. R. Swift, E. Orlandini, W. R. Osborn, and J. M. Yeomans. Lattice Boltzmann simulations of liquid-gas and binary fluid systems. *Physical Review E*, 54(5):5041–5052, November 1996.

[227] H. Tajimi. A Statistical Method of Determining the Maximum Response of a Building Structure During an Earthquake. In *2nd World Conference on Earthquake Engineering*, volume 2, pages 781–798, 1960.

[228] D. B. Thomas, L. Howes, and W. Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '09*, pages 63–72, 2009.

[229] D. B. Thomas, W. Luk, P. H. W. Leong, and J. D. Villasenor. Gaussian Random Number Generators. *ACM Computing Surveys*, 39(4), 2007.

[230] N. Thürey. *Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method.* Phd thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2007.

[231] J. Tölke, S. Freudiger, and M. Krafczyk. An adaptive scheme using hierarchical grids for lattice Boltzmann multi-phase flow simulations. *Computers & Fluids*, 35(8-9):820–830, September 2006.

[232] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, August 2008.

[233] P. Valero-Lara, F. D. Igual, M. Prieto-Matías, A. Pinelli, and J. Favier. Accelerating fluid–solid simulations (Lattice-Boltzmann & Immersed-Boundary) on heterogeneous architectures. *Journal of Computational Science*, 10:249–261, September 2015.

[234] F. G. van Zee and R. A. van de Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software*, 41(3):1–33, June 2015.

[235] V. Volkov. Better Performance at Lower Occupancy. `http://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf`, 2010. GPU Technology Conference 2010 (GTC'10).

[236] M. Wahib and N. Maruyama. Data-centric GPU-based adaptive mesh refinement. In *Proceedings of the 5th Workshop on Irregular Applications Architectures and Algorithms - IA3 '15*, pages 1–7, New York, New York, USA, 2015. ACM Press.

[237] C. S. Wallace. Fast Pseudorandom Generators for Normal and Exponential Variates. *ACM Transactions on Mathematical Software*, 22(1):119–127, 1996.

[238] M. Wang, B. Wang, Q. He, X. Liu, and K. Zhu. Analysis of GPU Parallel Computing based on Matlab. *CoRR*, abs/1505.0, May 2015.

[239] X. Wang and T. Aoki. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing*, 37(9):521–535, February 2011.

[240] Y. Wang, H. Du, M. Xia, L. Ren, M. Xu, T. Xie, G. Gong, N. Xu, H. Yang, and Y. He. Correction: A Hybrid CPU-GPU Accelerated Framework for Fast Mapping of High-Resolution Human Brain Connectome. *PLoS ONE*, 8(9), September 2013.

[241] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*, 35(8-9):910–919, September 2006.

[242] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby. A tensorial approach to computational continuum mechanics using object-oriented techniques. *Computers in Physics*, 12(6):620–631, 1998.

[243] B. Wescott and A. Macijeski. *Every Computer Performance Book.* CreateSpace Independent Publishing Platform, 1 edition, 2013.

[244] E. Westphal. Voting and Shuffling to Optimize Atomic Operations. `https://devblogs.nvidia.com/parallelforall/voting-and-shuffling-optimize-atomic-operations/`, 2015.

[245] M. J. Wichura. Algorithm AS241: The percentage points of the normal distribution. *Applied Statistics*, 37:477–484, 1988.

[246] S. W. Williams. The Roofline Model. In D. H. Bailey, R. F. Lucas, and S. W. Williams, editors, *Performance Tuning of Scientific Applications*, pages 195–216. CRC Press, 2010.

[247] S. W. Williams, A. Waterman, and D. Patterson. Roofline: An Insight Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4), April 2009.

[248] N. Wilt. *The CUDA Handbook - A Comprehensive Guide to GPU Programming.* Addison-Wesley, 2013.

[249] F. T. Winter, M. A. Clark, R. G. Edwards, and B. Joó. A Framework for Lattice QCD Calculations on GPUs. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1073–1082. IEEE, May 2014.

[250] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein. Comparison of different propagation steps for lattice Boltzmann methods. *Computers & Mathematics with Applications*, 65(6):924–935, March 2013.

[251] D. A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models - An Introduction.* Springer, Berlin, 2000.

[252] X. Q. Xing, D. L. Butler, and C. Yang. Lattice Boltzmann-based single-phase method for free surface tracking of droplet motions. *International Journal for Numerical Methods in Fluids*, 53(2):333–351, January 2007.

[253] Q. Xiong, B. Li, J. Xu, X. Fang, X. Wang, L. Wang, X. He, and W. Ge. Efficient parallel implementation of the lattice Boltzmann method on large clusters of graphic processing units. *Chinese Science Bulletin*, 57(7):707–715, March 2012.

[254] I. Yamazaki, T. Dong, R. Solcà, S. Tomov, J. J. Dongarra, and T. C. Schulthess. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *Concurrency and Computation: Practice and Experience*, 26(16):2652–2666, November 2014.

[255] B. Zhang, S. Xu, F. Zhang, Y. Bi, and L. Huang. Accelerating MatLab code using GPU: A review of tools and strategies. In *Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2011 2nd International Conference on*, pages 1875–1878. IEEE, 2011.

[256] G. Zhang, P. H. W. Leong, D.-U. Lee, J. D. Villasenor, R. C.-C. Cheung, and W. Luk. Ziggurat-based hardware Gaussian random number generator. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 275–280, 2005.

[257] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):126:1–126:11, 2008.